HOWTO: Creating and Running IRON Experiments

1. Preface

To facilitate development and testing of IRON, we created a set of automation scripts and tools that we use to configure and run experiments, as well as gather results from experiments and perform various analyses. This document describes:

- How we create a flexible and composable network emulation tesbed
- What is needed to configure and run an IRON experiment using the testbed
- The various scripts and tools supporting this process, and
- How these scripts and tools are used, including a simple example

2. Test Network Emulation

As described in the project overview and illustrated in Figure 1 below, IRON functions as a overlay network that interconnects two or more user sites using a public network infrastructure as the underlay. The user sites (red clouds) are cryptographically isolated from the public network (black cloud) using encryption devices (orange boxes) located at the edges of the public network. IRON nodes (blue circles) are placed between the user site hosts and the encryptors, and use over overlay tunnels (in green) to steer traffic along alternate underlay paths when one or more of the direct underlay paths are impaired or otherwise unusable.

FIgure 1. The network model for the EdgeCT program consists of distributed secure user sites cryptographically isolated from the third party network. In-line devices (shown here as IRON nodes) developed under the program provide improved performance and resilience to misconfigurations and attacks within the third party network.

Emulating this environment for development and testing purposes consists of three main networking components:

1. Sources and sinks of application traffic
2. IRON nodes, which are functionally equivalent to routers
3. Network emulation to model base path characteristics and to dynamically inject or remove impairments along those paths.

In our testbed we organize a set of three hosts into an enclave emulation unit; interconnecting multiple enclave emulation units via an ethernet switch allows us to easily compose test networks with varying numbers of enclaves. As shown in Figure 2, each enclave emulation unit consists of:

1. An App Node running on the first of the three hosts that is used to source and

This document is subject to the restrictions on the cover page.

sink all application traffic for the enclave.

2. An IRON node running on the second of the three hosts that provides the EdgeCT services. Each IRON node is configured with two separate WAN-facing interfaces to support dual-homed operation as a means for providing further resilience. This allows an IRON node to be connected to two distinct networks (e.g., CMNT and SIPRNet, or Comcast and Fios).
3. A WAN emulation node running on the third of the three hosts. The WAN emulation host consists of two separate path emulators, one for modeling the network paths for each network, and a support router. The support router provides the means for routing traffic between different enclaves.

Note that each of the three hosts is connected to a common experiment control network that is used for loading software, starting and stopping experiments, and retrieving experimental artifacts. The control interfaces to each of the enclave hosts shown here are not shown in subsequent drawings for simplicity.

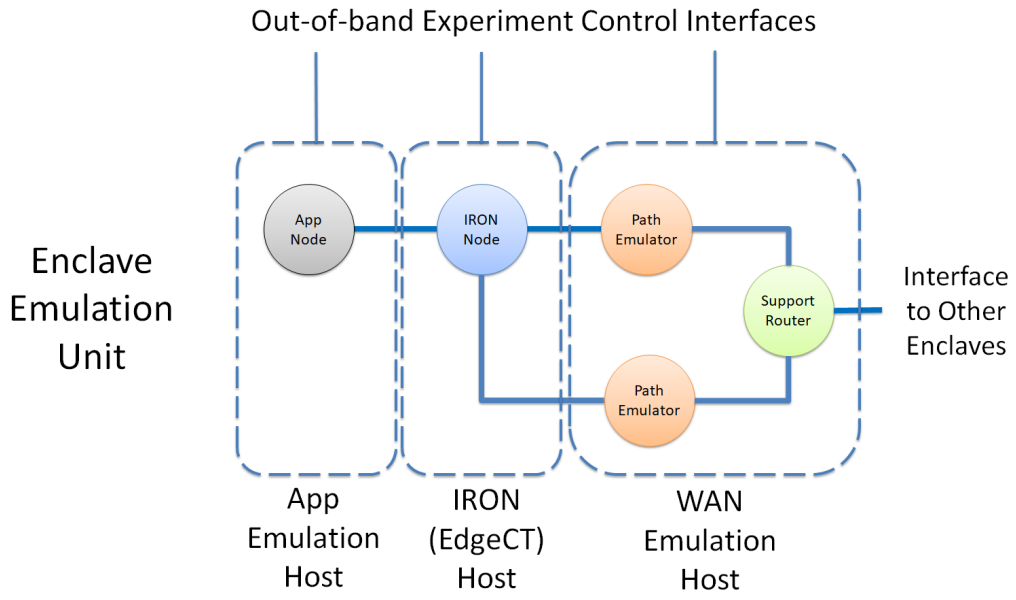

FIgure 2. Our test network infrastructure is organized into enclave emulation units. Interconnecting two or more of these units via an ethernet switch allows creating test networks with varying numbers of enclaves as needed.

Note also that we use a separate physical interface for each of the connections shown in Figure 2. Hence the App Emulation Host requires two physical interfaces, the IRON Host uses four physical interfaces, and the WAN emulation host uses eight physical interfaces.

This in turn requires that routing in each of the three enclave emulation hosts be configured so that the underlay network being modeled works correctly without any

IRON software or without any link/path emulation software running. Hence the app node in one enclave can communicate with the app node in any other enclave. Moreover, since the path emulators are functionally equivalent to ethernet bridges, the physical interfaces associated with each path emulator are typically bridged together when the path emulation software is not running.

We typically assign the four primary IRON components to separate CPU cores (if available). We typically assign each path emulator to a separate core as well (our path emulators use busy wait loops to provide highly accurate timing). Hence it is recommended that physical hosts used as IRON hosts and WAN emulation hosts each have at least four cores.

3. Testbeds

Multiple enclave emulation units are interconnected using a large Ethernet switch to create an experiment testbed, an example of which is shown in the Figure 3 below:

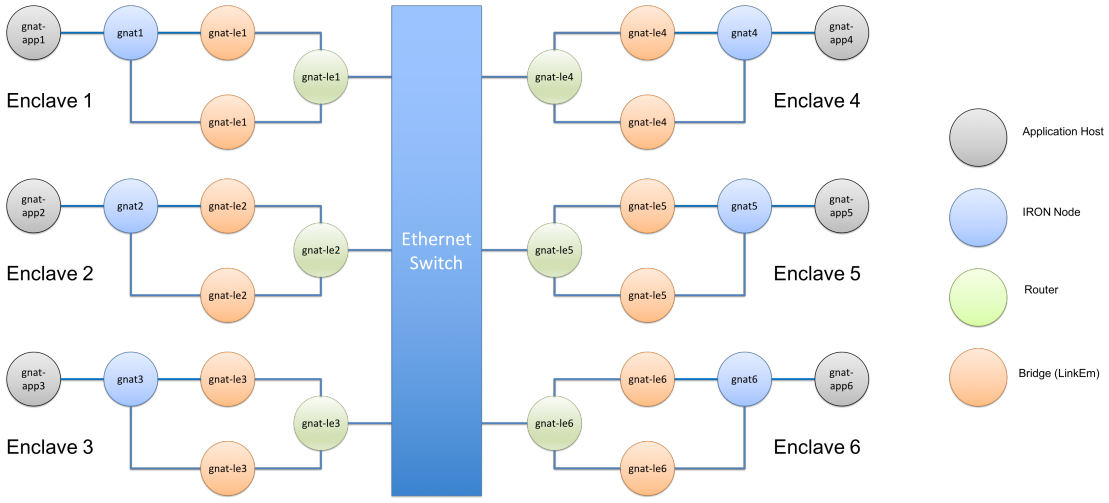

Figure 3: Enclave emulation units are hosted on real physical nodes. Enclaves are interconnected using a large Ethernet switch.

Organizing the testbed in this fashion enables different combinations of enclaves to support an experiment. For example, consider a 3 enclave experiment that is run multiple times. Run 1 might occur on Enclaves 1, 2, and 3 and subsequent runs might occur on Enclaves 1, 5, and 6, Enclaves 2, 3, and 4, and Enclaves 1, 3, and 5. The physical organization of the testbed permits the use of *any* 3 enclaves to support our hypothetical 3 enclave experiment. If an experimenter wishes to run on the same enclaves (for example, to do some regression testing on a fixed set of hardware), this can be accomplished. When an experiment is run, the experimenter can request either the explicit enclaves desired or the number of required enclaves. In either case, the real physical enclaves reserved for the experiment are provided

to the automated experiment execution scripts. Creating and running experiments will be described in the *Creating A New Experiment* and *Experiment Execution* sect ions below.

Note that while dual homing is supported by the above testbed, it need not be used for all experiments.

3.1. Configuring Testbed Nodes

Once organized into enclave units, our testbed nodes require some configuration before they are ready for hosting an IRON experiment. The interfaces on the testbed nodes must be configured, **sudo** must be properly configured to work with the automated experiment execution scripts, and a minimum set of applications need to be installed on the application nodes.

Note that the IRON software has been tested on Ubuntu 14.04 and Ubuntu 16.04. Utilization of other versions of Linux is an exercise left to the reader.

3.1.1. Interface Configuration

As previously described, the routing in the testbed nodes is configured so that the underlay network being modeled works correctly without any IRON software or without any link/path emulation software running. Hence the application node in one enclave can communicate with the application nodes in all other enclaves. Figu re 4 depicts our example physical testbed with all of the interface addresses identified. Note that there are many ways that the addressing could be assigned and that this only serves as an example of how we have configured our testbed.

