From Simulation to Emulation- An Integrated Approach for Network Security Evaluation

Martin Möller, Rainer Bye, Karsten Bsufka, Ahmet Camtepe, Sahin Albayrak
DAI-Labor, Technische Universität Berlin
martinsemails@googlemail.com, rainer.bye@dai-labor.de, karsten.bsufka@dai-labor.de,
ahmet.camtepe@dai-labor.de, sahin.albayrak@dai-labor.de

Abstract: We present a virtual test bed for network security evaluation in mid-scale telecommunication networks. Migration from simulation scenarios towards the test bed is supported and enables researchers to evaluate experiments in a more realistic environment. We provide a comprehensive interface to manage, run and evaluate experiments. On basis of a concrete example we show how the proposed test bed can be utilized.

1 Introduction

The design and development of security solutions such as Intrusion Detection Systems (IDS) is a challenging and complex process. The evolving system needs to be evaluated continuously from the first idea to an implemented prototype. There exist several ways to study a system, whereas the most accurate one is the analysis of the deployed solution in the production environment. However, in the case of IDS evaluation, real experiments incorporating attack scenarios can not be done in the operational environment. The induced risk of failures, such as service loss is too high. In addition, at the beginning of the IDS development process there exists no finished system to be evaluated.

A simulation environment simplifies the studied problem and allows researchers and developers to concentrate on the most critical issues [LPD10]. However, if all experiments and analysis have been done and one wants to realize this protocols or application for real world use there is a need for a more realistic environment for testing and evaluation. For this very reason, evaluation is often carried out in small testbeds, but as the environment usually needs to consist of several hosts and network equipment it is costly to be maintained and configured. Virtual machines are a solution for modeling mid-scale networks, but the simulated experiments need be recreated from scratch.

In the scope of this work, we present a solution to migrate simulation experiments from *NeSSi*^{2}, the Network Security Simulator, into a *scenario-based virtual testbed*. Scenario-based virtual test beds enable the description and execution of network elements, topology and configuration of the deployed hosts in a configuration language. It provides a central user interface from where experiments can be defined, executed and analyzed. Instead of using simulation we use virtualization. The hosts execute a Linux kernel and are able to communicate via the standard Linux network stack. The overall approach provides the

desired realism necessary for evaluation of security solutions while keeping administrative cost low. First we will briefly describe *NeSSi*^{2} and then we discuss scenario-based test beds. We present our approach, conduct a case study and conclude in Section 6.

2 *NeSSi*^{2} Simulation Experiments

NeSSi^2 is a packet-level, discrete-event simulator [SBC+10]. It is built upon the JIAC agent framework [HKH09], which is utilized in modeling and implementing the network participants such as routers, clients, and servers as software agent entities. The front-end consists of a graphical user interface that allows the creation of arbitrary IP network topologies. An experiment in *NeSSi*^{2} consists of the following parts:

The *network topology* describes the static information contained in the network, i.e. the nodes of the network, their (initial) properties and how they are interconnected. Node *profiles* allow the customization of network node behavior in order to automatically generate traffic adhering to well-defined characteristics, simulate network outages by means of router or link failures as well as evaluating network-based defense measures. The basic elements used to describe these profiles are applications: an *application* represents any kind of protocol or mechanism to be executed on an individual node, e.g. a spreading attack, a detection unit or a benign application protocol. A *scenario* contains behavior definitions attached to a particular network topology. For instance, a scenario may contain the traffic generation properties for different nodes in the network. Finally, the *session* information contains additional information specific for the selected scenario. For example, the user may be interested only in logging a certain type of traffic, or the scenario should be carried out repeatedly, but with different amount of ticks (Tick is the term for the atomic discrete time unit). This kind of information is encapsulated in the session object for a particular scenario.

3 Related Work

VNUML [GFF+09] can be used to simulate Linux-based network applications and services. The testbeds can be comprised of tenth of nodes and networks inside one Linux machine. VNUML consists of two parts. A XML based language to describe scenarios. A scenarios may consists of several virtualized hosts and networks. The second part are a set of Perl scripts that enable scenario deployment. Besides the plain deployment of User Mode Linux (UML) machines the XML language provides basic configuration options out of the box to configure virtual machines. Networks are realized by using TUN/TAP devices and UML switch. *EDIV* [GFdVM09] is an evolution of VNUML to allow the management of virtualization scenarios deployed on multi-host distributed infrastructure. It takes VNUML simulation description and splits it into smaller ones that can be distributed to several physical machines. EDIV then takes care of connecting the virtual networks in way that is transparent to the user.

