SANDIA REPORT
SAND2015-3324
Unlimited Release
April 2015

Network Randomization and Dynamic Defense for Critical Infrastructure Systems

Adrian R. Chavez, Jason Hamlet, Erik Lee, Mitchell Martin, William Stout

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

SAND2015-3324
Unlimited Release
April 2015

Network Randomization and Dynamic Defense for Critical Infrastructure Systems

Adrian Chavez, Jason Hamlet, Erik Lee, Mitchell Martin, William Stout
Networked Systems Survivability & Assurance, Assurance Technologies and Assessments
Sandia National Laboratories
P.O. Box 5800
Albuquerque, New Mexico 87185-MS0672

Abstract

Critical Infrastructure control systems continue to foster predictable communication paths, static configurations, and unpatched systems that allow easy access to our nation's most critical assets. This makes them attractive targets for cyber intrusion. We seek to address these attack vectors by automatically randomizing network settings, randomizing applications on the end devices themselves, and dynamically defending these systems against active attacks. Applying these protective measures will convert control systems into moving targets that proactively defend themselves against attack. Sandia National Laboratories has led this effort by gathering operational and technical requirements from Tennessee Valley Authority (TVA) and performing research and development to create a proof-of-concept solution. Our proof-of-concept has been tested in a laboratory environment with over 300 nodes. The vision of this project is to enhance control system security by converting existing control systems into moving targets and building these security measures into future systems while meeting the unique constraints that control systems face.

ACKNOWLEDGMENTS

The authors acknowledge the work that produced the results presented in this paper was funded by the U.S. Department of Energy/Office of Electricity Delivery and Energy Reliability (DOE/OE) as part of the Cybersecurity for Energy Delivery Systems (CEDS) Program.

CONTENTS

FIGURES

TABLES

NOMENCLATURE

ARP	Address Resolution Protocol
API	Application Program Interface
ASLR	Address Space Layout Randomization
AUC	Area Under Receiver Operating Characteristic Curve
CEDS	Cybersecurity for Energy Delivery Systems
CIDR	Classless Inter-Domain Router
CIP	Critical Infrastructure Protections
DB	Database
DHCP	Dynamic Host Configuration Protocol
DNS	Domain Name Service
DOE	Department of Energy
DPID	Datapath ID
FILO	First-In-Last-Out
FN	False Negative
FP	False Positive
GEN	Random IP Generator
IDS	Intrusion Detection System
IP	Internet Protocol
ISR	Instruction Set Randomization
IT	Information Technology
LLVM	Low Level Virtual Machine
MAC	Media Access Control
MCC	Matthew's Correlation Coefficient
NR	Network Randomization
NWR	Network Randomization Algorithm and OpenFlow Interface
NETMAP	Network Mapping Database Module
OF	OpenFlow
OVS	OpenFlow vSwitch
PLC	Programmable Logic Controller
REST	Representational State Transfer
RTU	Remote Telemetry Unit
SCADA	Supervisory Control And Data Acquisition
SDN	Software Defined Networking
SNL	Sandia National Laboratories
SNMP	Simple Network Management Protocol
SVM	Support Vector Machine
TN	True Negative
TP	True Positive

1. NETWORK RANDOMIZATION

Critical Infrastructure control systems continue to foster predictable communication paths and static configurations that allow easy access to our Nation's most critical assets. This makes them attractive and easy targets for cyber attack. This project addresses these attack vectors by automatically reconfiguring network settings and randomizing application communications dynamically. Applying these protective measures will convert control systems into moving targets that proactively defend themselves against attack.

The network randomization module is responsible for managing network reconfiguration, securely communicating reconfiguration specifications to other network nodes, and ensuring that connectivity between nodes is uninterrupted. This will help eliminate the class of adversaries who rely on known static addresses of critical infrastructure network devices for attack.

1.1 Implementation

The Network Randomization (NR) application is a multi-component module written for the POX software-defined-networking (SDN) controller. POX is an open-source SDN controller, written in the Python language. As such, it can be run on virtually any OS platform that can handle Python (Windows, Mac OS and Linux). At the time of this document's publication, we have primarily used POX in Ubuntu 14.04 LTS with Python version 2.7.6.

At the heart of the NR application are three components: the network randomization algorithm and OpenFlow interface (nwr), the random IP 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 functionality of these components is spread out through a number of files, located in the POX directory structure. Those files (code and otherwise) required to run the NR application are shown below (note: all files contained in ~/pox/ext/, except where indicated).

Python files
nwr.py
gen.py
netmap.py
nwr_json.py
nwr_webservice.py
../pox/web/jsonrpc.py

Other files
network.csv
dpid.db

The remainder of this section includes: (1) high level explanations of the three core components, to include necessary input parameters; (2) the RESTful API; and (3) considerations and requirements for integration.

1.1.1. nwr

The nwr component is the main body of the NR application. Upon initialization, the nwr module reads the network specification file network.csv and populates the dpid.db sqlite3 database. Thereafter, it creates random IP generator objects for each network under randomization. At this point, the nwr module is then registered with the POX core object.

After registration, the nwr module listens for POX OpenFlow (OF) events regarding the connection of OF compatible switches to the POX core module. It then creates a Switch object for each of the connected switches, and keeps track of those Switch objects in a Python dictionary, keyed by the unique datapath-id (dpid) of the OF switch. When a switch is connected to the OF controller, it 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 where controller logic determines how to treat the packet (controller logic in this project is contained in the nwr module).

The Switch class contains the primary nwr algorithm. The algorithm is commenced when a packet_in event is generated by the switch; this packet_in event is indicative of the packet not matching any flow rules (often called a rule-miss). The nwr module hosts an address-resolution protocol (ARP) server. So, if the packet is an ARP packet, the controller checks the netmap database to see if the requestor and the requested IP are part of the network. If they are, it responds with the appropriate MAC address. The nwr module also has the ability to pass DHCP traffic unscathed, to ensure endpoints are properly addressed (this feature was added due to testing conditions in virtual environments, ideally IP addresses would be static on endpoints and thus not require DHCP traffic to pass). After these two conditions have been passed, the following algorithm is traversed:

```
tIP           true IP on the workstation
oIP           overlay IP for the overlay IP network
dpid          datapath ID ~ an Openflow-enabled Switch
controller    Openflow controller for dpid
database      sqlite3 network map
database_o    ephemeral oIP/MAC mapping
```

```
(1) Local SRC to remote DST (received on local port)
if switch receives SRC tIP (implied: no rule match)
    Check database to match tIP from correct dpid port
    if match
        Check if DST tIP valid
            if valid
                if on different dpid
                    change SRC tIP->oIP and DST tIP->oIP
                    output action to egress port (uplink port)
                    install rule on local switch
                        idle timeout: n1, hard timeout: n2
                    install rule on remote switch to change oIP->tIP
                        idle timeout: n1, hard timeout: n2+1
                    send packet
                if on same dpid
                    if on local switch, output action: send to correct DST port
                            send packet
            if invalid
                drop packet
    if no match
        drop packet
```

The algorithm only allows communication (and random IP address assignment) to occur 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 (see below). When an ARP is received from a gateway address for an oIP, the gateway IP and MAC are checked against the netmap database. If validated, then database_o is accessed for the MAC of the oIP and subsequently returned to the gateway. A similar process is employed for DHCP servers.

