5-2015

A Foundation for Cyber Experimentation

Evan Lawrence Stoner

A Foundation for Cyber Experimentation

by
Evan Lawrence Stoner

Bachelor of Science
Software Engineering
Florida Institute of Technology
2013

A thesis submitted to
Florida Institute of Technology
in partial fulfillment of the
requirements for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
May 2015

We, the undersigned committee, hereby approve the attached thesis.

A Foundation for Cyber Experimentation

by
Evan Lawrence Stoner

Marco Carvalho, Ph.D.
Major Advisor
Executive Director, Harris Institute for Assured Information
Associate Professor, Computer Sciences and Cybersecurity

Thomas Eskridge, Ph.D.
Committee Member
Visiting Associate Professor, Computer Sciences and Cybersecurity

---

Richard Ford, Ph.D.
Committee Member
University Professor, Computer Sciences and Cybersecurity

Benjamin Hutz, Ph.D.
Committee Member
Assistant Professor, Mathematical Sciences

Richard Newman, Ph.D.
Interim Department Head, Computer Sciences and Cybersecurity

Abstract

Title: A Foundation for Cyber Experimentation

Author: Evan Lawrence Stoner

Committee Chair: Marco Carvalho, Ph.D.

Despite the variety of environments and tools available to computer scientists performing cyber experimentation, the community lacks a single cohesive platform that is capable of unifying these tools across environments and research domains. In pursuit of this unifying platform, we introduce a new framework that is agnostic to the underlying environment, tools, and domain. We show the effectiveness of this framework by (1) demonstrating how an example experiment can be described within the framework and what benefits this offers, (2) presenting a prototype implementation of the framework, and (3) offering a collection of approaches to automated experimentation that the framework accommodates.

Contents

List of Figures

Acknowledgements

My parents, Wayne and Lisa Stoner, have pushed me to challenge myself throughout my entire academic career. I simply would not be where I am today without the encouragement they have given me, year after year. My brother, Eric Stoner, has provided intellectual and emotional support in ways that only a brother can.

I must extend immense gratitude to my advisor, Dr. Marco Carvalho. The wisdom, direction, and skills you have shared with me over the last three years have been invaluable, and I look forward to continuing our work.

Many friends have been instrumental in my completion of this milestone. Ben Remy, Alec Bertossa, and Sebastian Rainer have kept me from losing my mind over the last five years. Troy Toggweiler, Adrian Granados, and Dr. Thomas Eskridge have been sounding boards as I developed my ideas, both good and bad. Dee Casale has kept my life at Florida Tech on track.

This thesis is dedicated to Richard and Joan Lawrence (a.k.a. Grampy and Grammy), whose love and advice have meant so much to me.

Chapter 1

Introduction

There are a variety of environments available to computer scientists performing research on networked systems. The resources available within these environments come in many flavors: simulated, emulated (via virtualization), and bare metal; wired and wireless; co-located and globally dispersed; powerhouses and tiny processors; and the number of attributes goes on. Scientists can choose environments that suit the needs of their unique projects, many of them freely accessible with an academic affiliation. But despite the amount of choices that researchers have in selecting an experimentation platform, they are still required to manually design, build, execute, and analyze their experiments[1]. While tools exist to aid the user in accomplishing isolated goals—such as provisioning a virtual machine, or collecting measurements—there is not yet available a tool that is capable of completing the entire experiment workflow. We find that all tools currently available are tightly coupled with at least one of the following:

1. A specific environment or testbed. For example, one tool may be great for working with resources on GENI [22], but be completely useless with DeterLab [4].

[1] In this thesis, we use the term "experiment" to refer to the empirical evaluation of a system. Our work focuses on helping a user complete such an experiment by building infrastructure, executing procedures, and collecting results.

2. A specific research domain. For example, one tool may be great for performing highly controlled cyber security experiments, but be completely useless for a student exploring network performance testing.

3. A specific segment of the experiment lifecycle. For example, one tool may be great for building resources in the environment, but be completely useless when it comes to installing software on those resources.