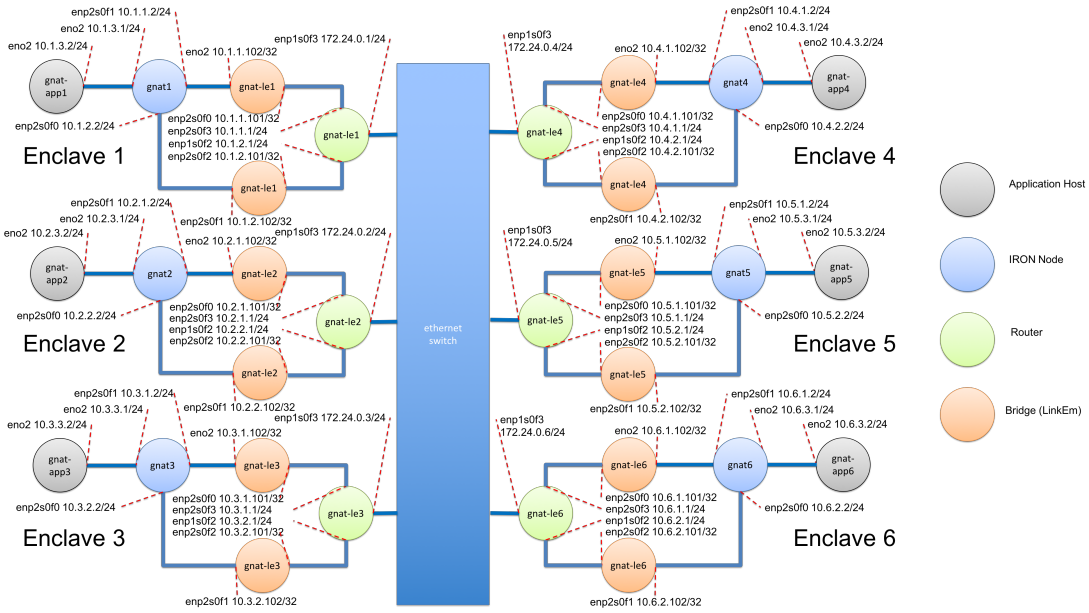

Figure 4: Physical testbed showing interface addresses

In addition to configuring the interface addresses, we set up the necessary static routes in the */etc/network/interfaces* file for each of the testbed nodes. The configuration for the App Emulation Host, **gnat-app1**, is depicted below:

gnat-app1 Interface File

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eno1
iface eno1 inet static
   address 128.89.151.170
   gateway 128.89.151.1
   netmask 255.255.254.0

# The application network interface
auto eno2
iface eno2 inet static
      address 10.1.3.2
      netmask 255.255.255.0
      mtu 1350
      up route add -net 10.0.0.0/8 gw 10.1.3.1 dev eno2
      up route add -net 172.24.0.0/16 gw 10.1.3.1 dev eno2
```

The configuration for the IRON Host, **gnat1**, is depicted below:

gnat1 Interface File

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eno1
iface eno1 inet static
   address 128.89.151.201
   gateway 128.89.151.1
   netmask 255.255.254.0

auto eno2
iface eno2 inet static
      address 10.1.3.1
      netmask 255.255.255.0
#     up route add -net 10.1.3.0/24 gw 10.1.3.2 dev eno2

auto enp2s0f1
iface enp2s0f1 inet static
      address 10.1.1.2
      netmask 255.255.255.0
      up route add -net 10.0.0.0/8 gw 10.1.1.1 dev enp2s0f1
      up route add -net 172.24.0.0/24 gw 10.1.1.1 dev enp2s0f1

auto enp2s0f0
iface enp2s0f0 inet static
      address 10.1.2.2
      netmask 255.255.255.0
      post-up /sbin/ip route flush table 2
      post-up /sbin/ip route add default via 10.1.2.1 dev enp2s0f0 table
2
      post-up /sbin/ip rule add from 10.1.2.2 table 2
      post-down /sbin/ip rule del from 10.1.2.2 table 2
```

The configuration for the WAN Emulation Host, **gnat-le1**, is depicted below:

gnat-le1 Interface File

```
# The loopback network interface
auto lo
iface lo inet loopback

# The primary network interface
auto eno1
iface eno1 inet static
   address 128.89.151.211
   gateway 128.89.151.1
```

```
   netmask 255.255.254.0


auto br_2s0f0_eno2
iface br_2s0f0_eno2 inet static
      address 10.115.11.1/32
      bridge_ports eno2 enp2s0f0
      post-up ip address add 10.1.1.101/32 dev enp2s0f0
      post-up ip address add 10.1.1.102/32 dev eno2

auto br_2s0f2_2s0f1
iface br_2s0f2_2s0f1 inet static
      address 10.115.12.1/32
      bridge_ports enp2s0f1 enp2s0f2
      post-up ip address add 10.1.2.101/32 dev enp2s0f2
      post-up ip address add 10.1.2.102/32 dev enp2s0f1

# The following are taken care of in the above bridge config post-up
commands
#
#auto enp2s0f0
#iface enp2s0f0 inet static
#      address 10.1.1.101
#      netmask 255.255.255.255

#auto eno2
#iface eno2 inet static
#      address 10.1.1.102
#      netmask 255.255.255.255

#auto enp2s0f2
#iface enp2s0f2 inet static
#      address 10.1.2.101
#      netmask 255.255.255.255

#auto enp2s0f1
#iface enp2s0f1 inet static
#      address 10.1.2.102
#      netmask 255.255.255.255

auto enp2s0f3
iface enp2s0f3 inet static
      address 10.1.1.1
      netmask 255.255.255.0
#     up route add -net 10.1.1.0/24 gw 10.1.1.2 dev enp2s0f3
      up route add -net 10.1.3.0/24 gw 10.1.1.2 dev enp2s0f3

auto enp1s0f2
iface enp1s0f2 inet static
      address 10.1.2.1
      netmask 255.255.255.0
#     up route add -net 10.1.2.0/24 gw 10.1.2.2 dev enp1s0f2
```

This document is subject to the restrictions on the cover page.

```
auto enp1s0f3
iface enp1s0f3 inet static
      address 172.24.0.1
      netmask 255.255.255.0
#     up route add -net 10.1.0.0/16 gw 172.24.0.1 dev enp1s0f3
      up route add -net 10.2.0.0/16 gw 172.24.0.2 dev enp1s0f
      up route add -net 10.3.0.0/16 gw 172.24.0.3 dev enp1s0f3
      up route add -net 10.4.0.0/16 gw 172.24.0.4 dev enp1s0f3
      up route add -net 10.5.0.0/16 gw 172.24.0.5 dev enp1s0f3
      up route add -net 10.6.0.0/16 gw 172.24.0.6 dev enp1s0f3
      up route add -net 10.7.0.0/16 gw 172.24.0.7 dev enp1s0f3
      up route add -net 10.8.0.0/16 gw 172.24.0.8 dev enp1s0f3
      up route add -net 10.9.0.0/16 gw 172.24.0.9 dev enp1s0f3
      up route add -net 10.10.0.0/16 gw 172.24.0.10 dev enp1s0f3
      up route add -net 10.11.0.0/16 gw 172.24.0.11 dev enp1s0f3
      up route add -net 10.12.0.0/16 gw 172.24.0.12 dev enp1s0f3
      up route add -net 10.19.0.0/16 gw 172.24.0.19 dev enp1s0f3
      up route add -net 10.20.0.0/16 gw 172.24.0.20 dev enp1s0f3
      up route add -net 10.21.0.0/16 gw 172.24.0.21 dev enp1s0f3
```

This document is subject to the restrictions on the cover page.

```
up route add -net 10.22.0.0/16 gw 172.24.0.22 dev enp1s0f3
up route add -net 10.23.0.0/16 gw 172.24.0.23 dev enp1s0f3
up route add -net 10.24.0.0/16 gw 172.24.0.24 dev enp1s0f3
```

Recall that since the path emulators in the WAN Emulation Host are functionally equivalent to ethernet bridges, the physical interfaces associated with each path emulator are typically bridged together when the path emulation software is not running. The configuration for interface br_2s0f0_eno2, which bridges interfaces en p2s0f0 and eno2, and interface br_2s0f2_2s0f1, which bridges interfaces enp2s0f2 and enp2s0f1, accomplishes this.

When the automated experiment execution scripts start an experiment, the scripts need to bring down the bridge interfaces before running the path emulation software. When the experiment terminates, the path emulation software terminates and the bridge interfaces need to be brought back up. In order to do this, the automated experiment execution scripts must be able to identify the interfaces impacted by the experiment. Note that not all experiments will impact both bridge interfaces. For example, any experiment with a single-homed enclave will only impact one of the bridge interfaces. The automated experiment execution scripts require that the bridge interfaces be named in accordance to the following convention:

br_WANIF_LANIF

where *WANIF* is the name of the WAN-facing interface to be bridged and *LANIF* is the name of the LAN-facing interface to be bridged. If the name of either of the these interfaces is longer than 5 characters, then the last 5 characters of the interface name are used, as this is typically all that is necessary to uniquely identify the interface. This is required because there is a limit on the length of the bridge interface names, typically 16 characters. When the link emulation software starts, a reference address is passed into the scripts that enables the scripts to determine the LAN-facing and WAN-facing interfaces that are part of the affected bridge. The reference address is automatically provided when an experiment is created using the experiment creation scripts described in the *Creating A New Experiment* sectio n below and does not need to be provided by the user. For a more detailed explanation of the operation and capabilities of the link emulation software, see her e.

3.1.2. Configuring sudo

The IRON components require root privileges when executed. This can be accomplished by adding the appropriate users to the */etc/sudoers* file on the testbed nodes. Additionally, the automated experiment execution scripts generally start the IRON components from a remote execution node via ssh commands.