The specification of these IP addresses, along with several other pertinent properties of the network under the NR application, must be entered at the top of the nwr.py file. Details are described below.

```
arp_queue = deque(maxlen=0)
```

The arp_queue is a first-in-last-out (FILO) queue that keeps track of the number of responses to an ARP request. Its purpose is for performance. Since ARPs are broadcast packets, it is possible that many OF-switches may receive identical ARPs for the same address (e.g., as sent by a gateway router). The arp_queue attempts to limit the number of ARP responses already sent for a particular ARP request. However, setting this to 0 length will not impact functionality of the module.

```
FLOOD_DELAY = 5
```

The FLOOD_DELAY variable is also another performance enhancer. It simply provides a wait period for the newly connected switch before it starts to broadcast traffic.

```
# Interval to roll/randomize IPs (in seconds)
# Specify as a number or "RANDOM"
# ROLL_INTERVAL = 4
# If RANDOM, specify the interval, e.g. 30 to 60 seconds
ROLL_INTERVAL = RANDOM
ROLL_MIN = 30
ROLL_MAX = 60
```

The roll interval section prescribes the timeout period for each randomization flow-rule. Each flow rule's idle timeout is set for infinity. Thus, this period is used for the hard timeout of a flow-rule. For a fixed time, set the ROLL_INTERVAL to the time in seconds. To randomize the life-time of a flow-rule, the ROLL_INTERVAL may be set to RANDOM. Then, the interval of time to randomize in can be specified in ROLL_MIN and ROLL_MAX.

```
# Dictionary Networks
# [hash] = network address; [0]=CIDR; [1]=approx devices; [2]=gateway
networks = {
            "192.168.0.0" : [24, 100, "192.168.0.1"],
            "10.0.0.0"    : [24, 100, "10.0.0.1"]
           }
```

The Dictionary Networks section describes the subnetworks that will be under IP randomization. The data structure is a Python dictionary, where the key (or hash) is the network address. Each hash is linked to a list structure that defines: [0] the classless inter-domain router (CIDR) prefix, given as an integer [1] the approximate number of devices in the said subnetwork; and [2] the gateway IP address for the subnetwork. Careful thought should be given to the CIDR and number of devices in the network. Ideally, the number of devices in the network should be not more than half the size of the CIDR, and even less if multiple communications between endpoints may be expected. This is due to the assignment of random oIP addresses to the endpoints. Consider two endpoints A and B, each in a subnetwork NA and NB, respectively. Random oIPs are assigned for each unidirectional communication flow. Thus, for A → B, there will be an oIP for A from NA, and an oIP for B from NB . For each bidirectional communication between two endpoints, two oIPs will be used from the pool of available oIP addresses for each endpoint. Thus, for example, if every device in a network reports back to a server at the same time, and there are n devices, then there must be at least 2n available oIPs to assign (if the communications are bidirectional).

```
dhcpservers = set()
dhcpservers.add("192.168.255.254")
dhcpservers.add("10.0.0.254")
```

Finally, if the subnetworks and/or environment require the use of DHCP servers for IP address assignment, their addresses should be added to the dhcpservers set as shown above.

1.1.2. gen

The gen component contains the RandomIPGenerator class. This object is instantiated with the following class variables:

```
dq: deque(maxlength=network_size)
mac_map{}
```

The dq variable is a Python deque data structure whose depth is initialized with the size of network (total assignable IPs under the defined CIDR). Its purpose is to keep track of the used oIPs, so as to avoid reassignment or collision of oIPs. The mac_map consists of elements that are keyed by an oIP; each item stores the true MAC of the endpoint. This is primarily used for ARP responses to gateways that may not be part of the subnetworks under randomization.

The following getter functions make up the rest of the class:

```
def getRandomIP (self, network_address, cidr=0, mac=None)
  def ipGenCIDR(self, cidr_prefix, network_address)

def getOIPMac (self, oIP)
```

The getRandomIP provides a randomly generated oIP in the respective CIDR. The supplied MAC is then used as an entry for the mac_map dictionary. The getRandomIP function contains a sub definition for ipGenCIDR. This is done to allow future randomizer functions (e.g., IPv6). The getOIPMac returns the true MAC for the passed in oIP.

1.1.3. netmap

The netmap component provides the necessary interface to the sqlite3 database that stores the true network map(s), as contained in the file dpid.db. The sqlite3 db is flat, consisting of a single table with the following fields:

Index	Field	Type	Example
0	host_name	string	"rtu_1675"
1	host_ip	string	"172.16.0.100"
2	host_mac	string	"00:00:00:01:de:cb"
3	dpid	string	"00-00-00-12-23-34"
4	dpid_port	int	1
5	dpid_uplink	int	10

All entries are derived from the network.csv file, which is described in more detail in a later subsection. The netmap component itself consists of the NetworkMapperDB class. The class has two class variables: (1) connection (which connects to the sqlite3 database file); and (2) cur (a function under the connection object used to query the sqlite3 database). The remainder of the class is comprised of the following functions.

```
def getSource (self, src_ip, src_dpid, src_port)
def getDest (self, dst_ip)
def getMAC (self, ip)
```

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, and packet information detailing the dpid (switch) and port it was received on, a check is done against the data in the dpid.db database. If the IP, dpid and port are validated, the dpid_uplink port is returned to the caller (for the crafting of the forwarding action in the nwr flow-rule action). If the data is invalid, None is returned. 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, None is returned. If it is, a list containing the destination MAC, dpid, dpid_port and dpid_uplink is returned. The final def, getMAC, is primarily used by nwr's ARP server, to retrieve MAC addresses for validated source/destination pairs. The function returns None for unfound MAC addresses.