In this thesis, we respond to this gap in the cyber experimentation domain by proposing a framework that will support the execution of cyber experiments from design to evaluation. The framework we propose is not targeted at any one platform or process; rather, it is a foundation that provides researchers and tool developers with a simple model in which to perform experimentation. Based on the shortcomings we have identified with existing tools, we define the following requirements for our framework:

1. The framework must be opinionated in terms of experiment design. The benefit of imposing a core experiment structure is that we can then allow the internal workings of individual components in that structure to be very flexible. As an analogy, consider what interfaces are to object-oriented programming. This also provides us with a common vocabulary to use across all experiments.

2. The framework must be agnostic to environment and tooling. That is, it must not make assumptions about what resources the environment offers or what tools are capable of interacting with those resources. Hence, the framework must be plug-in based, so that it can interact with existing environments and tools.

3. The framework must be agnostic to research domain and goals. For example, the framework must be useful to researchers performing formal, well-defined, controlled experiments; and to students engaging in a more exploratory journey.

4. The framework must support repeatability. This provides a mechanism by which peers can verify one's results, and as an added benefit supports future work in experiment automation.

In order to identify a framework that would provide such a foundation to researchers, we need to identify the commonalities among diverse types of experiments, which ultimately requires either a survey of experiments or a survey of tools. The former method would allow us to assess firsthand what processes scientists are executing as they perform experiments. Under the assumption that scientists make tools that suit their needs or the needs of others, the latter method would also give us this information, and additionally give us an idea of how these scientists have abstracted experimentation. Since the tools survey appears to offer more information, and computer scientists frequently fail to document their methodologies even when required to do so [13], we choose that method.

In addition to the tools survey, we review existing work whose goals are similar to ours (i.e., identifying a flexible cyber experimentation framework) and consider the abstractions that they have developed. We find that the existing frameworks suffer from the same tight coupling problems as the tools. We use the survey of existing tools and frameworks as well as several years of experience with cyber scenario emulation to develop a new framework, as well as a prototype implementation of this framework.

The main contributions of this thesis are as follows:

- We describe the capabilities and limitations of existing experimentation tools and frameworks.
- We introduce a new framework, describe an example experiment within the context of the framework, and discuss a prototype implementation of the framework.
- We outline how future work in automated experimentation should leverage our framework.

The rest of this thesis is organized as follows: Chapter 2 contains a literature review that covers existing work on the definition of cyber experimentation, as well as experimentation tools from academia and industry. Chapter 3 analyzes these tools and finds common and unique features among them. Chapter 4 describes the framework we propose based on existing frameworks and our tools analysis. Chapter 5 outlines the workflows required to execute an example experiment with and without a unifying tool. Chapter 7 explores how this framework can be applied to automated experimentation. Chapter 8 summarizes and concludes our findings.

Chapter 2

Literature Review

In this chapter, we review cyber experimentation tools that are currently available to academic researchers, as well as open-source tools available to software developers and IT operators. We find that these tools have several similarities with their academic counterparts, and we have personally had success with some of them. For all tools, we summarize their origin, purpose, workflow, and development status. We then consider existing work on cyber experimentation frameworks and compare their assumptions and goals.

2.1 Experimentation in the field

Due to the lack of research in cyber experimentation itself, we take to investigating existing tools to determine the underlying model that drives most cyber experiments. For the sake of diversity, we don't restrict ourselves to tools developed for academics—we include popular tools from the open-source community that have parallels to academic tools.

2.1.1 Tools from academia

These tools have been developed for use with popular testbeds—such as GENI, PlanetLab [16], and DeterLab—and help with some subset of the experimentation lifecycle. Some tools overlap heavily, while others serve completely different roles. This means that a researcher will likely use a number of these tools to complete a single experiment. Some of these tools have been presented in the literature with a more philosophical look at what cyber experimentation is; this discussion is extremely valuable, but we save it for Section 2.2 where we can include additional work in this area.

Control and Management Framework (OMF)[1] [18]

OMF is a testbed control and management framework originally developed for the ORBIT wireless testbed at Rutgers University's WINLAB and is now maintained by National Information Communications Technology Australia (NICTA). Like many testbed management frameworks, it includes a Resource Broker, a Disk Image Manager, a Chassis Manager, and additional infrastructure to support the provisioning of raw resources. In addition to management resources, OMF provides mechanisms for control of the resources (the Experiment Controller and Resource Controllers) and measurement collection (OML, summarized below).