While it is possible, it is generally very inconvenient to have to provide a password for every sudo command that is executed as there are many during the course of starting and stopping an experiment. It is best to set up sudo on the testbed nodes so that no password is required. We typically run all experiments as the **iron** user. The following lines added to the */etc/sudoers* files give the **iron** user sudo privileges and indicate that no password is required when the **iron** user issues the sudo command:

```
# all the other users
iron ALL=(ALL:ALL) NOPASSWD: ALL
```

3.1.3. Installing Applications

3.1.3.1. Installing mgen

The automated experiment execution scripts use *mgen* as the source and destination application for the experiment's traffic flows. If not installed on the testbed application nodes, this can be accomplished with the following command:

sudo apt-get install mgen

Following the installation, issuing the command

mgen

on the testbed application node produces the following output:

```
mgen: version 5.02
mgen: starting now ...
13:00:33.522278 START Mgen Version 5.02
13:00:33.522335 STOP
```

3.1.3.2. Installing gnuplot

The automated experiment execution scripts provide an option to post process the experiment results into goodput, delay, and loss plots at the destination node for each of the experiment's flows. In order to generate these plots, the mgen output files are processed with the trpr application and the trpr output is then plotted with gnuplot. We modified the trpr application to support plotting large numbers of flows. As such, the trpr application is built with the IRON executables and is deployed when an experiment is run. However, gnuplot is also required to post process the

experiment results and is not installed when an experiment runs. If gnuplot is not installed on the testbed application nodes, this can be accomplished with the following command:

sudo apt-get install gnuplot5-qt

Following the installation, issuing the command

gnuplot --version

on the testbed application node produces the following output:

```
gnuplot 5.0 patchlevel 3
```

4. Testbed Abstraction

4.1. Generic Terminology For Testbeds

A design goal for the automated experiment execution scripts is that they can be used without modification to run experiments on different testbeds. The first step to accomplish this is to start thinking about the testbeds in generic terms, i.e., as set of generic nodes connected by links. Consider Figure 5, which is an abstraction of the Example Physical Testbed described in the previous section.

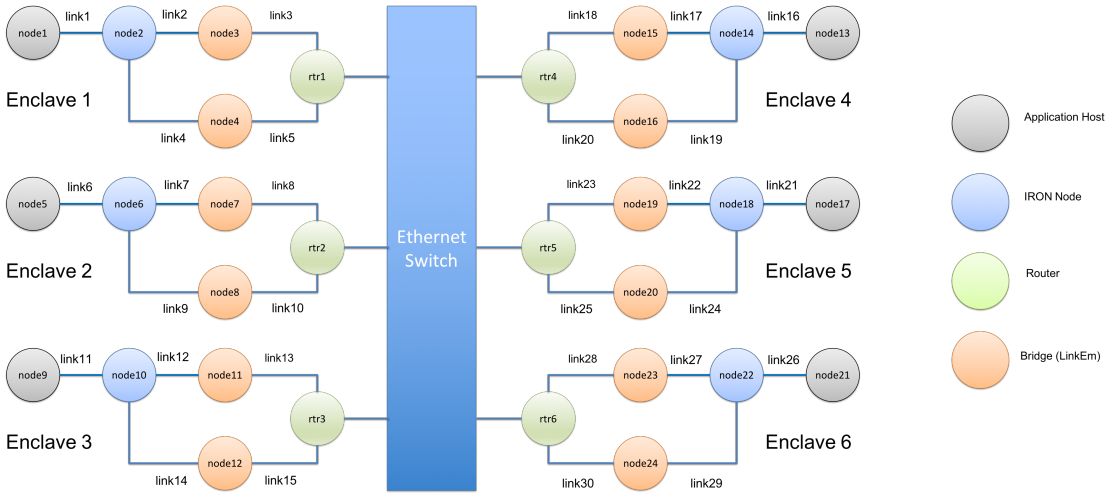

Figure 5: Abstraction of example physical testbed to generic nodes and links

One can now refer to the nodes and the node interfaces in the experiment configuration files in a generic fashion, for example node1 and node1:link1. Thinking about a testbed in this fashion provides an abstraction layer that provides the flexibility to run an experiment configured with these generic names to be run

on many different physical instantiations of the topology. We call this the "experiment topology".

Notice that our enclaves have four nodes in them consisting of one application node, one IRON node, and 2 link emulation nodes and 5 links connecting the nodes. A testbed topology defined in the above fashion has an *enclave node offset* of ((enclave id - 1) * 4) and a *enclave link offset* of ((enclave id - 1) * 5). Our experiment topology assigns generic node identifiers in the following way for each enclave:

- The application nodes are (*enclave node offset* + 1 through N), where N is the number of application nodes in the enclave. In the single application node example above, these are **node1** for **Enclave 1**, **node5** for **Enclave 2**, **node9** for **Enclave 3**, etc.
- The IRON node is (*enclave node offset* + (N+1)), **node2** for **Enclave 1**, **node 6** for **Enclave 2**, **node10** for **Enclave 3**, etc. in the above example.
- The link emulation nodes are (*enclave node offset* + (N+2) and (N+3)), **node3** and **node4** for **Enclave 1**, **node7** and **node8** for **Enclave 2**, **node11** and **nod e12** for **Enclave 3**, etc. in the above example.

Our experiment topology assigns generic link identifiers in the following way for each enclave:

- The links between the application nodes and the IRON nodes are (*enclave link offset* + 1 through N), where N is the number of application nodes in the enclave. This is **link1** for **Enclave 1**, **link6** for **Enclave 2**, **link11** for **Enclave**, etc. in the above example.
- The link between the IRON node and the first link emulation node is (*enclave link offset* + (N+1)), **link2** for **Enclave1**, **link7** for **Enclave 2**, **link12** for **Encla ve 3**, etc. in the above example.
- The link between the first link emulation node and the black cloud is (*enclave link offset* + (N+2)), **link3** for **Enclave 1**, **link7** for **Enclave 2**, **link12** for **Encla ve 3**, etc. in the above example.
- The link between the IRON node and the second link emulation node is (*encl ave link offset* +(N+3)), **link4** for **Enclave 1**, **link9** for **Enclave 2**, **link14** for **E nclave 3**, etc. in the above example.
- The link between the second link emulation node and the black cloud is (*encl ave link offset* +(N+4)), **link5** for **Enclave 1**, **link10** for **Enclave 2**, **link15** for **Enclave 3**, etc. in the above example.

By adhering to the assignment of node and link identifiers described above for the experiment topology, the automated experiment execution scripts can figure out the generic node and link identifiers for any given enclave id.

The next section describes the testbed topology file, which binds the above

abstract version of the testbed with a real physical instantiation.

4.2. Testbed Topology Files

To run on a real testbed, we need to bind the abstracted experiment topology illustrated above with an instantiated physical topology of a real testbed. The goal is to have a common set of configuration files that can be bound to the desired physical nodes at runtime. The Testbed Topology File is the glue that binds the abstracted testbed description to a set of real physical nodes. Each Testbed Topology File may contain the following lines:

- '#' lines: This is a comment line.
- suffix <host name suffix>: This is the suffix to use when creating the fully qualified host names. This line can include the EXP_NAME keyword which will be substituted with the provided experiment name at configuration time. This feature is only used for ISI DeterLab experiments because the assigned hostnames for the DeterLab testbed nodes contains the name of the experiment in the fully qualified name.
- exp_base_dir: This is the base directory for the experiments on the remote testbed nodes.
- results_location: This is the directory that the experiment results will be written to on the local machine (the machine from which the experiment was executed).
- remote_execution_node: This is the node that the experiment will be executed from. This is an optional entry and is typically only set for DeterLab experiments, where the value is usually set to users.isi.deterlab.net.
- num_enclaves: The total number of enclaves in the testbed.
- app_nodes_per_enclave: The number of application nodes in each enclave.
- le_nodes_per_enclave: The number of Link Emulator nodes in each enclave. This must be 1 or 2.
- linkX nodeA nodeB: This describes a link between 2 nodes. The current scripts do not use this information, however, it is anticipated that this may be useful for automatically creating the DeterLab configuration files as a future task.
- nodeA hostname linkX=a.b.c.d,linkY=e.f.g.h: This describes the information for generic nodeA, including its hostname and the IP Addresses of its links.

The Testbed Topology Files reside in the **<IRON install dir>/IRON/experiments/testbeds/** directory.

Following is the testbed topology file for the Example Testbed previously described:

Example Testbed Topology File

```
# Topology file for BBN testbed
#
#  Enclave 1:                                    Enclave 4:
#
#      link1    link2      link3                 link18    link17
link16
# node1 --- node2 --- node3 ---+     +---+     +--- node15 --- node14 ---
node13
#             |                |     |   |     |                |
#             |               rtr1 --|   |-- rtr4               |
#             |                |     |   |     |                |
#             +------ node4 ---+     |   |     +--- node16 ------+
#              link4       link5     |   |    link20         link19
#                                    |   |
#                                    |   |
#  Enclave 2:                        |   |     Enclave 5:
#                                    |   |
#      link6     link7      link8    |   |     link23    link22
link21
# node5 --- node6 --- node7 ---+     | S |     +--- node19 --- node18 ---
node17
#             |                |     | W |     |                |
#             |               rtr2 --| I |-- rtr5               |
#             |                |     | T |     |                |
#             +------ node8 ---+     | C |     +--- node20 ------+
#              link9       link10    | H |    link25         link24
#                                    |   |
#                                    |   |
#   Enclave 3:                       |   |     Enclave 6:
#                                    |   |
#      link11    link12     link13   |   |     link28     link27
link26
# node9 --- node10 -- node11 ---+    |   |     +--- node23 --- node22 ---
node21
#             |                |    |   |     |                |
#             |               rtr3 --|   |-- rtr6               |
#             |                |    |   |     |                |
#             +------ node12 ---+    +---+     +--- node24 ------+
#              link14       link15            link30        link29


suffix bbn.com
exp_base_dir /home/${USER_NAME}
results_location ${HOME}/iron_results

num_enclaves 6
app_nodes_per_enclave 1
le_nodes_per_enclave 2

link1 node1 node2
link2 node2 node3
link3 node3 rtr1
link4 node2 node4
```

