Techniques for the Dynamic Randomization of Network Attributes

Adrian R. Chavez and William M.S. Stout
Sandia National Laboratories
Albuquerque, New Mexico, USA
{adrchav,wmstout}@sandia.gov

Sean Peisert
Lawrence Berkeley National Laboratory
& University of California, Davis
California, USA
sppeisert@lbl.gov

***Abstract*— Critical infrastructure control systems continue to foster predictable communication paths and static configurations that allow easy access to our networked critical infrastructure around the world. This makes them attractive and easy targets for cyber-attack. We have developed technologies that address these attack vectors by automatically reconfiguring network settings. Applying these protective measures will convert control systems into "moving targets" that proactively defend themselves against attack. This "Moving Target Defense" (MTD) revolves about the movement of network reconfiguration, securely communicating reconfiguration specifications to other network nodes as required, and ensuring that connectivity between nodes is uninterrupted. Software-defined Networking (SDN) is leveraged to meet many of these goals. Our MTD approach eliminates adversaries targeting known static attributes of network devices and systems, and consists of the following three techniques: (1) Network Randomization for TCP/UDP Ports; (2) Network Randomization for IP Addresses; (3) Network Randomization for Network Paths**

In this paper, we describe the implementation of the aforementioned technologies. We also discuss the individual and collective successes for the techniques, challenges for deployment, constraints and assumptions, and the performance implications for each technique.

Keywords—Moving Target Defense, Software Defined Networking, Computer Security, IP Address Hopping, Dynamic Defense

I. INTRODUCTION

Computer networks, in particular Critical Infrastructure (CI) systems, continue to foster predictable communication paths and static configurations [1] that provide a vector for accessing the critical assets of a network. Among the many vulnerabilities that CI systems possess, a static configuration can make CI systems attractive and easy targets for cyber-attack. In our research, we provide a means to address these attack vectors by automatically reconfiguring network settings and randomizing application communications dynamically. Applying these protective measures will convert CI systems into "moving targets" that proactively defend themselves against attack.

Our MTD architecture is comprised of several techniques to manage network reconfiguration, ensuring that connectivity between nodes is uninterrupted, while providing randomization at the application and network levels. The goal of our research is to significantly reduce the class of adversaries able to rely on known static Internet Protocol (IP) addresses of CI network devices to launch an attack. Our approach introduces uncertainty and unpredictability for adversaries reconnoitering a network to determine the function of nodes on computer networks, and also for adversaries attempting to map the topology of computer networks. It is comprised of the following techniques:

- Network Randomization for TCP/UDP Ports: our TCP/UDP port hopper is implemented at the host level. The Port Hopper follows a distributed system model that relies on time synchronization for all nodes communicating in the network. Communications between endpoints "hop" between network port numbers at specified intervals of time that are configurable. Those intervals include tolerance thresholds to account for delays incurred during transmission and traversal.

- Network Randomization for IP Addresses: the implementation of our IP Address-Randomizer consists of an SDN controller that manages communication among network devices that pass IP-addressed traffic. As traffic ingresses the network-layer device, its source-destination pairs are validated and rules are installed to encode/decode randomized IP addresses. The duration of these rules can be made static, random, or be forced to change from a trusted third-party. Since the IP Address-randomizer is implemented at the network-layer, it is transparent to the communicating endpoint processes, and may be implemented without endpoint modification.

- Network Randomization for Network Paths: the Network-Path Randomizer uses overlay networks to impede traffic analysis, given that traffic analysis is a technique often used by an adversary to identify endpoints. The underlying network consists of several nodes that form a physical mesh topology. Routing through the mesh is coordinated among the nodes via an SDN controller. For each flow through the network, the controller may assign asymmetric forward/reverse paths, with the ability to modify them given specified parameters (time, bitrate, etc). This approach also improves network resiliency to eavesdropping attacks and denial of service attacks by providing multiple possible communications paths between nodes.

The rest of this paper is organized as follows: Section 2 describes some of the past MTD research; Section 3 describes the threat model and the restrictions we place on the adversary; Section 4 describes our implementation of the MTD approaches; Section 5 outlines our experiment setup and results; and finally Section 6 concludes the paper, summarizing our results, some of the limitations of our approach, and some of our future areas of research.