Netkit [PR08] similar to VNUML provides virtual test beds by using UML and Linux network utilities such as TUN/TAP. In contrast to VNUML it does not use a XML file to describe the experiments, but uses several configuration files, to describe the scenario. Netkit aims to be utilized for education in network administration. *Emulab* [WLS+02] unlike other presented projects uses dedicated hardware experiments. Drawbacks of Emulab are administrative efforts and the need for certain hardware. Actually, a dedicated test bed administrator is recommended in the official documentation[1].

4 Approach

Simulation tools such as *NeSSi*^{2} don't consist solely of the simulation model but include tools to setup, run, rerun and analyze experiments. Out approach is to provide those tools for a test bed that consists of virtualized computers. Instead of a simulation model we use virtualization. Additionally we provide an interface to import *NeSSi*^{2} scenario descriptions in our virtual test bed. The interaction with the test bed is primarily done via a web browser interface, but shell access to virtual machines is possible as well.

4.1 Architecture

The test bed consists of two main building blocks (cf. figure 1). The test bed controller and the emulation part. The controller provides the main user interface via a web server. It manages the experiments and their deployment process. Therefore it generates a VNUML network description and passes it to EDIV that splits the network into parts and deploys them on a cluster of servers. The communication between the virtual machines residing on different servers is possible via VLAN interfaces on the servers. For each virtual network a VLAN id is chosen and corresponding sub interfaces are configured on the servers.

We utilize customized file system and kernel image, derived from the ones the VNUML project provides. The original image already contains software such as the *Quagga Routing Suite*[2] and common networking tools such as *nmap* or *tcpdump*. It is straight forward to add new software and we included software we utilized during our experimentations for instance *snort*, *smb* support and some scripts. Bash, perl and python scripts can be executed and used for application prototyping or custom configuration. The virtual machines use a copy on write file system. Thus all changes that happen during an experiment will be reseted on shutdown and experiments can rely on a consistent file system state.

To be able to collect data our test bed controller implements a syslog server. It collects syslog messages that are originated from the virtual host and stores them in a database. Thus it is as easy as running `logger 'hello world'` in a bash script or console to generate data from within the virtual machines.

[1] `https://users.emulab.net/trac/emulab/wiki/install/prerequisites.html`
[2] `http://www.quagga.net/`

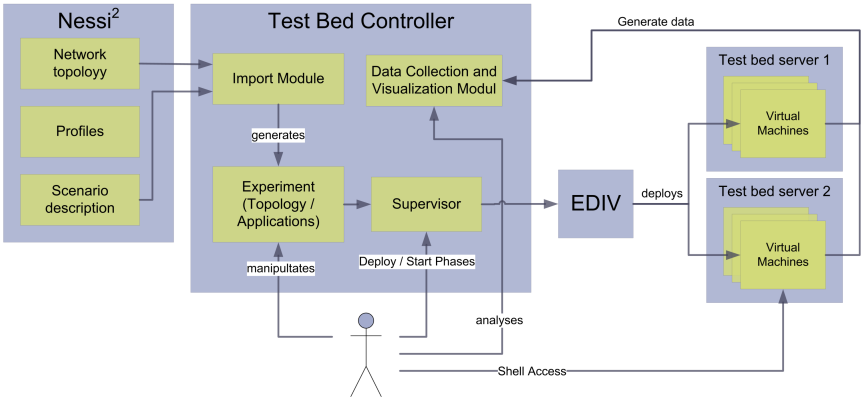

Figure 1: Architecture Overview

4.2 Interface

Using the web based interface users can configure experiments similar to *NeSSi*^{2}. The first step is to generate a new experiment. This can be done by importing an existing network topology from a *NeSSi*^{2} project or by starting with an empty topology. Once the experiment is created it can be customized by adding hosts, switches and hubs. Hosts can be configured in detail including Ethernet interfaces, routes and IP forwarding. The network graph can be plotted and nodes can be inspected (cf. figure 2).