1.1.4. RESTful API

The NR application provides a RESTful API via which a JSON formated request may be passed to the application. This request is used solely by an external application to force re-randomization of the network address space. This optional service is hosted as a webservice on port 8000 of the NR host machine. An example using the curl application is shown below.

```
curl -i -X POST -d \
'{"method":"force_randomization","params":{"dpid":"'$DPID'"}}' \
http://127.0.0.1:8000/OF/
```

```
curl -i -X POST -d '{"method":"force_randomization"}' \
http://127.0.0.1:8000/OF/
```

The first example depicts sending a randomization request for a specific switch, whose dpid should be specified in the $DPID variable in the following format: "XX-XX-XX-XX-XX" (where X is a hexadecimal value). The second example depicts the command to force randomization for all registered switches in the network(s) under randomization. For both commands, the correct IP address for the POX host machine should replace the 127.0.0.1 address.

If integrating into a Python file, code can be written like the example below.

```
import httplib
import json
POX_IP = "127.0.0.1" #IP address of the POX controller
DATA = {"method":"force_randomization"}
# To forcing just one dpid
# DPID = "00-00-00-12-34-56"
# DATA = {
#     "method":"force_randomization",
#     "params":{
#         "dpip":DPID
#         }
#     }

class ForceRandomization (object):
  def __init__(self, server):
    self.server = server
  def post(self, data):
    ret = self.get_data(data, 'POST')
    return json.loads(ret[2])
  def get_data (self, data, action):
    print "Sending force randomization command."
    path = "/OF/"
    headers = {
        "Content-type":"application/json",
        "Accept":"application/json",
         }
    body = json.dumps(data) #convert dict to json string
    conn = httplib.HTTPConnection(self.server, 8000)
    conn.request(action, path, body, headers)
    response = conn.getresponse()
    ret = (response.status, response.reason, response.read())
    conn.close()
    return ret
```

The RESTful API is implemented in the following files: nwr_json.py, nwr_webservice.py, and ../pox/web/jsonrpc.py.

1.1.5. Requirements

1.1.5.1. OpenFlow Compatibility

The linchpin to the NR system is the necessity of OpenFlow compatible switches. In our experiments and testing, we have integrated Open vSwitches (OVS) into each of our Linux-based endpoints. However, if this is not possible or feasible, endpoints may be located behind OpenFlow compatible switch(es), so long as they have a channel back to the OpenFlow controller.

For an Open vSwitch implementation, the following configuration commands may be used:

BRIDGE = the named OVS bridge on the endpoint
IP = the IP address of the POX controller host machine

```
ovs-vsctl set-controller $BRIDGE tcp:$IP:6633
ovs-vsctl set controller $BRIDGE connection_mode=out_of_band
ovs-vsctl list controller
```

Physical OpenFlow compatible switches should undergo the same type of configuration, based on switch specification(s).

1.1.5.2. Network Map

The NR application requires a network map that maps to the format described in the netmap section above. To reiterate:

Index	Field	Type	Example
0	host_name	string	"rtu_1675"
1	host_ip	string	"172.16.0.100"
2	host_mac	string	"00:00:00:01:de:cb"
3	dpid	string	"00-00-00-12-23-34"
4	dpid_port	int	1
5	dpid_uplink	int	10

These data should be included for each device (endpoint) on the network, collected in a comma separated values (csv) file named network.csv. This file should be placed in the ~/pox/ext/ directory before nwr initialization. As an example, the lines of the file should resemble something like:

```
gw_192,192.168.0.1,08:00:27:ab:92:54,00-00-00-00-00-00,0,0
gw_172,172.16.0.1,08:00:27:94:a7:71, 00-00-00-00-00-00,0,0
gw_10,10.0.0.1,08:00:27:0f:7f:52, 00-00-00-00-00-00,0,0
serverA_192,192.168.0.201,08:00:27:15:bd:18,08-00-27-15-bd-18,65534,1
serverB_192,192.168.0.202,08:00:27:ef:5c:22,08-00-27-ef-5c-22,65534,1
serverA_172,172.16.0.201,08:00:27:b0:03:9e,08-00-27-b0-03-9e,65534,1
serverB_172,172.16.0.202,08:00:27:e0:c6:0b,08-00-27-e0-c6-0b,65534,1
serverA_010,10.0.0.201,08:00:27:cc:9c:d8,08-00-27-cc-9c-d8,65534,1
serverB_010,10.0.0.202,08:00:27:d8:7b:6f,08-00-27-d8-7b-6f,65534,1
```

For those devices that are not under randomization but must be in the network for ARP purposes (e.g., router gateways), the dpid, dpid_port and dpid_uplink may be defined as 00-00-00-00-00-00,0,0 respectively. When one or more devices are behind a single dpid (switch), it is OK to have a one-to-many relationship for switch to endpoints.

1.1.5.3. Infrastructure

The current implementation of the NR system requires complete separation of the data network (e.g., the operational network) and the OF management network. As such, each device that provides a dpid must have at least two interfaces, one for the OF management network to connect to the POX controller machine, and any remaining interfaces for the data network. In our experiments and testing, our endpoints had Open vSwitches installed on them; one physical interface on the endpoint was reserved for the OF network, the other connected to the data network, through the Open vSwitch.

For example configuration, the result of an OVS show command (sudo ovs-vsctl show) might display the following:

```
Bridge "br0"
     Controller "tcp:10.0.0.100:6633"
     Port "eth0"
          Interface "eth0"
     Port "br0"
          Interface "br0"
               type: internal
```

And if using an OS like Ubuntu, an example /etc/network/interfaces file may be written as:

```
auto eth0
iface eth0 inet static
     address 0.0.0.0     # A promiscuous interface

auto eth1
iface inet static
     address 10.0.1.111       # OpenFlow out-of-band IP
     netmask 255.255.255.0

auto br0
iface br0 inet static
     address 192.168.0.11  # Operational IP
     netmask 255.255.255.0 # Operational netmask
     gateway 192.168.0.1      # Operational gateway
```

With such a configuration, the bridge's br0 interface will host the IP address for the operational network, and will automatically assume the MAC address of the physical eth0 interface (as added to the bridge br0). This simplifies not only the MAC addresses, but also the dpid – as the dpid of br0 will also be the MAC of eth0/br0. The eth1 interface is addressed for the out-of-band OF management network, and will connect to the POX controller (and the nwr module) upon initialization. Thus, as traffic passes through the network to the endpoint address (10.0.0.111 in this example), or originates from the endpoint, it will pass through the OF controlled OVS on the endpoint. This action will trigger a packet_in event for the nwr module, thereby starting the randomization algorithm.

If this is not possible for an endpoint, then locating that endpoint behind an OpenFlow compatible switch will suffice, again, as long that switch has an out-of-band channel back to the POX host machine.