This document is subject to the restrictions on the cover page.

```
link5 node4 rtr1
link6 node5 node6
link7 node6 node7
link8 node7 rtr2
link9 node6 node8
link10 node8 rtr2
link11 node9 node10
link12 node10 node11
link13 node11 rtr3
link14 node10 node12
link15 node12 rtr3
link16 node13 node14
link17 node14 node15
link18 node15 rtr4
link19 node14 node16
link20 node16 rtr4
link21 node17 node18
link22 node18 node19
link23 node19 rtr5
link24 node18 node20
link25 node20 rtr5
link26 node21 node22
link27 node22 node23
link28 node23 rtr6
link29 node22 node24
link30 node24 rtr6

# Enclave 1
node1 gnat-app1 link1=10.1.3.2
node2 gnat1 link1=10.1.3.1,link2=10.1.1.2,link4=10.1.2.2
node3 gnat-le1 link2=10.1.1.102,link3=10.1.1.101
node4 gnat-le1 link4=10.1.2.102,link5=10.1.2.101

# Enclave 2
node5 gnat-app2 link6=10.2.3.2
node6 gnat2 link6=10.2.3.1,link7=10.2.1.2,link9=10.2.2.2
node7 gnat-le2 link7=10.2.1.102,link8=10.2.1.101
node8 gnat-le2 link9=10.2.2.102,link10=10.2.2.101

# Enclave 3
node9 gnat-app3 link11=10.3.3.2
node10 gnat3 link11=10.3.3.1,link12=10.3.1.2,link14=10.3.2.2
node11 gnat-le3 link12=10.3.1.102,link13=10.3.1.101
node12 gnat-le3 link14=10.3.2.102,link15=10.3.2.101

# Enclave 4
node13 gnat-app4 link16=10.4.3.2
node14 gnat4 link16=10.4.3.1,link17=10.4.1.2,link19=10.4.2.2
node15 gnat-le4 link17=10.4.1.102,link18=10.4.1.101
node16 gnat-le4 link19=10.4.2.102,link20=10.4.2.101

# Enclave 5
node17 gnat-app5 link21=10.5.3.2
```

This document is subject to the restrictions on the cover page.

```
node18 gnat5 link21=10.5.3.1,link22=10.5.1.2,link24=10.5.2.2
node19 gnat-le5 link22=10.5.1.102,link23=10.5.1.101
node20 gnat-le5 link24=10.5.2.102,link25=10.5.2.101

# Enclave 6
node21 gnat-app6 link26=10.6.3.2
```

This document is subject to the restrictions on the cover page.

```
node22 gnat6 link26=10.6.3.1,link27=10.6.1.2,link29=10.6.2.2
node23 gnat-le6 link27=10.6.1.102,link28=10.6.1.101
node24 gnat-le6 link29=10.6.2.102,link30=10.6.2.101
```

Please refer to the testbed files in the IRON distribution, located in the **<IRON install dir>/IRON/experiments/testbeds/** directory for more examples, including the testbed files for running on the ISI DeterLab testbed.

5. Experiment Configuration Templates

While configuring experiments in terms of the nodes and links in the testbed topology file is possible and ultimately required by the automated experiment execution scripts, another layer of abstraction is provided that simplifies the configuration of an experiment. Configuration file templates are supported by the experiment execution scripts and provide the experimenter with a mechanism to describe configuration items in "notional" enclave terminology. This enables an experiment to be described to run on any combination of real physical enclaves without having to identify the real physical enclaves that will be used. As an example, consider a 3 Enclave experiment. The configuration templates enable the experimenter to describe the experiment in terms of Enclaves 1, 2, and 3, even if the experiment will run on a different set of physical Enclaves (e.g., Enclaves 6, 8, and 10). At runtime, the experiment execution scripts map "notional" enclaves to physical enclaves. The physical enclaves are then used to generate the node specific configuration files that are required to execute the experiment on the set of assigned real testbed nodes.

The experiment execution scripts accomplish this by substituting "notional" enclave replacement strings, strings that are enclosed in '$' characters in the experiment configuration templates, with the information that is extracted from the testbed topology file. These replacement strings have the following format: *$replacement_string$*. The following set of configuration template replacement strings are inserted into the configuration templates when creating an experiment, fully described in the *Creating An Experiment* section below, and are generally not modified by an experimenter:

- ***$enclaveX_appY_wan_addr$***: Refers to the WAN-facing address of application Y in Enclave X.
- ***$enclaveX_iron_node$***: Refers to the IRON node in Enclave X.
- ***$enclaveX_iron_lan_addr$***: Refers to the LAN-facing address of the IRON node in Enclave X.
- ***$enclaveX_iron_lan_link$***: Refers to the LAN-facing generic link identifier of the IRON node in Enclave X.

- ***$enclaveX_iron_wanY_link$***: Refers to the WAN-facing generic link identifiers of the IRON node in Enclave X. Recall that we typically create experiment testbeds that are dual homed. This directive enables us to refer to the appropriate WAN-facing generic link identifiers.
- ***$enclaveX_iron_wanY_addr$***: Refers to one of the WAN-facing addresses of the IRON node in Enclave X. Recall that we typically create experiment testbeds that are dual homed. This directive enables us to refer to the appropriate WAN-facing address.

While the above configuration template replacement strings are not generally modified by the experimenter when configuring an experiment, they can be observed in the configuration template files that are generated when the experiment is created. There are, however, a set of configuration template replacement strings that are typically provided by the experimenter when finalizing the experiment details. These replacement strings are are as follows:

- ***$enclaveX_appY_node$***: Refers to application host Y in Enclave X.
- ***$enclaveX_leY_node$***: Refers to link emulation node Y in Enclave X.

If it is necessary to collect packet traces to further examine the results of the experiment, described in the *Common Debugging Techniques* section, the following configuration template replacement strings may need to be provided:

- ***$enclaveX_appY_wan_link$***: Refers to the WAN-facing generic link identifier of application Y in Enclave X.
- ***$enclaveX_leY_lan_link$***: Refers to LAN-facing generic link identifier of link emulation node Y in Enclave X.
- ***$enclaveX_leY_wan_link$***: Refers to WAN-facing generic link identifier of link emulation node Y in Enclave X.

6. Creating A New Experiment

So far, we have described: physical experiment testbeds, the abstraction of these testbeds to generic node and link identifiers contained in the testbed topology file, and a further abstraction, configuration file templates, which enables the experimenter to configure experiments in "notional" enclave terminology. A number of system experiments exist (in the **<IRON install dir>/IRON/experiments/** director y) and are documented here. Next, we will outline the process of creating a new experiment from scratch.

6.1. Define Experiment

Defining a set of objectives is the first step in creating a new experiment. At a high

level, these objectives include the following:

- Characterization of "what" the initial IRON network provides: This includes the "plumbing" required to enable the enclaves to communicate with each other and the initial link configuration between the enclaves.
- Characterization of "how" the IRON network will be utilized: This includes a specification of the flows, the desired utility functions for the flows, and the set of network impairments.

For the purposes of this tutorial, we will describe how to create the 3-node-system experiment. The high-level details of the experiment are summarized in the Figure 6 below:

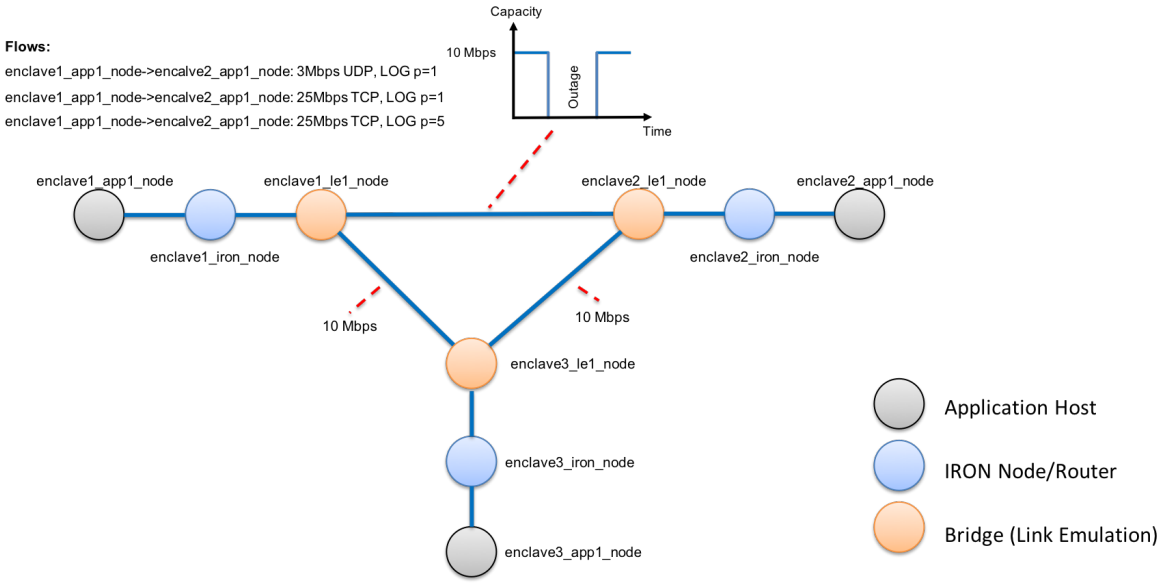

Figure 6: Summary details for 3-node-system experiment

Note that the nodes in the figure above are labelled in "notional" enclave terminology. Recall, for example, that ***enclave1_app1_node*** refers to application node 1 in Enclave 1. As previously described, thinking about the experiment like this abstracts the experimenter away from the real physical hardware that the experiment will run on and enables the experiment to run on any collection of real, physical enclaves that are available in the target testbed. The above figure contains all of the information that is required to continue with the process of creating our new experiment. It contains the details describing the nature of the desired IRON network, the "what", and the details of how that network is to be utilized (in terms of the desired flow dynamics and network impairment events), the "how". We can now proceed with gathering up the inputs and generating the experiment configuration files.