An typical experiment will include application that are executed on the hosts and communicate via the network. It is possible to define several phases and attach applications to them. A typical experiment would consists of a *setup* phase, a *execution* phase and a *shutdown* phase. Once the test bed is deployed we can execute those phases. Applications are implemented as scripts. Currently we support bash and python scripts which are stored in a repository. They can be created and edited via the web interface.

Setting up a computer network is not a trivial task. Every network interface has to be configured with IP addresses and and routes need to be set for very host. To avoid those time consuming and error-prone tasks we provide methods to automatically set valid IP addresses and routes. In addition we provide a set of scripts for common tasks like starting a web server or to set up RIP.

The log database collects messages that are generated by the virtual hosts. We provide filter and grouping options, that can be used to preprocess messages. The filtered results can be exported as text files or plotted as charts using the JavaScript graph library *flot*[3].

Once the experiment is configured it can be deployed to the test bed. Using the supervisor view the deployment can be issued and monitored. It provides system information about the physical servers like memory and CPU usage. The mapping of virtual machines to physical servers and the mapping of virtual networks to vlans is available as well. Once

[3] `http://code.google.com/p/flot/`

all virtual machines are up and running the previously defined phases can be launched. Access to each virtual machine is possible via a `ssh://` link.

4.3 Migrating a *NeSSi²* experiment

We tried to make the transfer of *NeSSi²* experiments into our test bed as seamless as possible. Using wizards it is possible to import the network description. As *NeSSi²* does not simulate the link layer we need to insert switches and hubs into the topology as needed. This can be configured, as well as how routes and IP addresses are handled. *NeSSi²* uses Java classes to implement applications. They can only be executed inside the simulation environment and replacements for them have to be found. In the test bed installed software like HTTP or FTP servers can be used as a replacement. Besides custom applications can be implemented using scripts. We provide a repository with some examples like a web client or a simple IDS which are used in the case study in section 5.

NeSSi² uses the concept of profiles and scenarios to assign applications to hosts. Using our the second wizard it is possible to import scenarios and define how *NeSSi²* applications are mapped to applications and scripts available in the test bed. It is possible to chose during which phase a script or application gets executed. Developing adequate replacements is the most time consuming task as for most *NeSSi²* applications replacement scripts have be developed from scratch. We gained good experience using python for this task. Using python it is possible to prototype applications fast.

The last part is data collection. Application in *NeSSi²* use the Java API to generate messages which are recorded during execution. In the test bed applications can generate syslog messages for this task. The next section presents a case study in which we show how the test bed can be used to evaluate experiments.

5 Case Study: Collaborative AIS

Here we present a reevaluation of a scenario presented by Luther et. al. [LBA+07]. They show how cooperative intrusion detection can help to reduce the false positive rate in anomaly detection. We will rebuild their network topology which consists of a LAN with several clients. At a certain point in time one of them gets compromised and starts a port scan of the local network. In a second scenario with the same topology a worm starts to spread through the local network. First, we shortly introduce artificial immune system (AIS) [HF00], a model for anomaly detection. Then we describe our setup in detail and finally, we present our results of using a cooperative vs none cooperative approach.

The AIS, similar to the biological immune system, is based on the distinction between self and non-self. Initially, an n-dimensional feature space is covered by detectors (i.e., n-dimensional vectors of features such as CPU utilization, memory usage, ..., number of network connections). During a training phase, these detectors are presented to feature vectors describing the self. Matching detectors are eliminated and the remaining are used

for the detection of anomalies. Using a distance measure, it is possible to calculate a threat level, which indicates the level of extraneousness.

5.1 Experiment Set-Up

Figure 2 shows the network topology from the original paper and our rebuild topology. Luther et al. modeled a typical client behavior which consists of NETBIOS, LDAP, Kerberos and NTP traffic. In addition, the clients produce HTTP traffic which varies at specific times per day. All traffic is implemented as UDP packets with correct ports and behavior but without meaningful content. The AIS is implemented on the hosts and has three different phases: monitoring, training, and detection. As we have to use real time in the test bed we used different sessions to do training and detection. For our experiment we collected the feature vectors in the test bed and used a standalone AIS implementation to do the detection.