2. APPLICATION RANDOMIZATION

Automatically discriminating between legitimate code and mobile malicious code running on a system is a difficult problem. It generally requires visibility into the internal state of the compromised application, and detailed knowledge of the application's normal behavior. This comes at a hefty price in terms of performance, and is still unreliable as a way to detect compromise.

The approach we took with application randomization is slightly different – we are attempting to shift the burden of work onto the malicious code rather than onto the defending system. One of the things that differentiates malicious code from legitimate is its perception of the environment in which it runs. Legitimate code is linked against libraries with defined symbols, and the locations of the symbols are filled in as needed to ensure that the program runs correctly.

Additionally, many conventions are in place between the application and its supporting libraries that enable correct communication; things like calling conventions, data structure layouts, and numerous other points of consistency that are implied in the protocol that connects application and library.

Malicious code, on the other hand, typically lands in a hostile environment and must bootstrap itself without support from the system. To do this, it is essential that the system have predictable, usable properties so that the malicious code can find the resources it needs and put them to use. If we could break that predictability, the malicious code would have an extremely difficult job ahead of it to spread beyond a single, tailor-made infection. That is the goal we are pursuing with application randomization.

Significant work has been done in this area, with good results. Address Space Layout Randomization (ASLR) loads chunks of memory at randomized offsets, making it more difficult to locate library functions and other important data. Instruction set randomization (ISR) changes the instruction set itself so that code written to execute on one machine will not work on another. Combined, these approaches provide some protection from malicious code infections, but they are not complete. ASLR can be defeated by a brute-force approach, because the size of the dislocated memory chunks is large enough that it's possible just to try a few thousand times and find what you're looking for. Randomized instruction sets are difficult to implement on today's hardware, and they can be defeated by return-to-libc attacks, and more generally by return-oriented-programming. In both of those cases, the attack uses the existing code on the system instead of executing its own code by carefully crafting a sequence of return addresses and stack arguments.

2.1 Looking to the Compiler for Help

ASLR and ISR are both instances of a more general scheme, which is to make the environment in which malicious code operates unpredictable and ultimately unusable. We can take that concept much farther if we can alter the way software is compiled and distributed. The concept of application randomization in a nutshell is to find every decision point in a compiler that could be made differently without harming function, and deliberately randomize it. Opportunities for randomization of this kind include function locations, instruction selection, register allocation, choice of internal calling conventions, code block ordering, switch case ordering, stack layout, internal data structure layout, and a variety of other implementation details that could be equivalently done in different ways.

By specifying a key that is unique to a given machine, and by altering the compiler to make pseudorandom decisions based on that key in the above places, we can reach deep into the structure of the resulting binary and create programs that are functionally equivalent across machines, but are so structurally different from each other that it's effectively impossible to navigate them without the map provided by the legitimate install process.

By distributing code in a partially-compiled intermediate language format, we can additionally make the installation process fast and painless as well as support the distribution of non-open-source code. In this scheme, the software is distributed as an intermediate bitcode representation which can be quickly randomized and transformed into machine code at the time of installation.

2.2 Implementation

Our prototype implementation of this concept uses the Low Level Virtual Machine (LLVM) compiler infrastructure to support this process. Programs are compiled into LLVM bitcode, which is a binary representation that serves as the intermediate representation for the optimization passes that LLVM provides for generating quality machine code. Our randomization transformation is implemented as an optimization pass in LLVM, which is run by the standard clang compiler tool chain when compiling C-code. The prototype implements function location randomization, which means that it globally randomizes the order in which functions appear within the resulting binary. When used in a library like the standard C library, this creates a significant obstacle for a malicious program that's attempting to use a method like return-to-libc or return-oriented-programming to bypass other security features.

The randomizing version of the compiler can be dropped in to an existing build system, as long as the clang compiler tool chain is supported. A helper script included with the randomizing compiler serves to make the process transparent by wrapping the standard tool chain with the workflow necessary to compile and link a randomized executable.

2.2.1 Randomizing Compilation Process

The typical compilation process for a large C-language application follows this basic order:
1) Individual C files are compiled to object files
2) Any assembly language files are compiled to object files
3) The set of object files so generated are linked into an executable or library

In order to support randomization, this basic process is altered to include an intermediate step through LLVM bitcode:
1) Individual C files are compiled to LLVM bitcode files
2) Assembly language files are compiled to object files (as before)
3) LLVM bitcode files are linked to create one large LLVM bitcode file. If the code was to be distributed, this file and the assembly objects would be the distribution package.

4) The randomizing optimization pass is run on the large LLVM bitcode file.
5) The LLVM bitcode file is compiled to an object file
6) The assembly object files and the large randomized object file are linked to create the executable or library, as before.

2.2.2 Example Run

The implementation includes the following source code files for the randomizing optimization pass.

FunctionSwap - A directory that holds the randomizing optimization pass source
musl-c-library-symbols* - Samples of the symbols dumped from the musl C library after two different compilation runs
runclang - A python script that acts as a wrapper for clang and llvm's opt command to make the build process mostly transparent.

The LLVM module optimization pass will randomize the order of functions in the module of the code runs on. In order to use it, a copy of the FunctionSwap directory is needed into the lib/Transforms directory of an LLVM-4 source tree. When it's in place, LLVM can be built with 'make' and 'make install'.

The optimization pass will be compiled to a shared object file name FuncSwap.so, which will be installed to the $PREFIX/lib directory specified for the LLVM installation.

The easiest way to use the code is with the provided wrapper for the clang toolset (runclang). runclang manages the build process so that it behaves like a normal two step compilation (compile, link). Behind the scenes, it generates bitcode instead of object files, then links the bitcode files, randomizes the linked module, and turns it back into object code for normal linking as described above. Next, runclang will need to be placed somwhere in accessible by the path environment variable, then when an autoconf build is performed of a source package, provide it the following values for the configuration settings:

```
CC=runclang
CCAS=runclang
CCASFLAGS=--asm
CFLAGS=--compile
LDFLAGS=--link
```

The configure command will then look something like this:

```
CC=${CC} CCAS=${CCAS} CCASFLAGS=${CCASFLAGS} CFLAGS=${CFLAGS}
LDFLAGS=${LDFLAGS} ./configure --prefix=$PREFIX --target=x86_64
```

After that, issuing 'make' and 'make install' as usual and it will work.