II. Related Work

A number of related approaches to mitigating attacks over a network exist.

Anonymization of network traffic is an active area of research with several implementations available in both the commercial and open source communities. For example, onion routing [2] depends on the use of an overlay network made up of "onion routers." The onion routers are responsible for cryptographically removing one layer of an encrypted packet at a time to determine the next hop routing information and forwarding each packet to their final destinations. The weaknesses are that side channel attacks exist including timing attacks [3], packet counting attacks [4], and intersection attacks [5] that can reveal the source and destination nodes of a communication stream. Similarly, garlic routing [6] combines and anonymizes multiple messages in a single packet but is also susceptible to the same attacks.

Tor is one of the most popular implementations used for onion routing with over 2.25 million users [7]. However, it has been shown empirically with the aid of NetFlow data, that Tor traffic can be de-anonymized with accuracy rates of 81.4% [8]. The results are achieved by correlating traffic between entry and exit points within the Tor network to determine the endpoints in communication.

While onion routing is useful for anonymization, it has been shown that overlay networks can be used to mitigate Distributed Denial of Service (DDoS) attacks [9]. The overlay networks reroute traffic through a series of hops that change over time to prevent traffic analysis, and thus prevent targeted DDoS attacks. In order for users to connect to the secure overlay network, they must first know and communicate with the secure overlay access points within the network. The knowledge of the overlay nodes prevents external adversaries from attacking nodes on the network directly. This design can be improved by relaxing the requirement of hiding the secure overlay access points within the network architecture.

Artificial diversity is another active area of research that is used to defend computer systems from attack. Introducing artificial diversity into the Internet Protocol (IP) layer has been demonstrated to work through Software Defined Networking (SDN) [10]. Flows, (based on incoming port, outgoing port, and incoming and outgoing Media Access Control (MAC) addresses) are introduced into software-defined switches. The rules for a packet that match a given flows parameters are rewritten with random source and destination IP addresses while the packet is in flight within the network. The drawback is that through traffic analysis, endpoints of the communication stream can still be learned. A passive adversary can observe traffic and correlate which flows map to which endpoints.

Steganography can also be used to hide and covertly communicate information between multiple parties within a network. The methods described in current literature [11] include the use of IPv4 header fields and reordering IPsec packets to transmit information covertly. The described approach would have to be refined to increase the amount of information ($\log_2 n!$ bits that can be communicated through n packets) that can be covertly communicated if a significant amount of information is desired to be exchanged.

Transparently anonymizing IP-based network traffic is a promising solution that leverages Virtual Private Networks (VPNs) and Tor [12]. Tor hides the users true IP address with the use of a Virtual Anonymous Network (VAN) while the VPN provides the anonymous IP addresses. The challenge for this solution is that every host must possess client software *and* a VPN cryptographic key installed, which hinders the scalability of this approach. To reduce the burden on larger scale networks, it may be more effective to integrate this approach into the network-level using an SDN-based approach.

Some combination of the benefits provided by these approaches is necessary to provide full network anonymity, which we believe could significantly improve security for time critical systems. Critical infrastructure systems fit this need and currently have no single solution existing to counter the reconnaissance phase of an attack. Some of the areas of prior work that need to be addressed in order to make their application feasible include reducing the overhead costs of multiple layers of encryption, the ability to easily scale up to a large number of nodes, and relaxing the requirement to hide nodes participating in the anonymous service. Additionally, critical infrastructure systems are composed of both legacy and modern devices that may not be capable of implementing IP address randomization, port randomization, or overlay networks at the end systems directly. For this reason, a transparent solution to the end systems is needed that can merge the above capabilities in a critical infrastructure environment. The approach that we present in this paper explores the use of software-defined networking as a solution.

III. Threat Model

The network randomization (NR) portion of our approach assumes that an adversary has successfully gained access to a system and is able to observe traffic within the network. The goal of the adversary may be D/DoS, reconnaissance, targeting a specific service, or targeting a specific host on the network. Our goal is to prevent that adversary from learning the true IP addresses and port numbers of the services being offered on a network to mitigate the scope of damage of targeted reconnaissance attacks

IV. Implementation

We implemented a proof-of-concept for randomizing IP addresses, port numbers, and network routes. Each of these techniques is described in detail below.