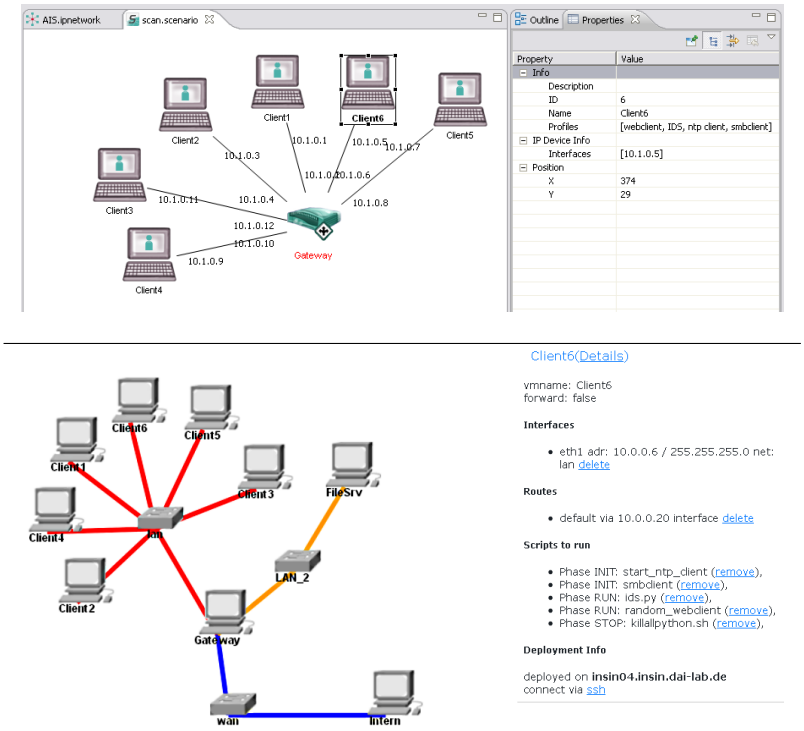

Figure 2: Topology in *NeSSi*^{2} and in the Test Bed Interface

We did not implement exactly their network traffic, but used industry standard software like Apache2 and samba to generate network traffic. As shown in figure 2 we have two servers one is meant to be inside the LAN the other mimics an Internet server. A

router called *Gateway* connects the clients to both of the servers. The internet server runs Apache2 with a cgi script that can generate HTTP responses with arbitrary length. We then use a python script to implement a web browser. This browser fetches a number of websites once in a while. The browser uses a normal distribution to determine the waiting time, number of request and file size. The second server called *FileSrv* runs a samba instance which is an open source implementation of the SMB protocol. We implemented a smb client that accesses a share on this server in a periodic manner. *FileSrv* additionally runs a ntp server which is regularly requested by the clients.

To generate the necessary feature vectors we use a python script on each client. It uses scapy[4] to sniff network packets and generate a *Report* every ten seconds that consists of the following features: *Unix time, normalized time, number of packets, number of TCP / UDP / ICMP / ARP packets, number of TCP connections, number of port scans, number of remote ports*.

5.2 Scenarios

In the first scenario *Client1* gets compromised. It then starts to scan the the subnet. In the original experiment a port scan was simulated. In our case we can use the nmap[5] portscanner. We run nmap with the following command line options:

```
nmap -e eth1 -O --randomize-hosts -n -T polite 10.0.0.1-20
```

It takes around 400 seconds to scan all twenty IP addresses due to the *polite* timing option which restricts the number of packets per second.

In the second scenario a worm starts infecting the network. Initially *Client6* is infected. The worm then tries to infect other hosts by sending UDP packets. He uses a local preference algorithm to choose new targets. Local IP addresses are preferred to other addresses. Once all local addresses have been used it concentrates solely on external addresses. We use a five second interval between each infection attempt. The worm is implemented as a python script. It is started at the beginning of the experiment on every client. It then listens to a certain UDP packet which signals the infection. The worm then tries to infect other hosts on the network.

5.3 Results and Discussion

We used our AIS implementation to calculate threat levels. Figure 3 show the threat level for *Client2* during scenario I. The scan happens between Report 22 and 58 of 113. To calculate the true and false positives we assume that each feature vector inside this interval should indicate a threat. This is only an approximation as we set the "-T polite" option

[4] http://www.secdev.org/projects/scapy/
[5] http://nmap.org/

with nmap to make the scan inconspicuous. We observe that the cooperative approach increases the accuracy for our detection. During the scan the cooperatively calculated threat is higher and outside this time span it is lower (cf. figure 3). The true positive rate increases from 26% to 31% with the cooperative approach.