Alternatively, the optimization pass can be run manually if desired. The optimization pass runs on an llvm bitcode file. To create one, first compile the source code with clang:

```
clang -c -emit-llvm -o <filename.bc> <filename.c>
```

This will generate the file <filename.bc>, which is an LLVM bitcode file. Now, run llvm's optimizer on the bitcode as follows:

```
opt -load FuncSwap.so -funcswap -o<filename-opt.bc>
<filename.bc>
```

This will randomize the function ordering in <filename.bc> and put the resulting file in `<filename-opt.bc>.`

To compile that further, first turn it into an object file with clang:

```
clang -c -o <filename.o> <filename-opt.bc>
```

After that, treat it just like any other object file for linking purposes.

One compilation produces the following list of addresses along with the associated functions in memory:

```
00000000000813da T __isgraph_l
00000000000a0912 T __islower_l
000000000008cf1e W __isoc99_fscanf
000000000000f7e7 W __isoc99_fwscanf
0000000000020570 W __isoc99_scanf
000000000000472c W __isoc99_sscanf
0000000000030884 W __isoc99_swscanf
00000000000001f4 W __isoc99_vfscanf
000000000006d01d W __isoc99_vfwscanf
```

While a separate compilation produces the following locations of functions:

```
00000000000844f0 T __isgraph_l
0000000000082c0e T __islower_l
0000000000018b4a W __isoc99_fscanf
0000000000015ac7 W __isoc99_fwscanf
000000000002b922 W __isoc99_scanf
0000000000038156 W __isoc99_sscanf
00000000000863f4 W __isoc99_swscanf
0000000000007cf7 W __isoc99_vfscanf
0000000000043b61 W __isoc99_vfwscanf
```

As seen from above, the two instances compiled yield different binaries.

2.3 Future Work

This prototype established the infrastructure necessary to support randomized compilation and distribution, and implemented one of the randomizing transforms. There are many more transforms to be implemented, each of which will combine with the others to make transmission of malicious code vastly more difficult with little or no cost in terms of performance in the application itself. Future work in this area will be focused on developing more randomizing transforms, and on making sure that the resulting code functions well when combined with other security features like non-executable stacks, stack canaries, ASLR, and ISR.

3. DYNAMIC DEFENSE

Dynamically defending a network against an active attack has many similarities to a chess master. Chess masters are amongst the best chess players in the world and often succeed by quickly recognizing patterns combined with strategically mastering the opening and middle moves of a game. Once a pattern is recognized, the chess master draws from the experience gained from many games to deploy appropriate response strategies. The same strategies apply to successfully defending critical infrastructure systems. Critical Infrastructure control systems often harness unpatched (legacy and modern) systems that allow easy access to our Nation's most critical assets making them attractive and easy targets for cyber attack. On the other hand, critical infrastructure systems communicate and execute programs in fairly predictable patterns relative to traditional IT based systems lending themselves well to dynamic defense strategies. Specifically, the dynamic defense project aims to better secure control systems by recognizing and dynamically defending them against active attacks.

3.1 Framework

Currently, it is extremely difficult to detect an attack until it is too late. We have developed a machine learning framework that recognizes active attack patterns in near real time. This framework, shown in Figure 1, is capable of recognizing known attack vectors and also of generalizing knowledge from known attacks to recognize new attacks. In our implementation, we respond to these attacks by transforming our networks and applications into moving targets by applying the techniques implemented from the network randomization project. The framework itself is dynamic, allowing the specifics of the detection mechanism to change over time. Each component in our framework is described in the following sections.

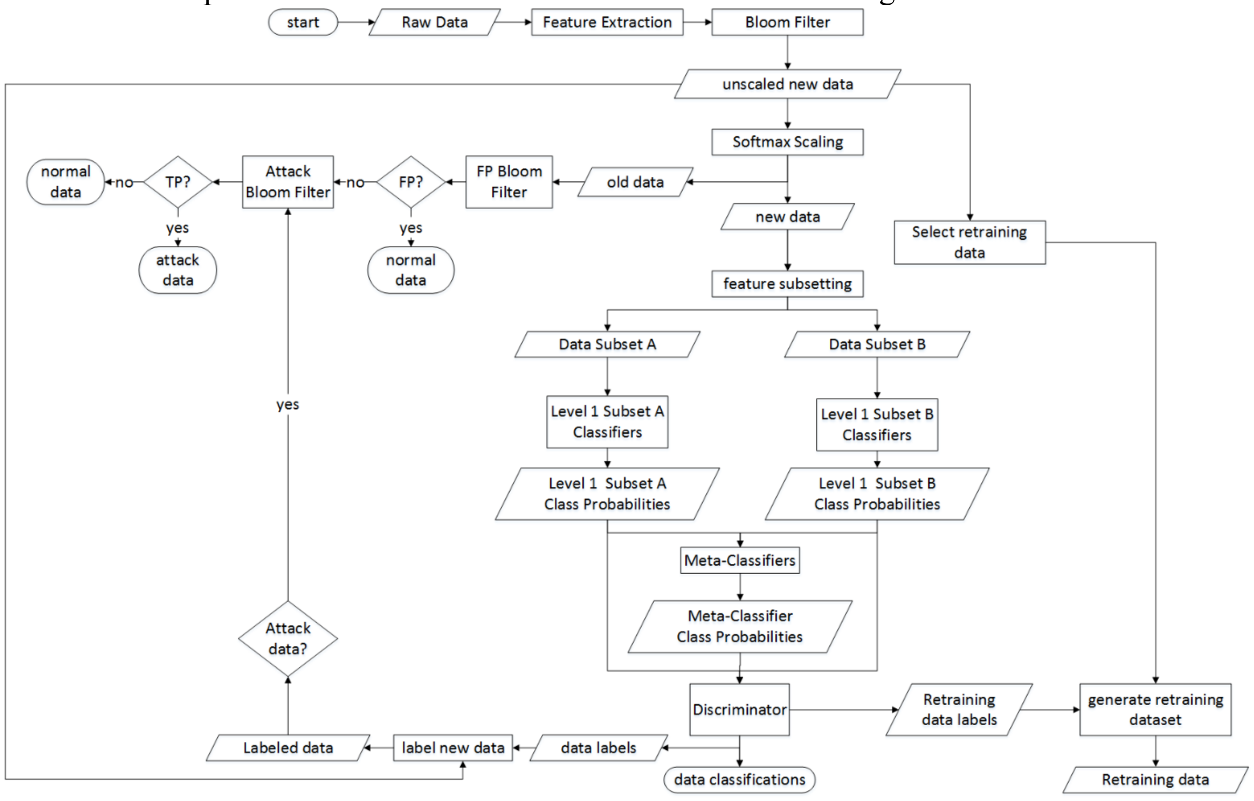

Figure 1: Our framework developed to classify traffic into anomalous or normal categories.

