BYOI: Bring Your Own Internet to Research and Education

Kyungrok Won and Wenliang Du
Syracuse University
Syracuse, New York, USA
{kwon01,wedu}@syr.edu

ABSTRACT

The SEED Internet emulator allows us to create a miniature Internet inside a personal computer. While this emulator standalone can have many applications, in many applications, it is often more desirable if external computers and devices can join the emulated Internet. For example, when using the emulated Internet in class, it will be more interesting if student can participate in the emulator using their computers or mobile devices. In IoT research, allowing IoT devices to connect to the emulator enables researchers to conduct investigation that is hard to do on the real Internet.

Allowing our emulator to join the real Internet is also useful for research and education. This way, we do not need to duplicate everything inside the emulator; we can leverage many services in the real Internet. We show how to achieve this in the SEED emulator. Moreover, using a single computer to do the emulation limits the scale of the emulation. To solve this problem, we have developed a mechanism to allow distributed emulation by joining multiple emulators to form a single and larger scale emulator.

KEYWORDS

Internet emulation, networking, BGP

1 BRING YOUR OWN INTERNET

The SEED Internet emulator can be used as a standalone emulation, but in many scenarios, it is often necessary to connect external computers and devices to the emulator. For example, when we use the emulator in the class to demonstrate an interesting attack on the Internet, if we can let students feel the impact, instead of just seeing the attack, the effectiveness on learning will be better. To feel the impact, students' computers or smartphones should be connected to the SEED emulator. Another use case is the IoT devices. For IoT research, to benefit from the Internet emulator, it is necessary to let the physical IoT devices join the emulation.

The problem that we are trying to solve is how to let external physical devices join the emulation, which consists entirely of virtual devices. To solve this problem, we need to allow an external device to join one of the networks inside the emulator. There are two specific problems that we need to solve. First, networks inside the emulator are virtual, so how to attach a physical device to a virtual network? Second, once the device is attached to the network, it needs to get an IP address. We need to create an DHCP server inside the emulator to do this.

1.1 Setting Up DHCP

When an external device is attached to a network inside the emulator, it must be assigned an IP address. This IP address cannot be arbitrary; it must be routable inside the emulator, or nobody can reach it. Inside the emulator, each network's prefix is announced by the BGP routers of its autonomous system, so other hosts inside the emulator know how to reach this network. When an external device is attached to such a network, it must use the same network prefix.

Assigning IP address automatically is typically conducted by a DHCP server on the network. To achieve this goal, we have implemented a DHCP component, which allows us to easily add a DHCP server to a network. The following code snippet creates a DHCP server, and then install it to a host inside AS-151.

```
# Create a DHCP server (virtual node).
dhcp = DHCPService()

# Default DhcpIpRange : n.n.n.101 ~ n.n.n.120
# Set DhcpIpRange : n.n.n.125 ~ n.n.n.140
dhcp.install('dhcp-01').setIpRange(125, 140)

# Customize the display name (for visualization)
emu.getVirtualNode('dhcp-01')
   .setDisplayName('DHCP Server 1')

# Create a new host in AS-151
# Use it to host the DHCP servers.
# We can also host it on an existing node.
as151 = base.getAutonomousSystem(151)
as151.createHost('dhcp-server-01').joinNetwork('net0')

# Bind the DHCP virtual node to physical node.
emu.addBinding(Binding('dhcp-01', filter =
           Filter(asn=151, nodeName='dhcp-server-01')))
```

With this setup, when an external host is attached to the `net0` of AS-151, if it is configured to use DHCP, it will automatically get an IP address in the range of from `10.151.0.125` to `10.151.0.140`, where `10.151.0.0/24` is the network prefix assigned to `net0`.

1.2 Hardware Setup

We show how to attach an external device to a network inside the emulator. The main idea is illustrated in Figure **??**. On the outside, all the external devices are either directly plugged into a switch, or connected to a WiFi access point that is plugged into a switch. This switch is then connected

to the targeted network. This setup is standard, similar to the network connection in the real world.

The main challenge is how to connect the physical switch to the network inside the emulator, so the physical switch becomes an extension of the emulated network. Therefore, when an external device is attached to this switch, it is essentially attached to the network inside the emulator, and thus becomes a member host of the emulation.