A. NR with Port Randomization

Randomizing network port numbers increases the difficulty required for an adversary to learn the services running on a system. Well-known services are typically run on port numbers less than 1024 and are defined in the /etc/services file on UNIX-like systems. One of the first steps an adversary typically takes during the reconnaissance phase of an attack is to probe a system or sniff traffic on the network to learn which services are running. This knowledge is used to gather vulnerabilities related to those services and subsequently plan attack vectors based on those services. Port randomization forces the adversary to both track and reverse engineer the communication protocols to learn information that is typically easy to determine without network port randomization.

Port randomization has been implemented on each host within a network with the aid of the netfilter kernel module. In our implementation, the iptables firewall utility is used to facilitate the random port mappings. iptables provides an interface for a system to filter traffic entering, passing through, and leaving a system based on policy rules defined by a user. The port randomization implementation leverages the Network Address Translation (NAT) iptable filter chain. Rules applied to the NAT chain allow a user to filter just before a packet has been routed on the incoming interface and just before a packet leaves the outgoing interface. The NAT filter rules allow a user to redirect or overwrite port numbers and IP addresses in each packet if it matches user-specified IP header parameters.

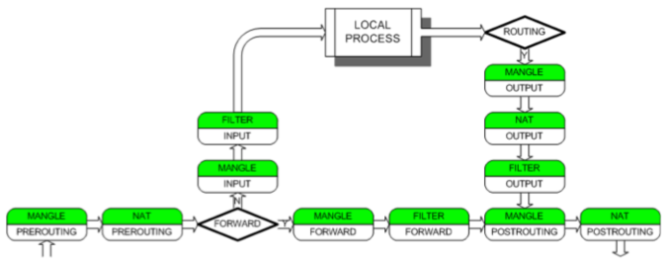

Figure 1: *iptables* filter engine control flow.

Figure 1 shows how packets traverse the filtering engine [13] provided by *iptables*. With the NAT chain, first the packet will pass through the PREROUTING chain. The PREROUTING chain allows a user to overwrite both the IP addresses and port numbers before routing takes place. If the potentially modified packet is to be forwarded to a remote system on the network, then it will be passed through the POSTROUTING chain where packets can be redirected to another IP address or port number if desired. If the packet is destined for a local host process, then the potentially modified packet is sent to that process. If the local process wishes to send a packet outbound, then that packet will pass through the NAT OUTPUT rules where modifications of IP addresses and port numbers can be made, as was the case with forwarding.

The algorithm used to randomize port numbers in our implementation is as follows, in pseudo-code:

```
// well known ports
$portMaps = [1, 2, ..., 1023]
random.seed(time())
// random permutation of well known ports
random.shuffle($portMaps)

for $i = 0 to len($portMaps) do:
   // map a random port number coming into
   // a system to a well known port number
   $inboundRule = iptables -t nat
        -A PREROUTING -p tcp
        --dport $portMaps[$i] -j DNAT
        --to-destination 127.0.0.1:($i+1)
   // map a well known port number leaving
   // a system to a random port number
  $outboundRule = iptables -t nat

        -A OUTPUT
        -p tcp --dport ($i+1) -j DNAT
        --to-destination
        127.0.0.1:$portMaps[i]
   // Apply the rules to the system
   syscall($inboundRule)
   syscall($outboundRule)
```

In the algorithm, first, all nodes synchronize clocks and seed the random number generator with time. We note that time was used for simplicity in our test implementation, but in practice a non-predictable seed should be used. Next, the port mappings are randomized and iterated to create random mappings. The random mappings are then appended to an incoming PREROUTING *iptables* NAT rule as well as to an outgoing OUTPUT rule. Both rules to create and invert the mappings are necessary for both sides of the communication channel to be aware of the translation. A *cron* job is then scheduled to run every minute on each machine participating in the port mapping. Our implementation is required to run on each endpoint participating in the port randomization. However, a similar implementation can be deployed at the network layer within an SDN setting so that our approach can scale to a larger number of nodes and be transparent to the end nodes.