3.1.1 Raw Data

Initially, features are extracted from raw input data. For our experiments, we employed the Kyoto 2006+ dataset, which consists of features extracted from network traffic collected at Kyoto University from a regular mail and DNS server and several honeypots between 2006 and 2009 [1]. The vectors in this dataset consist of 14 features selected from the features in the KDD Cup '99 dataset, plus source and destination port and IP address, detection results from three intrusion detection systems, and the start time and duration of the session [2]. We use only the first 14 features:

- *duration*: length (number of seconds) of the connection
- *service*: network service on the destination, e.g., http, telnet, etc.
- *src_bytes*: number of data bytes from source to destination
- *dst_bytes*: number of data bytes from destination to source
- *count*: number of connections to the same host as the current connection in the past two seconds
- *same_srv_rate*: % of connections in the count feature to the same service
- *serror_rate*: % of connections in the count feature that have ``SYN'' errors
- *srv_serror_rate*: % of connections whose service type is the same to that of the current connection in the past two seconds that have "SYN" errors
- *dst_host_count*: among the past 100 connections whose destination IP address is the same to that of the current connection, the number of connections whose source IP address is also the same to that of the current connection
- *dst_host_srv_count*: the number of connections in the dst_host_count feature whose service type is also the same to that of the current connection
- *dst_host_same_src_port_rate*: % of connections in the dst_host_count feature whose source port is the same to that of the current connection
- *dst_host_serror_rate*: % of connections in the dst_host_count feature that have "SYN" *dst_host_srv_serror_rate*: % of connections in the dst_host_srv_count feature that "SYN" errors
- *flag*: normal or error status of the connection

since they can be extracted from any system. From these 14 features, we derived several additional features. Inclusion of these features improves the performance of the machine learning algorithms:

- *total_bytes*: *src_bytes* + *dst_bytes*
- log10(*src_bytes*)
- log10(*dst_bytes*)
- *src_bytes* / *duration*
- *dst_bytes* / *duration*
- *total_bytes* / *duration*
- *log_duration*: log10(*duration*)
- *count* / *duration*
- *dst_host_count* / *duration*
- *dst_host_srv_count* / *duration*

- *dst_bytes * dst_host_count:*
- *dst_bytes * dst_host_srv_count*
- *same_srv_rate * srv_serror_rate*
- *same_srv_rate * dst_host_count*
- *same_srv_rate * dst_host_srv_count*
- *srv_serror_rate * dst_host_srv_count*
- *dst_host_count * dst_host_srv_count*

Among these, several other features were extracted but were not used as they only contributed to a small performance increase or none at all for the machine learning algorithms. These features are as follows:

- *unix_time: unix timestamp for the connection*
- *orig_p: the originating port*
- *resp_p: response (destination) port*
- *orig_h: originating host*
- *resp_h: response (destination) host*
- *protocol_type: protocol type of the connection: TCP, UDP and ICMP*
- *flag: connection status. The possible status for this are: SF, S0, S1, S2, S3, OTH, REJ, RST0, RSTOS0, SH, RSTRH, SHR*
- *land: if the source and destination addresses and port numbers are equal then, this variable takes the value 1 else 0*
- *wrong_fragment: sum of bad checksum packets in a connection*
- *urgent: sum of urgen packets in a connection. Urgent packets are packets with the urgent bit activated*
- *srv_count: sum of connections to the same destination port number*
- *rerror_rate: % of connections that have activated the flag REJ, among the connections aggregated in count*
- *diff_srv_rate: % of connections that were to different services among the connections aggregated in count*
- *srv_diff_host_rate: % of connections that were to different destination machines among the connections aggregated in srv_count*
- *dst_host_same_srv_rate: % of connections that were to the same service, among the connections aggregated in dst_host_count*
- *dst_host_diff_srv_rate: % of connections that were to different services, among the connections aggregated in dst_host_count*
- *dst_host_srv_diff_host_rate: % of connections that were to different destination machines among the connections aggregated in dst_host_srv_count*
- *dst_host_rerror_rate: % of connections that have activated the flag REJ, among the connections aggregated in dst_host_count*
- *dst_host_srv_error_rate: % of connections that have activated the flag REJ, among the connections aggregated in dst_host_srv_count*

Of course, the framework can be used with other types of data. In particular, host based features, such as system performance statistics and statistics derived from sequences of system calls, can also be used [3]. We represent the service and flag features with a one-hot binary encoding.

This raw data set is used to train our machine learning algorithms for future classification of unknown traffic and should be representative of what is observed in normal operations. Although we used a readily available dataset for our proof-of-concept implementation, actual traffic of the system where the dynamic defense technology will be deployed should be used for training the machine learning algorithms.

3.1.2 Bloom Filters

After feature extraction we process the data with a sequence of Bloom filters. Bloom filters are a probabilistic method for determining the newness of data. A Bloom filter consist of an array in which all values are initialized to zero. When data is received by the BBloom filter, the data is first hashed by several hash functions. The hashed values are then used as indices into the BBloom filter array. At each index pointed to by a hashed value, the value in the Bloom filter is set to 1. With 100% probability old data will have a '1' at each index, while with high probability that is adjustable by controlling the parameters of the Bloom filter, the new data will have a '0' at one or more indices [5].

We use Bloom filters for data reduction and to help control the false positive and false negative rates. The first Bloom filter separates new data from old so that we only have to analyze new data with the machine learning models. Old data is then sent to a Bloom filter containing feature vectors from false positives. If the data is found in this filter, then it is labeled as being normal data. If the data is not in the false positives Bloom filter, then it is processed by a Bloom filter containing feature vectors from attack data. If the data is found in this filter then it is labeled as being attack data, otherwise it is labeled as being normal data. The false positive and attack Bloom filters are initially populated with known vectors, with the class label removed, from a labeled training set. During operation, all feature vectors are added to the initial Bloom filter, and any vectors classified as attacks are added to the attack Bloom filter, again with the class labels removed. In practice, we envision that, in addition to these automated procedures, an analyst could maintain the false positives and attacks Bloom filters. In support of this, our framework adds all feature vectors classified as attacks to a database. In practice, the raw data associated with these vectors would also need to be stored.

3.1.3 Softmax Scaling

Next, all new data is softmax scaled as a preprocessing step. Softmax scaling is a normalization approach that does minimal harm to the information content of the input dataset, Softmax scaling compresses all numerical values to the range [0,1].The transformation is linear at the mean but has smooth nonlinearities at both extremes of the range. This helps the machine learning models process data that is outside of the ranges observed during training [4].The normalized data, v_n, is calculated as $v_n = \frac{1}{1+e^{-v_t}}$, where $v_t = \frac{v_i - mean(v_i)}{std\,dev(v_i)/2\pi}$ and v_i is the unscaled data. There is one v_t for each continuous feature, and new v_t values are calculated during retraining cycle. The softmax scaled data is then provided as input for the Bloom filters.