6.2. Generate Experiment Configuration Files

The *create_exp.sh* script, located in the **<IRON install dir>/IRON/experiments/scripts/** directory, generates the configuration files and templates for our experiment. A full description of the script and all of its options and arguments is obtained by running the following command:

create_exp.sh -h

A more detailed description of the most commonly included options and arguments is provided below.

- The base directory option, -B, identifies the location for the generated experiment configuration files and templates.
- The duration option, -l, specifies the duration of the experiment, in seconds. The default value is 60.
- The delay option, -d, identifies the initial delay value, in ms, for the links between the enclaves. The default value is 10.
- The throttle option, -t, identifies the initial rate, in Kbps, for the links between the enclaves. The default value is 100000.
- The size option, -b, identifies the buffer size, in bytes, for the links between the enclaves. The default value is 500000.
- The arguments to the script include the number of experiment enclaves and a name for the experiment.

Currently, the *create_exp.sh* script only supports the creation of a full mesh topology of the enclaves. A future enhancement might include providing a specification configuration file that express a more general connectivity matrix of the enclaves. If a full mesh is not what is desired for the experiment, this can be adjusted following the creation of the experiment. Additionally, the *create_exp.sh* sc ript creates an initial set of links between the enclaves that are homogeneous. If a homogeneous set of links is not what is desired for the experiment, this can be adjusted following the creation of the experiment.

6.2.1. Gather Inputs

The next step is to gather the inputs for the experiment creation script. The following list of configuration inputs is sufficient to create our experiment:

- Number of Enclaves: 3
- Experiment Name: 3enclave_example_exp
- Link Characteristics: 10 Mbps, 10 ms delay, 50 Kbyte buffers, Packet Error Rate Error Model, No Jitter Model
- Base Directory: <IRON installation dir>/IRON/experiments
- Flow Definitions: 2 TCP flows and 1 UDP flow from Enclave 1 to Enclave 2
- Service Definitions: 1 high priority and 1 low priority TCP flow, and 1 low

priority UDP flow

- Network Impairment: Direct link between Enclaves 1 and 2 severed during run for 10 seconds

6.2.2. Script Execution

Now that we have defined the objectives for our experiment and collected the inputs for the experiment creation script, we are ready to generate the experiment's configuration files and templates. For our example, we will provide the destination directory for the generated experiment configurations, set the link rates to 10 Mbps, 10 ms delay, 50 Kbyte buffers, and provide the required number of enclaves and experiment name to the *create_exp.sh* script. The following commands will generate our experiment:

cd <IRON installation dir>/IRON/experiments/scripts/

./create_exp.sh -B <IRON installation dir>/IRON/experiments -t 10000 -d 10 -b 50000 3 3enclave_example_exp

6.2.3. Generated Outputs

The experiment configuration files are placed in the directory provided to the experiment creation script, **<IRON installation dir>/IRON/experiments/3enclave_example_exp** in our example. The generated outputs are shown below:

Generated Experiment Configuration Files
<IRON installation dir>/IRON/experiments/3enclave_example_exp
cfgs
amp_common.cfg
amp_services.tmpl
bin_map.tmpl
bpf_common.cfg
bpf_enclave1.tmpl
bpf_enclave2.tmpl
bpf_enclave3.tmpl
lem_init.tmpl
lem.tmpl
process.cfg
system.cfg
tcp_proxy_common.cfg
tcp_proxy_enclave1.tmpl
tcp_proxy_enclave2.tmpl
tcp_proxy_enclave3.tmpl
traffic.tmpl
udp_proxy_common.cfg
udp_proxy_enclave1.tmpl
udp_proxy_enclave2.tmpl
udp_proxy_enclave3.tmpl
exp.tmpl

At this point, the generated set of configuration files and templates contain the directives to construct an IRON network enabling our 3 enclaves to communicate with each other. The *.cfg files contain configuration information that is not node specific. The *.tmpl files are the configuration template files containing enclave specific information in the form of configuration template replacement strings, described in the *Experiment Configuration Templates* section above. The *exp.tmpl* f

ile contains the experiment's high-level configuration information including: length of the experiment, application, IRON, and link emulation nodes in the experiment, and any desired packet capture nodes in the experiment.

Note: The bolded configuration templates (*amp_services.tmpl*, *lem.tmpl*, and *traffic.tmpl*) **must** be modified after they have been created.

Recall that the experiment creation script provides the "what", of an IRON network over which the enclaves can communicate. The experimenter needs to configure "how" the IRON network is to be utilized, which include the network services that will run, the details of how the (emulated) links will operate as well as the traffic profiles. . Modifying these files is the means of accomplishing this. The remaining configuration template files typically do not need to be modified unless the experimenter wishes to deviate from the default IRON network experiment configuration. The configuration files and templates contain comments that explain each of the configurable items, should the experimenter find it necessary to change any of them.

6.3. Optionally Modifying the IRON Network "Plumbing"

Before proceeding with configuring "how" the generated IRON network is to be used, we must ensure that the generated network aligns with the experiment objectives. Recall that the *create_exp.sh* script that was used to create our new experiment creates a full mesh of enclaves with homogeneous links between each enclave. If either of these script creation assumptions is not desired for the created experiment, the experiment configuration templates can be modified before proceeding with finalizing our experiment.

6.3.1. Modifying Generated Full Mesh

The configuration of the connectivity between the experiment enclaves is contained in the *bpf_enclaveX.tmpl* and *lem_init.tmpl* configuration template files. The *PathController.X.Type* and *PatheController.X.Endpoints* lines in the *bpf_enclaveX.tmpl* file can be removed for any links in the created configuration template that are not required for the experiment. If any lines are removed, the *Bpf.NumPathControllers* line **must** also be modified to reflect this change. Note that the *PathController.X.Type* and *PathController.X.Endpoints* entries **must** contain X values between 0 and *Bpf.NumPathControllers*.

Additionally, any Paths in the initial link emulation configuration file template, *lem_init.tmpl*, can be removed, if desired. This will eliminate the step of setting up link emulation paths that will not be used.

Note that the modifications described in this section are optional and need only be

done if the desired experiment connectivity is not a full mesh of the experiment enclaves.

6.3.2. Modifying Initial Link Characteristics

The initial link configuration information is located in the *lem_init.tmpl* configuration template file. The configuration information for any link that should deviate from generated configuration template can be made in this file. See the link and path emulation utility, documented here, for the details explaining how to configure the links to match the desired starting link characteristics. Note that this step in the process is optional and need only be done if the starting states of the links are not homogeneous.

6.4. Finalize Experiment Details

Thus far, we have a generated a set of configuration files that enable our enclaves to communicate with each other. We now need to configure "how" we will use this network of enclaves to achieve the experiment objectives. This includes configuring the experiment flows, utility function definitions, and network impairments.

6.4.1. Flow Definitions

The flow configuration information is entered in the *traffic.tmpl* configuration template. Three types of flows can be configured in this file, tcp, udp, or short_tcp. The format of the directive for TCP and UDP flows is as follows:

> **protocol src dest num_flows src_port dst_port start_time end_time packet_size data_rate**

The format of the directive for short TCP flows is as follows:

> **protocol src dest num_flows lo_port hi_port start_time end_time flow_size_bytes interflow_duration**

Refer to the comment section in the *traffic.tmpl* file for a complete description of the flow definition directives. Our example experiment has 3 flows, 1 UDP flow and 2 TCP flows. The next step is to configure the 3 flows for our example experiment as follows:

```
udp $enclave1_app1_node$ $enclave2_app1_node$ 1 30777 30777 10.0 50.0
1024 3Mbps
tcp $enclave1_app1_node$ $enclave2_app1_node$ 1 29778 29778 10.0 50.0
1024 25Mbps
tcp $enclave1_app1_node$ $enclave2_app1_node$ 1 29779 29779 10.0 50.0
1024 25Mbps
```

Note that the source and destination of each of the flows are specified in configuration template replacement string notation described previously in the *Experiment Configuration Template* section above. Since our enclaves have only 1 application node and our flows are from Enclave 1 to Enclave 2, the above replacement strings define our flows and provide the automated experiment execution scripts with the required information. The automated experiment execution scripts will replace instances of the configuration template replacement strings, i.e., **$enclave1_app1_node$**, with the appropriate generic node id extracted from the testbed topology file.

6.4.2. Flow Service Definitions

Now that we have defined a set of flows for the experiment, we need to define the Service Definitions for the flows. The format for defining the Service Definitions is fully described in the comment section of the *amp_services.tmpl* configuration file.

Our experiment objectives call for 1 high priority Log utility TCP flow, 1 low priority Log utility TCP flow, and 1 low priority Log utility UDP flow. The following entries accomplish this:

```
2 tcp_proxy add_service
29778-29778;type=LOG:a=10:m=25000000:p=1:label=low_prio_tcp;
2 tcp_proxy add_service
29779-29779;type=LOG:a=10:m=25000000:p=5:label=high_prio_tcp;
2 udp_proxy add_service
30750-30799;1/1;1500;0;0;120;0;type=LOG:a=10:m=25000000:p=1:label=low_pri
o_udp
```

Note that the port ranges of our utility function definitions **must** match the ports in our flow definitions. It is also important to note that the flow service definition lines should be added to the *amp_services.tmpl* file in the section labeled "User defined flow and service definitions should be placed here:". The default service definitions, appearing at the end of the *amp_services.tmpl* file, **must not** be deleted and must appear as the last entries in the file.