B. NR with IP Address Randomization

The goals of randomizing the network IP addresses are to deceive an adversary and mitigate "hitlist" type attacks [14] without placing additional burden on the end users of a network. To achieve these goals, it is desirable to randomize the IP addresses at the switch level so that the randomization is transparent to the end user themselves. This architecture eliminates the need for every client to load and run the IP address randomization protocol. Only the network switches participating in this protocol are required to maintain and perform the IP address mappings which allows the IP address randomization to scale on a per-switch basis rather than a per-user basis. This architecture also reduces the attack surface from every client on the network to only the switches running on the network.

The IP address randomization application is a multi-component module written for an OpenFlow-based software-defined-networking (SDN) controller. At the heart of the IP address randomizer are three components: the network randomization algorithm and OpenFlow interface (*nwr*), the random IP address generator (*gen*) and the network mapping database module (*netmap*). Ancillary modules/code provide for a RESTful API via which an external application may trigger a force-randomization action for the network.

The *nwr* component is the main body of the IP address randomizer application. Upon initialization, the *nwr* module reads a network specification file and populates a backend database. Thereafter, it creates random IP address generator objects for each network under randomization.

The *nwr* module listens for OpenFlow (OF) events regarding the connection of OF compatible switches. It then creates a

Switch object for each of the connected switches, and keeps track of those Switch objects in using the unique datapath-id (dpid) of the OF switch. Using a deny-by-default approach, the Switch object will only switch packets when said packets match installed flow rules on the switch. If there is not a flow rule that the packet may match against, the packet header is sent to the controller to determine how to treat the packet.

The *nwr* module hosts an address-resolution protocol (ARP) server. So, if the packet is an ARP packet, the controller checks the backend database to see if the requestor and the requested IPs are part of the network. If they are, it responds with the appropriate MAC address. After this condition is met, the primary packet processing algorithm is traversed.

The algorithm only permits communication (and random IP address assignment) between entities that are part of the network as specified, and are directly connected to an OpenFlow-compatible switch. Special considerations have been given to devices that may not be able to meet the latter requirement – particularly router gateways and/or DHCP servers. Gateways are included in the network map, and are also specified in the *nwr* code. When an ARP is received from a gateway address for a random IP address, the gateway IP address and MAC addresses are checked against the backend database. If validated, the database is accessed for the MAC address of the random IP address and subsequently returned to the gateway. A similar process is employed for DHCP servers.

When the two endpoints in a communication request are validated, random IP addresses are assigned to each. Flow-rules are then constructed and sent to the endpoint's first hop OpenFlow switch. These flow rules contain matches for the respective endpoints to translate real IP addresses into the assigned random IPs (and vice versa). For this, there are two implementations: (1) reactive IP address randomization, and; (2) proactive IP address randomization. The former only installs flow-rules when communication is initiated by an endpoint. The latter enumerates links for all pre-defined connection requirements, and actively installs flow-rules to first-hop switches.

A roll interval prescribes the timeout period for each of the randomization flow-rules. Each flow rule's idle timeout is set for infinity. Thus, this period is used for the hard timeout of a flow-rule. The roll intervals may be set for static periods of time, or for random periods of time between a set of upper and lower bounds.

The *gen* component contains the logic for random IP address generation. It contains a queue data structure whose depth is initialized with the size of network (total assignable IP addresses under the defined network length). Its purpose is to keep track of the used random IP addresses, so as to avoid reassignment or collision. Additionally, an array is kept to track the random IP address and the true MAC address of the endpoint. This is primarily used for ARP responses to gateways that may not be part of the subnetworks under randomization.

The *netmap* component provides the necessary interface backend database that stores the true network map(s). All entries are derived from a network specification file. The *netmap* component itself consists of the several functions to aid the primary *nwr* switch algorithm. The `getSource` function is used by *nwr* to verify that a packet received from some IP is allowed to be within the network(s) under randomization. Using the source's IP address, and packet information detailing the data path identifier (dpid; the unique identifier of an OpenFlow switch on a network) and port it was received on, a check is done against the data in the database. If the IP address, dpid and port are validated, the dpid's "uplink" port is returned to the nwr (for the crafting of the forwarding action in the nwr flow-rule action). If the data is invalid, nothing is returned and the packet(s) is dropped. The `getDest` function does a similar test, but on the destination IP address for the packet. If the destination IP address is not in the database, nothing is returned (and the packet is dropped). If the destination is valid, an array of information containing the destination MAC address, dpid, port and dpid uplink port is returned. The final function, `getMAC`, is primarily used by *nwr*'s ARP server, to retrieve MAC addresses for validated source/destination pairs. The function returns nothing for unfound MAC addresses.