3.1.4 Generating Retraining Datasets

Our framework generates datasets for periodic retraining. Retraining data consists of a mix of old data from previous training or retraining datasets, and of newer data processed since the last retraining cycle. In our current implementation retraining occurs after 200,000 new feature vectors have been processed. Of these new feature vectors, approximately 6% are considered for inclusion in the retraining dataset. Of these 6%, with 80% probability we add the vector to the retraining dataset. The remaining 20% of the time we uniformly at random choose a feature vector from the previous training or retraining dataset for inclusion. Additionally, we target our retraining datasets to consist of an even balance of 5000 attack vectors and 5000 normal vectors. To achieve this balance, we check for deviations from it immediately before retraining. If more than 60% of the retraining dataset has the same class label then we remove some vectors and replace them with vectors from the previous training or retraining dataset to achieve a balance. Since labeled datasets are required for training, we label the new data with the classification provided by our models.

3.1.5 Feature Subsetting

The feature vectors are split into two subsets by uniformly at random assigning each feature to one or both subsets. The one-hot encoded representations of the service and flag features are included in both subsets. The benefit to splitting the feature vectors into two feature sets is to attempt to provide independence between the models trained on the different feature sets. When the models trained on the different feature sets are later combined into an ensemble, if the classification errors of the subsets of models are uncorrelated, due to their independent feature sets, then the final ensemble can have better performance than any of the individual classifiers [6].

3.1.6 Boosted Classifiers

We train an ensemble of classifiers on each of the feature subsets. Each ensemble contains one each of the Naïve Bayes, logistic regression, support vector machine (SVM), and random forest classifiers [7,8]. We name these “level 1 classifiers”. We use the Orange Data Mining Toolbox for our implementations of the models [9]. Each of these classifiers has a number of configurable hyper-parameters. For instance, in Naïve Bayes we modify the probability estimator used for estimating prior class probabilities amongst m-estimates, Laplace estimation, and relative frequency estimation, and can also choose whether to tune the classification threshold for choosing between classes. In SVM we can select amongst different SVM types, kernel types, various parameters for the different SVM types and kernels, and termination criteria. In logistic regression we modify the solver type and regularization parameter, and with random forests we vary the number of trees in the forest and the number of features to consider when determining where to split the nodes during tree induction [9].

We perform a random search to optimize selection of these hyper-parameters [10]. In each iteration of the search we randomly assign values to the hyper-parameters for the classifier under consideration, and then create a boosted learner consisting of 30 learners with these hyper-parameters [13]. The resulting boosted learner is then trained on half of the training data and tested on the other half. The division of the training data is random and differs for each of the machine learning techniques. The learners are evaluated using Matthew’s correlation coefficient (MCC) [11]. In terms of true positives (TP), true negatives (TN), false positives (FN), and false negatives (FN), the MCC is defined as $MCC = \frac{TP \times TN - FP \times FN}{(TP+FP)(TP+FN)(TN+FP)(TN+FN)}$ and has range [-1,1] with a score of 1 indicating perfect performance, 0 indicating no better than chance, and -1 indicating complete inaccuracy in prediction. The random search is terminated if an iteration's score differs from the previous maximum by less than 0.001 and the maximum score exceeds 0.7, or when a maximum of 15 iterations are reached. Experimentally, we find there to be little gain from introducing additional iterations to the search.

We always seek to optimize the performance of the ensemble, rather than that of the individual classifiers [12]. As such, the optimization begins with an empty ensemble and searches for an optimal hyper-parameter selection for the Naïve Bayes classifier. Then, the logistic regression, SVM, and random forest classifiers are added, in turn, to the ensemble. Combining heterogeneous classifiers into an ensemble in this way has been shown to increase prediction accuracy [14].

3.1.7 Boosted Meta-Classifiers

The classification probabilities from our two level-1 classifiers are input to an ensemble of meta-classifiers. During training we randomly choose to include between 8 and 15 classifiers in our ensemble of meta-classifiers. Each of the meta-classifiers is randomly chosen to be a Naïve Bayes, logistic regression, SVM, or random forest. Each of the meta-classifiers takes as input the classification probabilities from a random selection of two to six of the level-1 classifiers. As with the level-1 classifiers, we perform a random search for hyper-parameter optimization, form a boosted learner containing 30 learners with these hyper-parameters, and use MCC to score the models.

3.1.8 Boosted Discriminator

The classification probabilities from the two blended level-1 ensembles and from each of the meta-classifiers are input to a random forest for final classification. As always, we perform a random search for hyper-parameter optimization and form a boosted collection of 30 learners with these hyper-parameters. During testing, the output from this discriminator is used as the classification for each feature vector. These classifications are then used to label the (unscaled) testing data so that it can be incorporated into the retraining dataset and, as necessary, added to the attacks Bloom filter.

3.2 Dynamic Aspects

There are many dynamic aspects of our framework. For example, one can randomize the selection of subsets of features for the level-1 ensembles. The number of level-1 ensembles can also be varied. We can adjust the number and type of classifiers in the level-1 ensembles and also the number and types of meta-classifiers. The selection of retraining data is randomized, and the balance of new and older data in the retraining datasets can be varied. We can also vary the frequency of retraining. We can also train several distinct copies of the framework and randomly choose which of the copies to use to classify incoming data. All of this randomization should frustrate adversaries that attempt to inject malicious data to train the models to classify adversarial data as being normal.

3.3 Results

We have tested our framework with the Kyoto 2006+ dataset. Our initial training was a balanced training set of 6132 samples (3066 attack vectors and 3066 normal vectors) selected by randomly 1000 attack samples and 1000 normal samples each from the 01 January 2007 to 07 January 2007 data. We then removed duplicate entries from this dataset, and then balanced the result so that there would be equal numbers of attack and normal vectors. Retraining datasets were generated using the procedure described in Section 3.2.4, making these 6132 samples the only ground-truth labeled data used in our experiment. We used the 08 January 2007 to 31 August 2009 data as our test set. In total, the test set consisted of 89,087,080 samples. Our results are presented in Table 1 where they are also compared to previous work [15,16]. In Table 1 the AUC score is the area under the receiver operating characteristic curve. The AUC indicates the probability that a classifier will rank a random positive instance higher than a random negative instance, so it represents a tradeoff between precision and recall.