6.4.3. Network Impairment Definitions

The link emulation nodes in our experiment infrastructure handle experimenter defined impairments. The details of the link and path emulation utility is fully documented here. Our experiment calls for a network impairment 20 seconds into the run, lasting for 10 seconds, that severs then restores the direct link between Enclaves 1 and 2. The network impairment information is entered in the *lem.tmpl* co nfiguration file. The format of the entries is fully described in the comment section of the file. The following entries will accomplish the desired network impairment event:

```
sleep 20
$enclave1_le1_node$ set path:2 per:1.0
$enclave2_le1_node$ set path:1 per:1.0
sleep 10
$enclave1_le1_node$ set path:2 per:0.0
$enclave2_le1_node$ set path:1 per:0.0
```

The experiment creation script creates an IRON network that is configured to support egress modeling, meaning that network impairment modeling is employed as packets are leaving an enclave. Two link emulation nodes need to be modified in order to completely sever the link between Enclaves 1 and 2. Line 2 modifies the Enclave 1 link emulation node, setting the Packet Error Rate on Path 2, the path to Enclave 2, to 1.0 (a 100% packet error rate). Similarly, line 3 modifies the Enclave 2 link emulation node, setting the Packet Error Rate on Path 1, the path to Enclave 1, to 1.0. The end result of these link emulation directives drops all packets leaving Enclave 1 destined Enclave 2 on the direct path, and vice versa. During the outage, the only remaining path from Enclave 1 to Enclave 2 is the 2-hop path via Enclave 3. Ten seconds later, the action is undone by resetting the Packet Error Rates back to 0.0, restoring the direct path between Enclaves 1 and 2. Again, note that the entries in this file are specified in configuration template replacement string notation.

6.4.4. Optional Additional Configuration Modifications

At this point in the creation of our new experiment, we have the script generated set of configuration files and templates and the experimenter-modified templates describing how the IRON network is to be used. There are many configurable items in the generated configuration files and templates, any of which can be modified by the experimenter prior to experiment execution. If desired, this is the time to make modifications to the IRON component configurable items.

The experiment execution scripts also support a feature, referred to as parameterized experiments, which allows the experimenter to run a series of experiments, each having its own unique configuration. Note that this is optional

and need not apply to all created experiments.

6.4.4.1. Parameterized Experiments

The experiment execution scripts permit an experimenter to run a series of experiments in which one or more configurable items are changed from run to run while maintainting the same infrastructure. This is accomplished by inserting special tags into a configuration file or template and providing a parameter input file, named *params.txt*, which contains the values for the various tags. If a *params.txt* file is created for the experiment, it **must** be placed in the experiment's *cfgs* directory. As an illustration, consider a series of experiments where the experimenter wants to change the number of high priority TCP flows to observe its affect on system behavior. To accomplish this, the experimenter modifies the experiment's *traffic.tmpl* configuration template as follows:

Parameterized traffic.tmpl

```
udp $enclave1_app1_node$ $enclave2_app1_node$ 1 30777 30777 10.0 50.0
1024 3Mbps
tcp $enclave1_app1_node$ $enclave2_app1_node$ %NUM_HI_PRIO_TCP_FLOWS%
29778 29778 10.0 50.0 1024 25Mbps
tcp $enclave1_app1_node$ $enclave2_app1_node$ 1 29779 29779 10.0 50.0
1024 25Mbps
```

The line of interest from the above snippet from the *traffic.tmpl* configuration template file is line 2. Notice that the value for the number of flows on this line (previously "1") has been replaced by **%NUM_HI_PRIO_TCP_FLOWS%**, a parameter substitution tag. The names of parameter substitution tags are chosen by the experimenter and adhere to the following format: %tag_name%.

The automated experiment scripts will replace instances of the parameter substitution tags with the values extracted from the parameter input file, *params.txt*, which is illustrated below for our example experiment:

params.txt

```
# Parameter values for the number of high priority TCP flows.
NUM_HI_PRIO_TCP_FLOWS = 1 5 10
```

This *params.txt* input file specifies 3 values (1, 5, and 10) for the parameter substitution tag, **NUM_HI_PRIO_TCP_FLOWS**. Notice that the tag delimiters, '%',

are not included in the parameters input file. Each line in the file contains the tag name and the set of values that the tag can be assigned.
Note: All tags defined in this file **must** have the same number of values.
When a parameterized experiment is configured, multiple run directories are created, one for each set of configuration options defined in the *params.txt* input file. This will be described below in the *Experiment Execution* section.

7. Preparing Experimenter's Run-Time Environment

We have now successfully created our new experiment and are almost ready to execute it. However, we need to ensure that our environment is prepared before we can run our new experiment.

7.1. Shell (login environment) Set-up

Prior to running an IRON experiment, refer to the **<IRON install dir>/IRON/iron/README.txt** file for detailed information about how to set up your shell to build the IRON software and run the automated experiment execution scripts. The contents of this file are provided below as a convenience.

IRON README.txt file

```
Quick-Start Guide
=================================================================================

1.  Make sure you are using bash, tcsh or csh.

2.  Set the "IRON_HOME" environment variable to where this file is
located

        export IRON_HOME="<IRON install dir>/IRON/iron"

    or

        setenv IRON_HOME <IRON install dir>/IRON/iron

    where <IRON install dir> is the location of the IRON installation.
This
    may be placed in your .bashrc or .cshrc file for convenience.
However,
    make sure that this file tests if this environment variable has
already
    been set before setting it. A bash example of this is:

        if [ -z "$IRON_HOME" ] ; then
```

```
          export IRON_HOME=<IRON install dir>/IRON/iron
        fi

    A tcsh/csh example of this is:

        if (! $?IRON_HOME ) then
          setenv IRON_HOME    <IRON install dir>/IRON/iron
        endif

3.  Source the appropriate setup file, e.g., for bash:

        cd $IRON_HOME
        . setup/debug.bash

    or for tcsh/csh:

        cd $IRON_HOME
        source setup/debug.csh

    Substitute the correct path and setup file name (currently
debug.bash,
    optimized.bash, debug.csh, or optimized.csh) depending on your shell
and
    how you would like the software built.

4.  If the UNIX platform you are using is something different than Linux
with
    a 3.13 or 3.2 kernel, then create the needed debug and optimized
style
    files for your platform in the iron/build directory using the
    Linux_3.2_debug and Linux_3.2_optimized style files as templates.
The
    platform name may be determined using the command "uname -s", and the
    version number (of the form X.Y) may be determined using the command
    "uname -r" and ignoring everything after the first and second
    numbers. Replace instances of -DLINUX_3_2 with -DLINUX_X_Y.

5.  To build all native code (C/C++) software, perform the following:

        cd $IRON_HOME
        make clean
        make

6.  To build the unit test code, perform the following:

        cd $IRON_HOME
        make -f makefile.unittest clean
        make -f makefile.unittest

7.  To execute the unit test code, perform the following:

        cd $IRON_HOME
        ./bin/{style-file-name}/testironamp
```

```
./bin/{style-file-name}/testironbpf
sudo ./bin/{style-file-name}/testironcommon
./bin/{style-file-name}/testirontcpproxy
./bin/{style-file-name}/testironudpproxy
```

where {style-file-name} is the name of the file described in step 4 - for example Linux_3.2_debug.

The unit tests print out dots while they are executing and a status message.

8. To build the IRON documentation, perform the following:

```
cd $IRON_HOME/doc
make docs
```

9. To access the IRON documentation, use a web browser to open the file $IRON_HOME/doc/html/index.html

```
10. To execute the IRON python unit test code, cd $IRON_HOME/python
    and then follow the directions in the README.txt file.
```

7.2. Experimenter ssh Configuration

IRON experiments are typically run as a shared user so that not every experimenter requires special sudo privileges. We typically create a shared user, **ir on** for the purposes of this tutorial, to accomplish this. After creating the **iron** user public/private key pair (named *id_rsa_iron* in our example), the private-key file is distributed to all users that will run IRON experiments. The experimenter typically puts the private key in their **~/.ssh** directory. The public-key file is added to the /home/iron/.ssh/authorized_keys file on each testbed machine. Each experimenter can then modify their ssh configuration so that when an ssh command is made to a testbed machine, the command is executed as the **iron** user with the *id_rsa_iron* ke y for non-password authentication.

The following lines can be added to your ssh config file, **~/.ssh/config**, to accomplish this for our example experiment testbed:

```
Host gnat-app?
    HostName %h.bbn.com
    Port 22
    User iron
    IdentityFile ~/.ssh/id_rsa_iron
    StrictHostKeyChecking=no
    LogLevel=quiet
Host gnat-app?.bbn.com
    Port 22
    User iron
    IdentityFile ~/.ssh/id_rsa_iron
    StrictHostKeyChecking=no
    LogLevel=quiet

Host gnat?
    HostName %h.bbn.com
    Port 22
    User iron
    IdentityFile ~/.ssh/id_rsa_iron
    StrictHostKeyChecking=no
    LogLevel=quiet
Host gnat?.bbn.com
    Port 22
    User iron
    IdentityFile ~/.ssh/id_rsa_iron
    StrictHostKeyChecking=no
    LogLevel=quiet

Host gnat-le?
    HostName %h.bbn.com
    Port 22
    User iron
    IdentityFile ~/.ssh/id_rsa_iron
    StrictHostKeyChecking=no
    LogLevel=quiet

Host gnat-le?.bbn.com
    Port 22
    User iron
    IdentityFile ~/.ssh/id_rsa_iron
    StrictHostKeyChecking=no
 LogLevel=quiet
```

The IdentityFile entries in the above configuration **must** point to the **iron** user private key that was created. The above configuration ensures that all ssh commands to our testbed machines will occur as the **iron** user. This is the case when either the hostname, i.e., **gnat1**, or the fully qualified domain name, i.e., **gnat 1.bbn.com**, is used. Note that there is nothing special about the **iron** user. The shared user account can have any name, however, this name **must** be provided to

the automated experiment execution scripts as described in the *Running An Experiment* section below.