In scenario II (cf. figure 4) the worm starts spreading at Report 48 from *Client6*. All clients are infected after Report 59. It takes some more time until they have tried to infect the complete sub net. Around Report 83 all of them begin to concentrate solely on external networks what is not recognized by our AIS. If we consider only the interval from 59 to 83 we can observe a slight increase from 94% to 97% for the true positives.

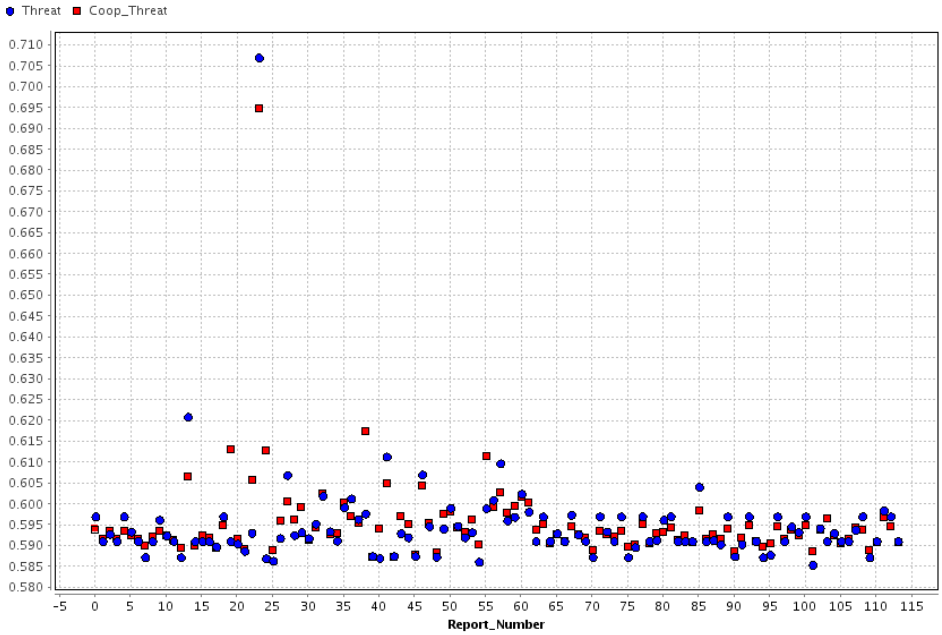

Figure 3: Threat Levels for cooperative and none-cooperative AIS for Client5 in Scenario I

In contrast to the original experiment we can detect the worm from scenario II better than the scan from scenario I. For true positives we observed an improvement similar to the improvements reported by Luther et al. The high true positive rate in scenario II may be explained by the fact that the worm generates a lot of ARP traffic when he tries to infect the whole subnet (253 hosts for each infected host). In scenario I we restricted nmap to scan only 20 hosts what results in less ARP traffic. The fact that a detail like ARP traffic can massively influence the results shows the usefulness of the virtual test bed. For the theoretical development of an IDS it is less important but if one tries to develop a concrete implementation and test it against different types of real world attacks it can be quite valuable using such a tool.

The presented case study utilizes a relative small topology where the savings of the import from *NeSSi*^{2} are smaller than for a large topology. Moreover this was one of our first