C. NR with Route Randomization

IP address randomization is still susceptible to traffic analysis. A passive adversary can determine the endpoints in communication by monitoring ingress and egress points in a network regardless if IP address randomization is deployed. Therefore, it becomes necessary to randomize the paths that packets traverse through the network to prevent such correlations. To randomize paths, the network topology describing endpoints and interconnections is needed to compute all possible network paths between each pair of nodes that does not contain loops. The set of network nodes a packet can traverse are defined as an overlay network as shown in Figure 2. When two endpoints communicate, a random path is selected as the circuit switched line of communication within the overlay network for a configurable period of time. This circuit is periodically randomized and is user configurable on how frequently it is randomized. A Breadth First Search (BFS) algorithm is used to generate all possible paths. A stack is maintained to ensure that no paths are included that contain loops to prevent. Since the paths packets take within the network are constantly changing, an adversary must correlate traffic at every switch participating in the overlay network instead of just the entry and exit points of packets.

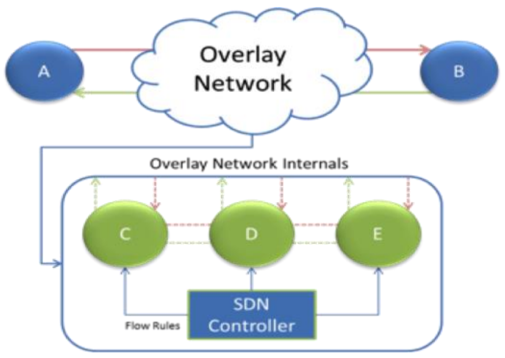

Figure 2: Nodes A and B communicating through an overlay of switches C, D, and E with flows installed from the SDN Controller.

Our implementation operates within an SDN setting where the controller is responsible for learning the topology and installing random flows for packets to traverse. Since a centralized controller manages the installed flows, the path randomization is transparent to the end devices. The initialization costs associated with the path randomization implementation itself are the storage and computational costs to calculate all possible non-looping paths between each pair of nodes. Because the paths are constantly changing, our implementation proactively installs flows to avoid the same startup costs each time paths are randomized. The performance overheads vary depending on network topology, link speeds, and additional hops taken within the randomly selected network path.

V. Experiment Set Up and Results

To assess the efficacy of our network randomization approaches, we leveraged research done by Van Leeuwen, et al [15] regarding the development of defensive work factors for MTDs. The experimental setup consisted of a highly-instrumented testbed using virtual machines running the three different randomization techniques. For the port and IP hopping techniques, each endpoint in the virtual system ran network performance monitoring software to capture performance data for TCP and UDP; the former was comprised of bandwidth counts, while the latter consisted of bandwidth, jitter and packet loss counts. Included in the parameters for testing were the rotation intervals for the reactive IP address randomizer. To test route randomization, we used the IPerf [16] tool to measure Round Trip Times (RTT), bandwidth, and data transfer times.

Table 1 displays the TCP bandwidth tests for the Port-hopper and IP address randomization techniques. Tests were run for 240 seconds and 480 seconds, to observe the smoothing effect for longer periods of time. For proactive IP address randomization, the randomization time were fixed at 30 seconds. For the Port-hopper, hopping was set at 1 second intervals. The reactive IP address randomizer had randomization intervals set at 10, 30, 60 and randomly between 30 and 60 seconds.

Table 1: TCP Bandwidth Tests for IP Address and Port Randomization

Test Duration (s)	Technique	BW Percentage
240	Port hopper	0.977628
480	Port hopper	0.981913
240	IP-proactive	0.998191
480	IP-proactive	0.998593
240	IP-reactive 10s	0.711685
	IP-reactive 30s	0.908663
	IP-reactive 60s	0.952302
	IP-reactive random	0.955762
480	IP-reactive 10s	0.724593
	IP-reactive 30s	0.910393
	IP-reactive 60s	0.967297
	IP-reactive random	0.953270

RTT, data transfers and bandwidth measurements were collected with path randomization disabled and enabled as shown in Table 2. The experiment was performed using five overlay switches between a pair of nodes communicating within a virtual machine network. With path randomization enabled, longer routes than the direct path were randomly selected which negatively affected the RTT, data transfers and bandwidth measurements.