8. Reserving Testbed Nodes

It is generally true that only one instance of the various IRON components can be run on a host at a time. An exception to this rule is the link emulation component. For dual homed experiments, we typically run 2 instances of the link emulation executable on a single host. We therefore need an automated mechanism to reserve testbed nodes for our experiments so that we don't conflict with experiments already in progress. The IRON testbed reservation system accomplishes this.

The automated experiment execution script, *reserve_ctl.sh*, interacts with the testbed reservation script that runs on the testbed reservation server to manage the reservation of the physical nodes in the various experiment testbeds. The testbed reservation system uses standard Linux file locking mechanisms (*lockfile-create* an d *lockfile-remove* system calls) to reserve testbed nodes as an atomic operation, works with any testbed topology file (even new ones), and does not require that anything be pre-installed on the reservation server.

The following high-level steps occur when interacting with the testbed reservation system:

1. Create lockfile for the testbed
2. Modify testbed reservation state
3. Remove lockfile for testbed

The testbed lockfiles are located in the following directory:

/run/lock/iron_testbeds/

The testbed reservation system must support many experiments being run by many experimenters. To satisfy this objective, it runs as a shared user (typically the *iron* u ser at BBN) on the reservation server so that a global view of the status of all testbeds is available. This common user is typically the user that the experiments run as, described in the *Experiment Execution* Section below. The testbed reservation state for a testbed is kept in the following location on the reservation server:

/home/<shared user>/testbed_reservations/<testbed_name>/

In this directory are enclave lock files that include who locked the testbed enclave when. Note that the files in this directory are only text files and modifying them without grabbing a lockfile is both possible and dangerous as the integrity of the testbed may be compromised. To ensure proper operation, the *reserve_ctl.sh* script

should be used as it ensures that the policy of modifying the enclave lock files only occurs when the testbed lock file is acquired.

When interacting with the testbed reservation system for a testbed, the reservation script takes the following detailed actions:

1. The real testbed lock file, located in the **/run/lock/iron_testbeds/** directory, is created with the *lockfile-create* system command. If the lockfile already exists, the script waits until either the lockfile can be created or times out.
2. Once the testbed lock file is created, the **/home/<user>/testbed_reservations/<testbed_name>/** directory is "owned" by the script and can be modified.
3. Enclave lock files for reserved enclaves are created in the **/home/<user>/testbed_reservations/<testbed_name>/** directory. These files identify the owner of the enclave and when it was locked.
4. The real testbed lock file, located in the **/run/lock/iron_testbeds/** directory, is removed. After this happens, testbed enclaves can be reserved or released by other experimenters.

The ***reserve_ctl.sh*** **script is fully integrated with the automated experiment execution scripts and typically is not run stand-alone**, however, it can be, if necessary. A full description of the script and all of its options and arguments is obtained by running the following command:

reserve_ctl.sh -h

A more detailed description of the most commonly included options and arguments is provided below.

- The base directory option, -b, identifies the testbed topology file base directory. This is typically the **<IRON install dir>/IRON/experiments/testbeds/** directory.
- The -l option identifies the desired physical enclaves requested, as a colon separated list. For example, **-l 1:2:3**, is used to request locking physical Enclaves 1, 2, and 3 (discussed in the *Testbed Topology Files* section). Note that this action should not be done in a stand-alone mode as it will occur automatically when an experiment is run.
- The -L option identifies the desired number of physical enclaves requested. This option is used when it doesn't matter which real physical enclaves are used for the experiment. For example, **-L 3**, is used to request 3 physical enclaves. Note that this action should not be done in a stand-alone mode as it will occur automatically when an experiment is run.
- The query option, -q, queries the status of the testbed.
- The release option, -r, releases the colon separated list of enclaves. For example, **-r 1:2:3**, is used to release physical Envclaves 1, 2, and 3. This can

be used by the experimenter if the automated experiment execution scripts fail for any reason to release the enclaves that were locked for the failed experiment.

- The server option, -s, identifies the reservation server host.
- The user option, -u, identifies the user the script is to run as. The state of the testbed reservation locks will be found in the **/home/<user>/testbed_reservations/** directory on the reservation server. All experimenters should use a shared user here.
- The testbed topology file for the desired testbed is provided as an argument to the script.

The following command is used to query the status of our example testbed:

reserve_ctl.sh -b <IRON install dir>/IRON/experiments/testbeds -q bbn_testbed.cfg

The following is observed when all enclaves are available:

```
All enclaves available.
```

The following is observed when some of the testbed enclaves are reserved:

```
enclave1:
Locked by: sgriffin
Reserved on: Tue Oct 30 10:18:11 EDT 2018

enclave2:
Locked by: sgriffin
Reserved on: Tue Oct 30 10:18:11 EDT 2018

enclave3:
Locked by: sgriffin
Reserved on: Tue Oct 30 10:18:11 EDT 2018
```

For all enclaves that are reserved and currently unavailable, the output above shows who "owns" the enclave, user sgriffin, and when it was locked. This information may be useful if enclaves get "stuck" in a reserved state. It provides a reference to who has the locked enclaves so that additional information including intended duration of usage can be queried.

Note that the operation of the testbed reservation system is fully integrated into the experiment execution process, described in the "Experiment Execution" section below. The *reserve_ctl.sh* script is generally only run manually to query the state of the testbed reservations when not enough testbed nodes are currently available. The *reserve_ctl.sh* script can also be manually run to release testbed nodes that

are currently reserved. Note that the user that has the nodes currently locked must execute the script to release them.

9. Experiment Execution

9.1. Experiment Overview

Typically, an IRON experiment is executed on the machine where the IRON distribution is installed. This is usually **not** a testbed machine, but a machine that is able to reach all of the testbed machines. When an experiment runs, a number of the following steps may occur depending on which command-line options are provided to the automated experiment execution script:

1. The IRON executables are built on the local machine.
2. The experiment is staged on the local machine in the **/home/<user>/iron_exp_staging** directory, referred to as the "local staging directory".
3. The experiment is configured in the local staging directory.
4. The experiment is installed on the remote testbed machines (into the *exp_base_dir* defined in the Testbed Topology File)
5. The experiment is executed on the remote testbed machines.
6. The experiment results are post-processed on the remote testbed machines.
7. The experiment results are collected and placed on the local machine in the **/home/<user>/iron_results** directory.

The details of running an experiment with the automated experiment execution scripts and viewing experiment results are described in the following sections.

9.2. Running An Experiment

Once your environment is set up, the *run_exp.sh* script, located in the **<IRON install dir>/IRON/experiments/scripts** directory, is used to setup and run experiments. A full description of the script and all of its options and arguments is obtained by running the following command:

run_exp.sh -h

A more detailed description of the most commonly included run_exp.sh options and arguments is provided below.

- The make option, -m, builds the IRON executables. This is a local operation and does not interact with any testbed nodes.
- The stage option, -s, creates a local staging area for the experiments. Additionally, it creates a tarball that is used during the installation phase. If the

remote_execution_node is set in the Testbed Topology File, then the experiment tarball is copied to the remote node and untarred there. All remaining commands will run on this remote node.

- The configure option, -c, generates the experiment configuration files for a selected testbed topology. The configure command reads in the testbed topology file and the *exp.cfg* file for each experiment and expands the generic node names in the experiment configuration to fully qualified node names. It also interacts with the testbed nodes to dynamically determine the interface names to use during the experiment. The result of this command is a modified version of the *exp.cfg* file that is placed in the appropriate experiment directory in the staging area on the local machine, **${HOME}/iron_exp_staging/**. Additionally, this step of the process specializes any parameterized configuration files in the experiment cfgs directories and creates the appropriate number of experiment run directories for each experiment.
- The install option, -i, installs the IRON executables, test execution scripts, and the experiment-specific configuration files on the testbed nodes for all of the experiments to be run.
- The run option, -r, starts the experiments. Note that this results in possibly multiple runs for an experiment when the *params.txt* file is used. Additionally, multiple experiments may be provided to the script. The requirement for running multiple experiments is that each experiment that is to be run has to operate on the single testbed topology configuration. When each experiment terminates, the test components are stopped and the experiment results are collected and placed in experiment run specific directories.
- The process option, -p, generates plots of throughput, instantaneous queues, and instantaneous send rate in the appropriate **experiment/runX/nodeY/results/** directory for each IRON and application node.
- The node reservation option, -l or -L. The -l option is a colon separated list of the enclaves that are to be reserved, e.g., 1:2:3 indicates that the experimenter wishes to use Enclaves 1, 2, and 3. If any of the enclaves in the list are currently locked, the script terminates with an error code. The -L option indicates the number of enclaves that are required for the experiment. If the number of requested enclaves are not available for reservation, the script terminates with an error code. When the requested list or number of enclaves have been reserved, the file *enclaves.cfg* is created and placed in the **${HOME}/iron_exp_staging/** directory. This file indicates the real enclaves that are to be used for the experiment.
- The reservation host option, -o, identifies the testbed reservation system host. Note that if the experiment is running on the ISI DeterLab testbed, no testbed reservation process will occur.
- The user option , -u <user_name>, identifies the name of the user that the experiment is to be run as. Typically, this is set to *iron* for experiments run on the local testbeds and it is set to the experimenter's user name for ISI

DeterLab experiments. The user must have sudo privileges on the experiment nodes, preferably without prompting for passwords.