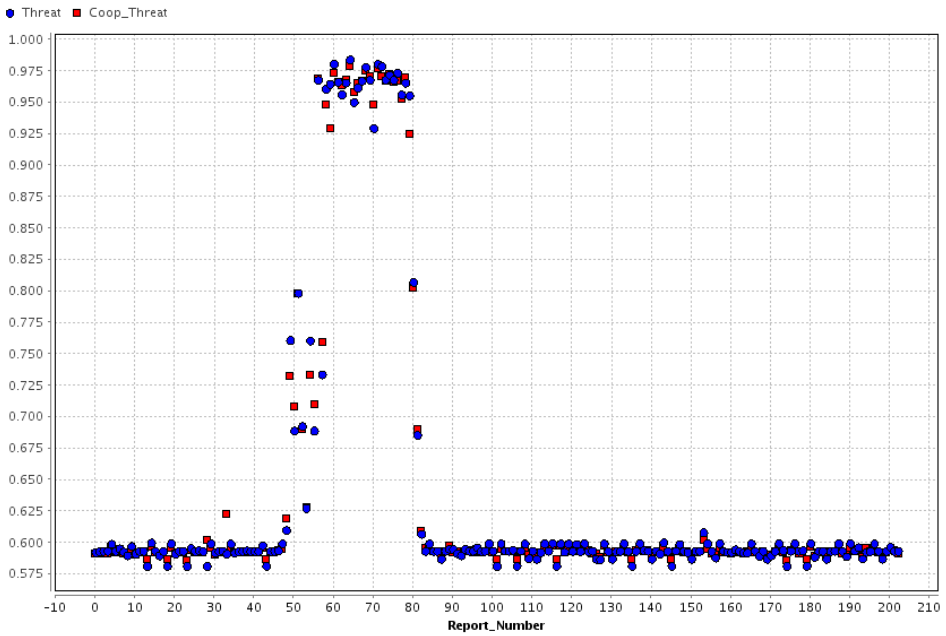

Figure 4: Threat Levels for cooperative and none-cooperative AIS for Client2 in Scenario II

uses of the test bed and we spent most of the with developing the scripts. For subsequent experiments we expect a that the savings will increase as we can reuse existing scripts. If replacements for the network applications exist in form of scripts, nearly the complete experiment can be transformed in a configuration for the test bed.

6 Conclusion

We presented a virtual test bed that is meant to be an addition to simulations tools. We tried to make the transition from a simulation to the virtual test bed as seamless as possible. An import wizard helps to import *NeSSi*^{2} scenarios and a comprehensive graphical user interface helps to manage virtual computer networks. We are able to import the topology and the host configuration from *NeSSi*^{2}. The Java classes that simulate applications inside *NeSSi*^{2} cannot be reused, but we use the profiles to map scripts to host and phases. Finally we presented a case study where the proposed test bed was utilized and we demonstrated how experiments can be implemented to work in the test bed.

Acknowledgments

We thank our fellow colleague Felix Rodemund for his work on adapting the AIS for our needs.

References

[GFdVM09] Fermin Galan, David Fernandez, Jorge E. Lopez de Vergara, and Francisco Monserrat. Demo of EDIV: Building and managing distributed virtualization scenarios in federated testbed infrastructures. *Testbeds and Research Infrastructures for the Development of Networks & Communities, International Conference on*, 0:1–3, 2009.

[GFF+09] Fermín Galán, David Fernández, Walter Fuertes, Miguel Gómez, and Jorge López de Vergara. Scenario-based virtual network infrastructure management in research and educational testbeds with VNUML. *Annals of Telecommunications*, 64:305–323, 2009.

[HF00] Steven A. Hofmeyr and Stephanie Forrest. Architecture for an Artificial Immune System. *Evolutionary Computation*, 8(4):443–473, 2000.

[HKH09] Benjamin Hirsch, Thomas Konnerth, and Axel Heßler. Merging Agents and Services — the JIAC Agent Platform. In Rafael H. Bordini, Mehdi Dastani, Jürgen Dix, and Amal El Fallah Seghrouchni, editors, *Multi-Agent Programming: anguages, Tools and Applications*, pages 159–185. Springer, 2009.

[LBA+07] Katja Luther, Rainer Bye, Tansu Alpcan, Sahin Albayrak, and Achim Müller. A Cooperative Approach for Intrusion Detection. In *IEEE International Conference on Communications (ICC 2007)*. IEEE, 2007.

[LPD10] E. Lochin, T. Perennou, and L. Dairaine. When Should I Use Network Emulation? *ArXiv e-prints*, February 2010.

[PR08] Maurizio Pizzonia and Massimo Rimondini. Netkit: easy emulation of complex networks on inexpensive hardware. In *Proceedings of the 4th International Conference on Testbeds and research infrastructures for the development of networks & communities*, TridentCom '08, pages 7:1–7:10, ICST, Brussels, Belgium, Belgium, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering).

[SBC+10] Stephan Schmidt, Rainer Bye, Joel Chinnow, Karsten Bsufka, Ahmet Camtepe, and Sahin Albayrak. Application-level Simulation for Network Security. *SIMULATION*, 86(5-6):311–330, 2010.

[WLS+02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 255–270, Boston, MA, December 2002. USENIX Association.