Table 2: No Randomization vs. Path Randomization

	RTT (ms)	10s Data Transfers (GB)	Bandwidth (Gbit/s)	1MB Transfer (ms)
Baseline	50.90798	22.25135	19.21674	69.11273
Random Path	62.64078	21.59769	20.59946	101.96778

To test UDP, tests were again run for 240 and 480 seconds; bandwidth streams were set for 1Mb, 10Mb, 50Mb and 100Mb. Here, we captured the resultant jitter and packet loss metrics, as show in the figures below. The bandwidth tests for UDP were less interesting, all three techniques performed at 99% or better, with the exception of reactive IP address randomization for 10 second rotation intervals – at 100Mbit streams, they were only able to satisfy 98% of the stream data rate.

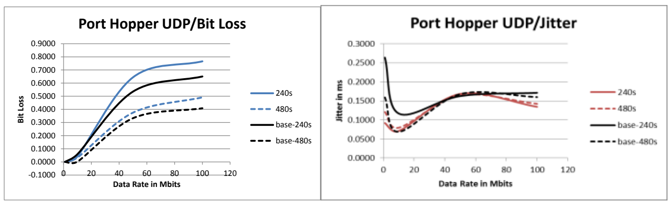

Figure 3: Port Hopper

Of the three techniques, the port hopper most closely represented the baseline traffic tests. At 100 Mbits, the smoothed averages of bit loss were approximately 0.10 units different. The jitter profiles between the tests remained largely the same.

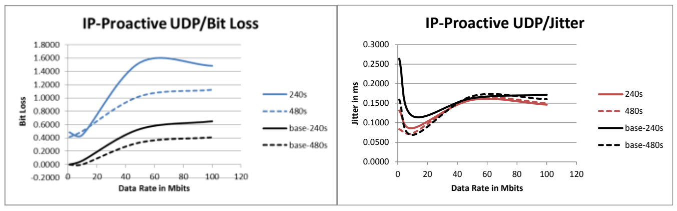

Figure 4: IP Address Proactive

The proactive IP address randomization technique displayed the same jitter profile as the baseline tests, finding a 'sweet spot' the 10 Mbit date rate. The bit loss was higher than the port hopper, exceeding the baseline by approximately 0.8 units.

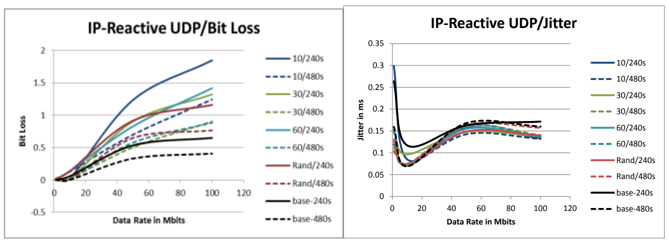

Figure 5: IP Address Reactive

Tests performed on the reactive IP address randomization technique involved varing the rotation time of the IP addresses, specifically for 10, 30, 60 and a random interval between 30-60 seconds. As expected, those experiments with smaller rotation times experience greater bit loss, with as much as approximately 1.2 units greater than baseline. Interestingly, we expected the bit losses to be higher than the proactive IP address randomization approach – due to the reactive IP address randomization approach having to install flows on demand, and the proactive IP address randomization installing them beforehand – but for a 30 second rotation time the bit losses are strikingly similar, at approximately 1.35 units. Like the previous techniques, however, the impact to jitter was minimal.

VI. Conclusions

Computer networks, in particular systems running in many critical infrastructure facilities, continue to foster predictable communication paths and static configurations that provide a vector for accessing the critical assets of a network. In our research, we provide a means to mitigate certain classes of attacks by automatically reconfiguring network settings and randomizing application communications dynamically.