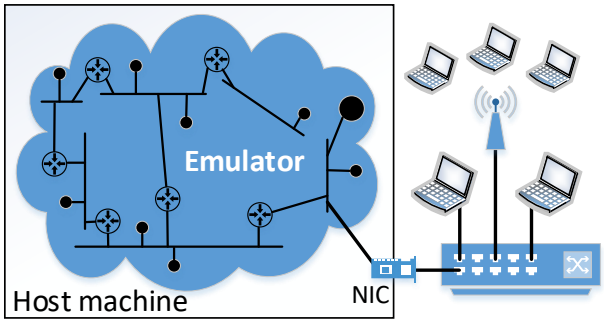

Figure 1: Connecting external devices to emulator

Networks inside the emulator are created by docker. They are virtual and they are actually implemented using Linux bridge, which behaves like a network switch and forwards packets between interfaces that are connected to it. In docker, attaching a container to a virtual network essentially attaches the container's network interface (virtual) to a Linux bridge. We can also attach a physical network interface to this bridge. After that, all the devices connected to the other end of this interface also become attached to this Linux bridge, and thus become attached to the virtual network corresponding to the bridge.

We show how to connect the NIC in Figure 1 to a virtual network inside the emulator. We can pick any network. In our explanation, we choose `net0` from AS-151 as an example.

We need to first find out the name of the physical interface. After running the emulator, if we use the `"ip address"` command to list all the network interfaces, we can get many, because the list includes all the virtual network interfaces and bridges created for the emulator. To hide these interfaces, we use the `grep` command to exclude the interfaces that have `veth` or `br` in their names.

```
$ ip -br addr | grep -vE 'veth|br'
lo                UNKNOWN        127.0.0.1/8
enp0s3            UP             10.0.5.5/24
docker0           DOWN           172.17.0.1/16
enx7cc2c633b399   UP             fe80::1c05:939...
```

In our setup, the interface `enx7cc2c633b399` is the one connected to the external physical switch, so we are going to use this name. Make sure that the interface is in the `UP` state before moving forward. More instruction regarding the interface can be found in Section 1.3.

Next, we need to find out the name of the Linux bridge corresponding to AS-151's `net0`. We know this network's prefix is `10.151.0.0/24`, so we can use the following command to find the bridge.

```
$ ip -br addr show to 10.151.0.0/24
br-425e1d573afc UP             10.151.0.1/24
```

We now add the physical interface `enx7cc2c633b399` to this Linux bridge using the `"ip link"` command. After doing that, we can also list all the interfaces connected to the bridge. The entry marked by ☆ is the one added by us. The other entries are for the containers attached to the same network.

```
$ sudo ip link set enx7cc2c633b399 master \
                   br-425e1d573afc

$ ip -br link show master br-425e1d573afc
veth40a62e5@if1577 UP   7e:c8:52:05:85:1a ...
enx7cc2c633b399    UP   7c:c2:c6:33:b3:99 ...    ☆
veth604dd32@if1411 UP   02:14:0c:6c:41:5f ...
veth883181a@if1495 UP   0e:78:c8:73:cc:9b ...
vethe028bd9@if1499 UP   9e:42:e5:8f:74:6b ...
veth36692ec@if1501 UP   32:bd:c9:5a:a0:27 ...
```

As soon as the physical interface is added to the AS-151 network, all the external devices plugged into the physical switch will get an IP address from the `10.151.0.0/24` network, and therefore they become a host on the network `net0` inside AS-151. They should be able to communicate with all the hosts inside the emulator.

1.3 Adding a Network Interface

The most tricky part of the process is the physical network interface involved in the setup. It depends on how we run the emulator.

If we run the emulator inside a Ubuntu machine, the setup will be the easiest. If the machine has a built-in ethernet interface, we can use this one. If not, we can get a USB-to-Ethernet adapter, so we can add an Ethernet interface to our machine. We do not need to do much of the setup in this case. All the commands used in Section 1.2 are from the Ubuntu operating system.

The emulator can also run directly inside other types of OS, such as Mac OS, but that will make the hardware setup described in Section 1.2 difficult, if possible at all. For example, To our limited knowledge, adding a physical network interface to a virtual network created by docker is not possible in the Mac OS.

If we run the emulator inside a virtual machine, the situation will be a little bit more complicated. All the network interfaces created for the virtual machines by the VM software (such as VirtualBox) are virtual, and in our experience, we have found out that adding them to the docker's virtual network (i.e., bridge) did not produce a reliable result. The best way is to use a physical network interface. The question

is how to get a physical network interface on the host into the VM.