Table 1: Results from testing our model on data from 08 January 2007 to 31 August 2009 (ensemble A), from testing it on the subset of data as in [15] and [16] (ensemble B), from a signature based IDS included int he dataset (Signature IDS), and from prior work (Anomaly Detection, Maximum Entropy, Linear SVM).

Classifier	MCC	Recall	False Positive Rate	AUC Score
Ensemble A	0.9355	0.9889	0.0495	0.9697
Ensemble B	0.9456	0.9932	0.0347	0.9829
Signature IDS	N/A	0.0900	0.0162	N/A
Anomaly Detection [15]	N/A	0.8093	0.0590	N/A
Maximum Entropy [16]	N/A	0.7729	0.0206	0.7204
Linear SVM [16]	N/A	0.9895	0.0353	0.9630

4. REFERENCES

1. Song, Jungsuk, et al. "Statistical analysis of honeypot data and building of Kyoto 2006+ dataset for NIDS evaluation." Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security. ACM, 2011.
2. The third international knowledge discovery and data mining tools competition dataset, KDD99-Cup, 1999. Available: http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.
3. Haas, Jason J., J. D. Doak, and Jason R. Hamlet. "Machine-oriented biometrics and cocooning for dynamic network defense." Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop. ACM, 2013.
4. Pyle (1999). Data Preparation for Data Mining. pp. 271–274, 355–359.
5. Bloom, Burton H. "Space/time trade-offs in hash coding with allowable errors." Communications of the ACM 13.7 (1970): 422-426.
6. Dietterich, Thomas G. "Ensemble methods in machine learning." Multiple classifier systems. Springer Berlin Heidelberg, 2000. 1-15.
7. Cortes, Corinna, and Vladimir Vapnik. "Support-vector networks." Machine learning 20.3 (1995): 273-297.
8. Breiman, Leo. "Random forests." Machine learning 45.1 (2001): 5-32.
9. Demšar, J., Curk, T., & Erjavec, A. Orange: Data Mining Toolbox in Python; Journal of Machine Learning Research 14(Aug):2349−2353, 2013.
10. Bergstra, James, and Yoshua Bengio. "Random search for hyper-parameter optimization." The Journal of Machine Learning Research 13.1 (2012): 281-305.
11. Matthews, Brian W. "Comparison of the predicted and observed secondary structure of T4 phage lysozyme." Biochimica et Biophysica Acta (BBA)-Protein Structure 405.2 (1975): 442-451.
12. Töscher, Andreas, Michael Jahrer, and Robert M. Bell. "The bigchaos solution to the netflix grand prize." Netflix prize documentation (2009).
13. Schapire, Robert E. "The strength of weak learnability." Machine learning 5.2 (1990): 197-227.
14. Borji, Ali. "Combining heterogeneous classifiers for network intrusion detection." Advances in Computer Science–ASIAN 2007. Computer and Network Security. Springer Berlin Heidelberg, 2007. 254-260.
15. Kishimoto, Kazuya, Hirofumi Yamaki, and Hiroki Takakura. "Improving performance of anomaly-based ids by combining multiple classifiers." Applications and the Internet (SAINT), 2011 IEEE/IPSJ 11th International Symposium on. IEEE, 2011.
16. Symons, Christopher T., and Justin M. Beaver. "Nonparametric semi-supervised learning for network intrusion detection: combining performance improvements with realistic in-situ training." Proceedings of the 5th ACM workshop on Security and artificial intelligence. ACM, 2012.
17. POX (About POX | NOXRepo) http://www.noxrepo.org/pox/about-pox/
18. Open vSwitch http://openvswitch.org/
19. OpenFlow Protocol, Open Networking Foundation https://www.opennetworking.org/sdn-resources/openflow

APPENDIX A: TVA QUESTIONS

Throughout the life of the project, we regularly communicated with Tennessee Valley Authority (TVA) for guidance and to consult on questions that would drive our R&D. A compilation of the questions and answers are listed below:

TVA Staff: John W Stewart, 423-751-4582, jwstewart@tva.gov

1. What does a typical attack or concern look like (spear-phishing, malware installed after insertion of a rogue USB drive, etc.)
Same as what you would see/expect on a typical IT system

2. What types of computing resources and constraints will we have?
Varies from relays to high power systems.

3. What kinds of interactions occur between the control system and the larger enterprise network?
Separate networks but some links exist for remote configuration

4. What data (SCADA) is currently collected?
SNMP is collected, voltage readings, syncrophaser, ...

5. What types of data can we collect (from PLCs, supervisory systems, etc)?
Can possibly get traffic captures of real network traffice

6. What kind of tools and processes monitor/debug/manage your existing networks?
A variety of graphical tools to monitor syslog and snmp messages.

7. What is the basic network topology (abstract)?
- Physical and logical composition and/or separations

Control system separated from corporate network

8. Supporting network infrastructure?
- Longhaul connectivity
- Layer 3 devices (local or distributed)
- Layer 2 devices (local)
- Physical layer devices
    - Legacy devices (modems, line drivers, CSU/DSU)
    - Interface standards

9. Typical End devices/Concentrators?
- Near-/Real-time requirements

10. Servers/Services?
- Specific Platforms
- Specific Services/Applications
- Multi-homing
- Distributed Systems/Services

11. Common attacks/perceived attack vectors?
stuxnet and other attacks that you would see in traditional IT systems

12. How many end devices (PLC, RTU, ...)?
On the order of thousands

13. Are most addresses statically configured or DHCP?
Statically configured

14. Is there a concentrator for network events/logs?
Yes, several

15. What is the maximum acceptable delay of a communication stream?
It varies depending on the criticality of the process or if needed to meet CIP requirements

16. What is done now to update/patch field devices?
Planned in advance and thoroughly tested within laboratory beforehand to evaluate any potential negative side effects

17. How much downtime can be afforded?
It varies based on the criticality of the process or if needed to meet CIP requirements

DISTRIBUTION

1	MS0672	Adrian Chavez	05629
1	MS0672	William Stout	05629
1	MS0671	Jason Hamlet	05627
1	MS0672	Mitchell Martin	05629
1	MS0672	Erik Lee	05629 (electronic copy)
1	MS0672	Han Lin	05629 (electronic copy)
1	MS0671	Jennifer Depoy	05628 (electronic copy)
1	MS0671	Alexander Roesler	05627 (electronic copy)
1	MS0899	Technical Library	9536 (electronic copy)

For Patent Caution reports, add:

1	MS0161	Legal Technology Transfer Center	11500

Sandia National Laboratories