- The testbed configuration file option, -t <testbed_topo_file>, identifies the name of the testbed topology file, which contains the mapping of generic node names to real hostnames and associates IP Addresses with generic link names. These files are located in the **IRON/experiments/testbeds/** directory. Note that this topology does not need to match the topology of the experiment created for DeterLab, but the nodes/links in the DeterLab experiment must be a subset of the nodes/links in this topology file.
- The arguments to the script are the names of the experiments to run. As described earlier, more than one experiment may be run provided that all can be run on the same testbed topology.

The following command is used to build the executables, stage, configure, install, run, and process our example experiment where the experimenter desires that real physical Enclaves 1, 2, and 3 be used for the experiment:

run_exp.sh -mscirp -l 1:2:3 -u iron -t bbn_testbed.cfg 3enclave_example_exp

If the experimenter does not care which physical enclaves are used, the following command can be used:

run_exp.sh -mscirp -L 3 -u iron -t bbn_testbed.cfg 3enclave_example_exp

9.2.1. Generated Configuration Files

A number of experiment run directories may be created when an experiment is configured using the *run_exp.sh* script described above. The run1 directory is always created, but up to N run directories may be created. The number of directories is determined by the number of values that are specified for the parameterized substitution tags in the *params.txt* input file. Recall that our example experiment has 3 values for the **NUM_HI_PRIO_TCP_FLOWS** substitution tag. This results in the following directory structure being created for our example experiment:

Directory Structure Following Experiment Configuration

```
/home/<user>/iron_exp_staging/
 3enclave_example_exp
    cfgs
    logs
    pcaps
    run1
       cfgs
       logs
       pcaps
    run2
       cfgs
       logs
       pcaps
    run3
        cfgs
        logs
        pcaps
 bin
 scripts
 testbeds
```

The information in the **3enclave_example_exp/cfgs/** directory is only used to configure the experiment. When the example experiment is configured, a script is run to perform the parameter substitutions in the configuration files. The information in the **3enclave_example_exp/cfgs/** directory creates the **3enclave_example_exp/run1/**, **3enclave_example_exp/run2/**, and **3enclave_example_exp/run3/** directories. The configuration files in each of these directories have been tailored per the entries in the *params.txt* file. Each experiment run will use the information contained in the corresponding runX directory. Also note that the executables, the scripts, and the testbed file for the experiment are also staged. This provides a record of the experiment that is run.

9.2.2. Experiment Results

Following the execution of an experiment, the experiment artifacts will, by default, be collected and copied to the **results_location** as specified in the experiment's testbed topology file. The following shows the directory structure for run1 of our example experiment. There will be similar directories for run2 and run3 of our example experiment.

Experiment Result Directory Structure

```
/home/<user>/iron_results/<date_time>/
 3enclave_example_exp
    run1
        enclave1
           app1
              cfgs
              logs
              pcaps
              results
           iron
              cfgs
              logs
              pcaps
              results
           le1
               cfgs
               logs
               pcaps
        enclave2
           app1
              cfgs
              logs
              pcaps
              results
           iron
              cfgs
              logs
              pcaps
              results
           le1
               cfgs
               logs
               pcaps
        enclave3
            app1
               cfgs
               logs
               pcaps
               results
            iron
               cfgs
               logs
               pcaps
               results
            le1
                cfgs
                logs
                pcaps
 bin
```

We see that at the top-level of of the experiment's results directory is a **bin/** director y that contains a copy of all of the binaries that were used for the experiment as well as status information from the GIT repository from which the experiment was run, if available. The GIT information contains the latest commit information and includes any local changes in the experimenter's GIT sandbox included in the experiment. This information is sufficient to recreate the experiment at a later date, if desired.

The above directory structure is organized by "notional enclaves" for each of the runs. We see **enclave1/**, **enclave2/**, and **enclave3/** directories which contain the configuration information, log files, packet captures, and results for the application (**app1/**), IRON (**iron/**), and link emulation (**le1/**) nodes for the respective enclaves. For simplicity, only the **run1/** directory is shown above. There would be similar directories for ***run2/*** and **run3/** for our example experiment. If the experiment was dual homed, there would also be a **le2/** directory for each enclave directory. Similarly, if there were more than one application node per enclave, there would be additional **appX/** directories.

If, for any reason, the experimenter needs to visit the physical node for any of the components that ran during the experiment, the *node_to_enclave_map.txt* file, located in the staging directory, can be viewed. This file is used by the automated experiment execution scripts and contains information of the following form:

generic_node_name hostname results_directory_name

The following depicts the entries for our experiment when run on real physical Enclaves 1, 2, and 3:

```
node1 gnat-app1 enclave1/app1
node2 gnat1 enclave1/iron
node3 gnat-le1 enclave1/le1
node4 gnat-le1 enclave1/le2
node5 gnat-app2 enclave2/app1
node6 gnat2 enclave2/iron
node7 gnat-le2 enclave2/le1
node8 gnat-le2 enclave2/le2
node9 gnat-app3 enclave3/app1
node10 gnat3 enclave3/iron
node11 gnat-le3 enclave3/le1
node12 gnat-le3 enclave3/le2
```

So, if we need to track down an issue with the source of our flows, we know we should visit **gnat-app1**.

9.2.3. Viewing Results

If the **-p** flag is provided to the *run_exp.sh* script, a set of post-processing steps, outlined in the *process.cfg* configuration file, are executed. In our example experiment, the post-processing steps include the generation of goodput, latency, and loss plots from the log files collected on the destination node. Recall that the flow definitions for our example experiment are from source *$enclave1_app1_node$* to *$enclave2_app1_node$*. To view the goodput plot at the destination (as a PNG image file), simply do the following:

cd /home/<user>/iron_results/<date_time>/3enclave_example_exp/run1/enclave1/app1/results

eog mgen_goodput.png

10. Common Debugging Techniques

If the generated experiment results do not look as expected, it may be necessary to re-run the experiment after reconfiguring it to generate additional debugging information to aid in figuring out what is going on. Two common modifications include changing the log levels to generate additional diagnostic output from the IRON components and enabling packet captures in various places in the IRON network.

10.1. Experiment Logging

The log level of the core IRON components, BPF (BackPressure Forwarder), TCP Proxy, and UDP Proxy, is controlled by the configuration item *Log.DefaultLevel* found in the *bpf_common.cfg*, *tcp_proxy_common.cfg*, and *udp_proxy_common.cfg*, respectively. The format of the value for the configuration item is any combination of the letters F (Fatal), E (Error), W (Warning), I (Information), A (Analysis), and D (Debug). By default, the log level is set to "FEW". Note that increasing the log levels can lead to very large log files as some of the diagnostic output is per packet.

If turning up the log levels generates too much logging output, each of the components can be configured to support increased logging on a finer-grained scale, by enabling and configuring class level logging. The configuration item *Log.ClassLevels* in the *bpf_common.cfg*, *tcp_proxy_common.cfg*, and *udp_proxy_common.cfg* files (found in the experiment configuration directory) accomplishes this. The format for specifying this configurable item is as follows:

Log.ClassLevels ClassName1=LogLevel1;ClassName2=LogLevel2;...;ClassNameN=LogLevelN

The following entry in the tcp_proxy_common.cfg configuration file configures the Socket class to log at levels FEWI and the SendBuffer class to log at levels FEWIAD:

Log.ClassLevels Socket=FEWI;SendBuffer=FEWIAD

The class level logging is generally useful during the development stage when turning up the log levels component-wide generates so much logging output that it is difficult to understand.

10.2. Experiment Packet Captures

Another useful debugging technique is looking at packet captures. The *exp.tmpl* file that was generated when we created our experiment is where we identify where, in the IRON network, we want the packet captures to occur. The process of creatig this file is fully described in the *Creating A New Experiment* section. The commented-out *PCAPS* and *DECAP* lines is where we specify this information. To avoid possible performance implications, we typically do not do packet captures on the IRON nodes. This leaves the application and link emulation nodes in our example experiment as available places to capture packets. The application node has a WAN-facing interface and the link emulation node has a LAN-facing (toward the application) and a WAN-facing (toward a remote enclave) interface. To identify where packets are capture, we need to identify the node and link in "configuration template replacement" string notation. This leads to the following possibilities for our packet captures, entered in the *PCAPS* entry in the *exp.tmpl* configuration template:

- *$enclaveX_app1_node$:$enclaveX_app1_wan_link$*: The WAN-facing link on the application node in enclave X
- *$enclaveX_le1_node$:$enclaveX_le1_lan_link$*: The LAN-facing link on the link emulation node in enclave X.
- *$enclaveX_le1_node$:$enclaveX_le1_wan_link$*: The WAN-facing link on the link emulation node in enclave X.

The following entry instructs the automated experiment execution scripts to capture packets on the LAN-facing interface of the link emulation node in enclave 1:

PCAPS=($enclave1_le1_node$:$enclave1_le1_lan_link$)

Note that the packets captured on the link emulation nodes are IRON CAT (Capacity Adaptive Tunnel) encapsulated packets. The *sliqdecap* utility can be used to decapsulate the packet capture after the experiment completes and the artifacts are collected and placed in the results directory previously described. The automated experiment execution scripts can also automatically decapsulate the packet captures on the link emulation nodes prior to the data collection phase. This

is controlled by configuring the *DECAP* entry in the *exp.tmpl* configuration template. To identify which packet captures are to be decapsulated, we need to identify the node, link, and type of decapsulation in configuration template replacement string notation. Assuming that we capture packets on the LAN-facing interface of the link emulation node, the following line will instruct the automated experiment execution scripts to decapsulate it for us:

DECAP=($enclave1_le1_node$:$enclave1_le1_lan_link$:sliq)

Note that the **:sliq** portion of the above example instructs the automated experiment execution scripts to use the *sliqdecap* utility to decapsulate the packet capture. The end result of doing this is a packet capture in which the IRON CAT headers have been removed, leaving the original application flows in the resulting packet capture file.