It will be difficult, if possible at all, to expose a physical ethernet network interface on the host machine to the VM, because VM software typically do not pass this physical device into the VM. However, most VM software, such as VirtualBox, does support USB pass through, i.e., a USB device connected to the host machine can pass through the host machine and become connected to the virtual machine. Therefore, we can use a USB-to-Ethernet adaptor to connect an physical ethernet network interface to the virtual machine.

There are many instructions on the Internet regarding how to add a USB device to a virtual machine, so we will not include one in this document. The procedure depends on the type of virtual machine software, but it is quite similar and straightforward.

1.4 Connecting at Multiple Locations

Connection to the Internet emulator can be conducted at multiple locations. Figure 1 only shows the connection at one location, but we can use the same method to connect external devices to the emulator at multiple locations. We just need to find another network inside the emulator, and connects a physical network interface to that network. This allows us to more realistically emulate the real world, where devices joining the Internet from different locations.

1.5 Applications

We have conducted an experiment using our setup. In the experiment, we put our Internet emulator on a laptop, and hook a WiFi access point to the emulator. We brought the setup to the classroom, asking students to connect their machines or smartphones to our WiFi. By doing that, students immediately became the participants of the emulation. The process was quite smooth, and it did not have much difference compared to connecting to the campus WiFi.

We then conducted the BGP network prefix hijacking inside the emulator, aiming to hijack our university's network prefix. Before the attack, students were able to access the university's website (due to our hybrid emulation setup, which will be discussed later), but as soon as the attack was launched, the university's website was no long accessible, while students could still access the rest of the Internet. Obviously, the attack only affects the routing inside the emulator, not the real world.

2 DISTRIBUTED EMULATION

Due to the resource limitation (CPU and RAM), conducting emulation on a single machine can be limited. To solve this problem, we have developed a mechanism to easily scale up the emulation by joining multiple emulators running on different computers together. These computers can be in the same physical location or remote (on the cloud), and they each run an instance of emulator. In this section, we describe how to join these emulators together to form a larger emulation. Being able to join multiple emulators enable users to easily expand the scale of the emulation.

2.1 Bridging Internet Exchanges

The merging of two emulators is done by simply bridging two Internet exchanges from the two emulators. Assume that we have two Internet emulators A and B, and we would like to join them to form a larger emulation. To achieve that, A and B should have a common Internet Exchange (say IX-100), which is emulated using a network. Therefore, A has one part of the network IX-100, and B has the other part. Bridging these two parts of the network will result in the merging of A's IX-100 and B's IX-100. See Figure 2.

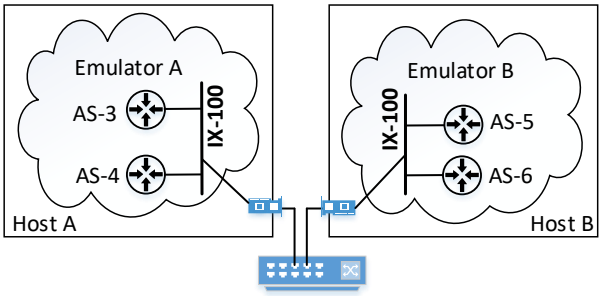

Figure 2: Bridging Internet Exchange

Once these two IXes are bridged, the BGP routers connected to A's IX-100 network and the BGP routers connected to B's IX-100 network are now connected to the same network, and they can peer with one another. After peering sessions are set up among these BGP routers, packets can cross the bridge from one emulator to another. For example, after the bridging, we can peer AS-3 with AS-5, and these two BGP routers will start forwarding the network prefixes that they know to each other. As results, AS-3 will now know how to reach the networks in the emulator B, and AS-5 will know how to reach the networks in the emulator A.

Two emulators can connect at multiple locations (IXes), so they can reach each other via different paths. We just need to bridge each pair of shared IXes between these two emulators. Moreover, multiple emulators (more than two) can be merged at the same location, as long as they share the same IX.

2.2 Bridging Using Switch

Bridging two IXes from different emulators is similar to connecting external devices to an emulator. As we can see in Figure 2, the physical switch is connected to the IX's network via a physical network interface. This setup is exactly the same as that described in Section 1.2. The only difference is that for connecting external devices, we pick an autonomous system's network, while here we pick an Internet exchange's network. Since we will not add new hosts to the network, there is no need to add an DHCP server to the network.