Our experimental results show that the port randomization approach provides the least amount of performance impact, while successfully maintaining connectivity of a communication session. While port hopping works well to thwart application-based attacks, it still does not address protocols below TCP/UDP (e.g., ICMP). Addressing those types of attacks, as well as others mentioned, may be done with the IP Randomization techniques. The experimental results for the IP address randomization showed that while the two approaches successfully maintained connectivity between two hosts communicating with one another, impacts to performance were greater than the port hopper, most notably with the reactive IP address randomization approach. The advantage to the reactive IP address randomization approach is that resources and flow-rules are only used when communication is required; the reactive IP address randomization approach may suit well for low-bandwidth applications such as SCADA. However, for systems that require greater bandwidth coupled with delay-intolerance, the proactive IP address randomization approach should be considered. Path randomization should be used with care since additional hops through the overlay network may cause potentially unacceptable delays. In time critical applications, limiting the number of additional random hops in the overlay network should be considered.

A. Limitations

1. Our focus for IP address randomization has only been at the switch level. The functionality of OpenFlow would allow this technique to be applied to other network elements, for example, routers, but this is beyond the scope of this paper.

2. The port randomization and IP address randomization techniques can be run in tandem but are separate pieces of code. A more elegant design would be to combine the two—this is possible but outside the scope of our project.

3. IP address randomization must consider a large enough bitmask. For reactive IP address randomization, four IP addresses are used for each bidirectional communication session, and two IP addresses are used for the proactive IP address randomization implementation.

References

[1] Rosslin John Robles, Min-kyu Choi, Eun-suk Cho, Seok-soo Kim, Gil-cheol Park, Jang-Hee Lee, “Common threats and vulnerabilities of critical infrastructure,” *International Journal of Control and Automation*, 2008.

[2] David Goldschlag, Michael Reed, Paul Syver- son, “Onion routing for anonymous and private internet connections,” Communications of the ACM, vol. 42, no. 2, pp. 39–41, 1999.

[3] Wang Shmatikov, Ming-Hsiu Vitaly, “Timing analysis in low-latency mix networks: Attacks and defenses,” *Proceedings of the 11th European conference on Research in Computer Security*, pp. 18–33, 2012.

[4] Jean-Franc ̧ois Raymond, “Traffic analysis: Protocols, attacks, design issues and open problems,” Springer-Verlag Lecture Notes in Computer Science, pp. 10–29, 2009.

[5] Roger Dingledine, Nick Mathewson, Paul Syverson, “Tor: The second-generation onion router,” 2004.

[6] Gildas Nya Tchabe and Yinhua Xu,”Anonymous Communications: A survey on I2P”,CDC Publication Theoretische Informatik-Kryptographie und Computeralgebra (https://www.cdc.informatik.tu-darmstadt.de), 2014

[7] http://metrics.torproject.org/users.html, 2014.

[8] Sambuddho Chakravarty, Marco V Barbera, Georgios Portokalidis, Michalis Polychronakis, Angeles D Keromytis, “On the effectiveness of traffic analysis against anonymity networks using flow records,” Springer-Verlag Lecture Notes in Computer Science, vol. 8362, pp. 247– 257, 2014.

[9] Angelos D Keromytis, Vishal Misra, Dan Rubenstein, “Sos: An architecture for mitigat- ing ddos attacks,” *Journal of Selected Areas in Communications*, vol. 21, 2003.

[10] J. H. J. Ehab Al-Shaer, Qi Duan, “Random host mutation for moving target defense,” ACM, 2012.

[11] Kamran Ahsan, Deepa Kundur, “Practical data hiding in tcp/ip,” Workshop Multimedia aand Security at ACM Multimedia, 2002.

[12] Lexi Pimenidis, Tobias K ̈olsch, “Transparent anonymization of ip based network traffic,” *In Proceedings of 10th Nordic Workshop on Secure IT-systems*, 2005.

[13] R. Russell, “Linux 2.4 nat howto.” http://www.netfilter.org/documentation/HOWTO/NAT- HOWTO.html, 2002.

[14] B. Rajesh, Y.R. Janardhan Reddy, B. Dillip Kumar Reddy, “A Survey Paper on Malicous Computer Worms,”, *International Journal of Advanced Research in Computer Science and Technology*, vol. 3, 2015.

[15] B.Van Leeuwen, “Operational Cost of Deploying Moving Target Defenses: Defensive Work Factors,” Pending Publication, 2015

[16] IPerf: https://iperf.fr/