Once the same physical switch is connected both IX networks, the IX networks in the two emulators are connected at the Layer 2, i.e., they become one single network.

2.3 Bridging Using VPN

Our emulator supports both physical bridging (using a physical switch, for machines at the same location) and virtual bridging (using Layer-2 VPN, for remote machines). Detailed instructions for this part will be added later. Contact us if this is what you need.

2.4 BGP Peering

After bridging the IX networks from the two (or more) emulators, we need to set up the BGP peering, so the BGP routers on the merged IX network can exchange network traffic. To peer two autonomous systems at an IX, we need to modify their BGP routers' configuration file `/etc/bird/bird.conf`, and reload the configuration file into the routing daemon using `"birdc configure"`.

We use an example to show how to peer AS-2 with AS-5 at IX-100. In our emulator, IX-100's network has a prefix `10.100.0.0/24`, and AS-N's BGP router on this network has an IP address `10.100.0.N`, where `N` is the autonomous system number. We add the following to AS-2's BGP configuration to specify that AS-2 peers with AS-5.

```
protocol bgp p_as5 {
  ipv4 {
    table t_bgp;
    import filter {
        bgp_large_community.add(PEER_COMM);
        bgp_local_pref = 20;
        accept;
    };
    export where bgp_large_community ~ [LOCAL_COMM,
                      CUSTOMER_COMM];
    next hop self;
    };

  local    10.100.0.2 as 2;
  neighbor 10.100.0.5 as 5;
}
```

The `local` option in the configuration specifies which AS the router belongs to and the IP address of the router. In this case, AS-2's IP address `10.100.0.2` is used. The `neighbor` option specifies the IP address of the peer and what AS it belongs to. This is the actual peering part. In this example, we set up a BGP session between `AS-2` and `AS-5`, so they can exchange route information using the BGP protocol. The IP address of AS-5's BGP router on this network is `10.100.0.5`.

It should be noted that when A and B peer with each other, it is not necessary that A and B will tell each other every network prefix that they know. What they tell each other depends on their peering relationship. In the BGP session above, AS-2 and AS-5 have peer-to-peer relationship. That is why in the `"import filter"` section, all the routers from AS-5 are marked as `PEER_COMM`. With this relationship, AS-2 only export the routes originated from itself or its downstream customers. AS-2 will not export to AS-5 the routes from its upstream providers. This is specified in the `export` filter. If we would like to overwrite this rule, we can replace the `"export where ..."` entry with `"export all"`.

Peering entry also needs to be added to AS-5's BGP router, specifying that AS-5 peers with AS-2.

```
protocol bgp p_as2 {
  ipv4 {
    table t_bgp;
    import filter {
      bgp_large_community.add(PEER_COMM);
      bgp_local_pref = 20;
      accept;
    };
    export where bgp_large_community ~ [LOCAL_COMM,
                      CUSTOMER_COMM];
    next hop self;
  };

  local    10.100.0.5 as 5;
  neighbor 10.100.0.2 as 2;
}
```

Even if AS-2 or AS-5 knows all the routes from its own side and export all of them to each other, it still does not guarantee that all the routes from one emulator can be propagated to the other emulator. This is because whether AS-2 or AS-5 will further propagate the routes learned from each other to its other peers depend on its peering relationship with them. That is just the nature of BGP, and also how things work in the real world. BGP peering relationship is quite complicated. Readers can find more details from this document (link). To improve the chance for all the networks on one side to be reachable from the other side, peering with multiple autonomous systems is suggested.

2.5 Overlapping Network Prefixes

When building distributed emulation, other than the shared Internet exchanges, we should try to avoid having overlapping network prefixes among different emulators. However, even if that happens, it is not a problem. They will be simply treated as IP anycast. Packets to the overlapping network prefixes will arrive at one of the networks; which network gets the packets depends on the routing. TCP services on such a network may have issues, as the other end of the TCP connection may change in the middle of a connection.

3 HYBRID EMULATION

By default, the SEED Internet emulator is isolated from the outside: If a node tries to send packets to the outside, the packet will be dropped by routers, because BGP routers do not know how to route the packets. This closed emulation by itself has many applications, but the scope of the applications

can be broaden if we allow a hybrid emulation, i.e., allowing the machines inside the Internet emulator to communicate with those on the real Internet. See Figure 3.

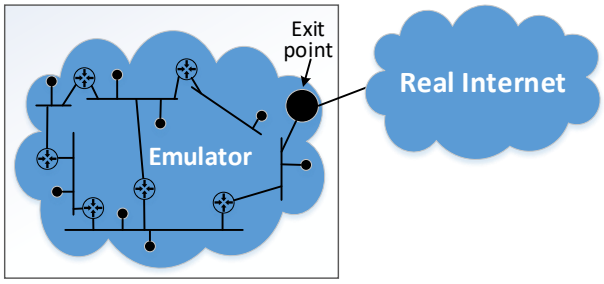

Figure 3: Hybrid emulation

With the hybrid emulation, we do not need to duplicate everything inside the emulator. For example, if there is a useful web service that we would like to include in our emulation, we do not need to duplicate that inside the emulation. Instead, we can just directly use it using our hybrid emulation technique.

Another application is to emulate the denial-of-service attack on the real-world target. With the hybrid emulation, our emulator can be used as a shadow Internet. Using the setup described in Section 1, students visiting the real Internet will go through the emulator. Their packets will eventually exit the emulator and reach the final destination. If we want to emulate a real denial-of-service attack against a real-world target, such as `example.com`, we can now do it inside the emulator, which to the user, is equivalent to the attack inside the real Internet.

3.1 Hybrid Internet

To allow the emulated Internet to reach the real Internet, we need to create an exit point inside the emulator. Packets going to the real Internet will be routed to this exit point, from where the packets will be forwarded to the real Internet. Multiple exit points can also be created.

We create an autonomous system inside the emulator, and use this AS as the exit point. The BGP router in this AS will announce two prefixes inside the emulator: `0.0.0.0/1` and `128.0.0.0/1`. These two network prefixes cover the entire IPv4 address space, so if a destination IP address does not match with any other entry in the routing table, it will match with one of these two prefixes. This makes the exit point a default destination: if a packet does not go to any network inside the emulator, it will be routed towards the exit point. If multiple exit points announce the same network prefixes, it will be treated as IP anycast, which is naturally supported by BGP.

If we only want to reach certain network in the real Internet, instead of every network, we can narrow down the scope of the prefix announcement. For example, if we only want to include the `example.com` in our emulator, we can just announce the `93.184.216.0/24` prefix from the exit point.

Once a packet to the real-world Internet reaches the BGP router of the exit point, the router will forwarded the packet out after conducting the NAT (Network Address Translation). NAT is necessary because we want the return packets to come back to the exit point, from where the packet can be forwarded to the emulated Internet.

The following code snippet shows how to create and set up an exit point inside the emulator. In this example, we create an autonomous system `AS-99999`, connect its BGP router to the Internet Exchange `ix100`.

```
# Create hybrid AS.
# AS99999 is the emulator's autonomous system that
#   routes the traffics to the real-world internet
as99999 = base.createAutonomousSystem(99999)
router  = as99999.createRealWorldRouter('rw-real-world',
                prefixes=['0.0.0.0/1', '128.0.0.0/1'])
router.joinNetwork('ix100', '10.100.0.99')
```

The `createRealWorldRouter()` API will create and set up an exit-point BGP router, so it can announce the specified network prefixes inside the emulator, as well as setting an NAT server to conduct the network address translation. Other than these special requirements, this BGP router is just like any other BGP router inside the emulator.

Show some testing results ...

3.2 Hybrid DNS Infrastructure

DNS is an essential infrastructure for the Internet, so it is supported in our emulator. The SEED Internet emulator can host its own DNS infrastructure, from the root servers, to TLD servers, and to the domain servers. To support the hybrid emulation, we need to include the real-world domains, so we can find the IP addresses of the machines on the real Internet. For example, if we want to visit the real `www.google.com` website from inside the emulator, our DNS infrastructure should help us resolve the IP address for this domain.

We support a hybrid DNS infrastructure, with some of the nameservers hosted inside the emulator and some hosted in the real Internet. Figure 4 illustrates the hybrid infrastructure, where the shaded nodes represent zones hosted inside the emulator, and the non-shaded nodes represent zones hosted in the real Internet.

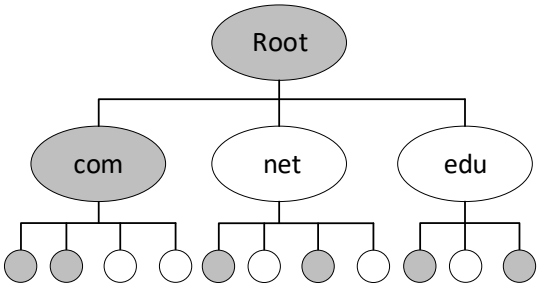

Figure 4: Hybrid DNS infrastructure

Hosting a hybrid DNS infrastructure like what is illustrated in Figure 4 is quite challenging. There are two scenarios that need to be dealt with differently. In the first scenario (the `com` branch in the hierarchy of Figure 4 is an example), the parent zone is inside the emulator. We can easily add the child zone's nameservers to the parent zone, regardless of whether the child zone is hosted inside the emulator or not. See the following code example.

```
# Create the root and com zone
dns = DomainNameService()
dns.install('root-server').addZone('.')
dns.install('ns-com').addZone('com.')

# Create the example.com zone
dns.install('ns-example-com').addZone('example.com.')
dns.getZone('example.com.').addRecord('@  A  1.2.3.4')

# Register google.com's nameserver to the com server
dns.getZone('com.')
   .addRecord('google.com  NS  ns1.google.com.')
   .addRecord('google.com  NS  ns2.google.com.')
   .addRecord('ns1.google.com A 216.239.32.10')
   .addRecord('ns2.google.com A 216.239.34.10')
```

In this example, we host the `example.com` nameserver inside the emulator, while using the real-world's nameserver for the `google.com` domain. For the first case, the `example.com` domain's nameserver will be automatically added to the `com` server. For the latter case, we need to manually add the `google.com` domain's nameserver information to the `com` nameserver.

The first scenario is not suitable if we need to register many child zones from outside to the parent zone. It is not just the amount of work involved; it is also the complexity, because some real-world domain's DNS is quite complicated. Therefore, the second setup is more appropriate (see the `net` branch in Figure 4). In this setup, we directly use the outside nameservers for the parent zone. The challenge that we now face is that if we host a child zone inside the emulator, how do we register it with the real-world parent zone?

There is no easy way to do that, because in order to register a child zone to its parent zone (which is real), we need to actually own the child zone. That is not feasible. We use a work-around to solve this problem. We create a forward zone in the local DNS server to directly forward the query to the child nameserver, instead of going through the entire process, from root zone, TLD zone, and so on. This way, we completely bypass the parent zone if the child zone of this parent is hosted inside the emulator, so there is no need to register to the parent zone.

In the following example, we create a nameserver for the root zone. We use the `setRealRootNS()` API to specify that the real-world zone file should be used to configure the root zone. The API will download the real root zone records from the Internet.

```
# Create the root zone
dns = DomainNameService()
dns.install('root-server').addZone('.').setRealRootNS()

# Create nameservers for example.net
dns.install('ns-example-net').addZone('example.net.')
dns.getZone('example.net.').addRecord('@  A  1.2.3.4')

# Create a local DNS server
ldns = DomainNameCachingService()
ldns.install('global-dns')
    .addFowardZone('exampple.net.', 'ns-example-net')
```

In this example, we did not create any TLD zone. Because the root zone file is from the real world, so the TLD servers from the real Internet will be used. We do want to host a second-level domain `example.net` inside the emulator. To do that, we added a forward zone to the local DNS server. This way, if the machines inside this emulation sends a DNS query for `google.com`, the query will go to the real google nameserver. However, for the query for `example.net`, the local DNS server will take a "shortcut" and directly forward the query to the emulated `example.net` nameserver.

4 SUMMARY

The SEED Internet emulator is an open-source project, which was developed initially for cybersecurity education, but its scope is now expanded beyond that. Its objective is to help users easily create a miniature Internet with required services deployed, so they can use the emulator to evaluate their research ideas. While the emulator is fully functioning and is already adopted by others, it is still under active development. More features will be added in the near future. We welcome other people to try the emulator, give us feedback, and help us make this open-source software to better serve the research and education community.

ACKNOWLEDGMENTS

The SEED emulator project was funded in part by the US National Science Foundation (No. 1718086) and Syracuse University's Meredith Professorship grant and CUSE grant.