In order to interact with OMF's control plane, a Ruby-based domain-specific language called the OMF Experiment Description Language (OEDL) was developed. OEDL contains commands to describe an experiment's resources, applications, parameters, execution commands, measurements, and events. The language is flexible and can be used to build sophisticated experiments that involve variables, conditional events based on measurements (i.e., feedback-based experimentation), and complex topologies [9]. Although powerful, OEDL tightly coupled with the OMF architecture and hence can only be used with testbeds built with OMF. So,

[1] http://omf.mytestbed.net

for example, one could not use OEDL to describe an experiment to be conducted in DeterLab.

The latest version of OMF, OMF 6.0, was released March 18, 2013. The latest repository commit was July 15, 2014.

OMF Measurement Library (OML)[2] [14]

OML was originally designed specifically for the measurement tasks of OMF-enabled testbeds, but it is now capable of standing alone. The OML architecture consists of instrumented applications which serve as sources and database-backed collection points which serve as sinks. (Filters or proxy servers may come in between.) Developers using the OML library in C, Ruby, Python, Java, or JavaScript can create "measurement points" inside their applications, which produce "measurement streams" over time. In addition, a developer can wrap an existing application and report metrics that it discovers (e.g. by parsing the program's standard output stream).

OML requires no commitment from the developer ahead of time in terms of which metrics will be reported, how frequently, etc. Rather, OML's simple text-based protocol allows for on-the-fly creation of measurement schemas.

OML 2.11.0 was released May 6, 2014. Repository commits are current.

Plush[3] and Gush[4] [1]

Plush and Gush are command-line tools built by Williams College that provide a user shell ("ush") for PlanetLab and GENI ("Pl" and "G"). With these tools, the user describes in an XML format the nodes their experiment uses, as well as file archives to transfer to the nodes and commands to execute on them. The user may

[2] http://oml.mytestbed.net
[3] http://plush.cs.williams.edu/
[4] http://gush.cs.williams.edu/

also execute one-off commands on one or many resources using the provided shell. Note that Plush and Gush do *not* reserve these resources from the testbeds—it's up to the user to do this first. Plush and Gush expose much of their functionality to both a GUI called Nebula[5] and an XML-RPC API, so users of varying levels of technical ability are capable of interacting with it.

Plush and Gush are no longer maintained.

Security Experimentation Environment (SEER)[6] [20]

SEER is a security experimentation tool created by SPARTA, Inc. that provides a configurable and extensible library of benign and malicious traffic generation tools ("agents"), as well as the mechanisms to deploy them. SEER was designed to work specifically with DeterLab and allows researchers to create and run experiments via a GUI or Python scripting interface. Graphs are available per-host that show network traffic, as well as other metrics reported by "collectors."

SEER 1.6 was released March 2012, and the latest repository commit was October 14, 2014.

jFed[7]

jFed is "a Java-based framework for testbed federation" built by the iMinds research institute and Ghent University for the European Commission's federated future Internet research project, Fed4FIRE. In addition to providing APIs for federated testbeds, jFed includes an experimenter GUI and CLI that allow the user to build and control resources on a number of different testbeds. Features of the experimenter GUI include graphical topology editor, resource specification (RSpec[8]

[5] http://gush.cs.williams.edu/trac/gush/wiki/NebulaPage
[6] http://seer.deterlab.net/
[7] http://jfed.iminds.be/
[8] http://groups.geni.net/geni/wiki/GENIExperimenter/RSpecs

XML) editor, SSH console access, one-off shell command execution, and scheduled shell command execution ("timeline").

We were unable to locate the jFed source code, but the latest build of the experimenter tools was December 19, 2014.

Omni[9]

Omni is a command line client for reserving GENI resources that works with XML-based resource specification (RSpec) files [19]. Omni's Python libraries can also be imported and used in other programs. It provides no mechanisms for actually *executing* an experiment—it just provides the raw resources.

Omni is under active development.

Network Experimentation Programming Interface (NEPI)[10] [17]

The NEPI website provides a clear and concise description of its purpose: "NEPI is a Python-based library to model and run network experiments on a variety of network evaluation platforms (e.g. PlanetLab, OMF wireless testbeds, network simulators, etc). It allows [the user] to specify resources, to define an experiment workflow and to automate deployment, resource control and result collection." NEPI provides a Python API by which a researcher can define experimental resources (e.g. "LinuxNode" and "LinuxApplication,") and query their measurement points (e.g. "stdout" and "stderr" for the "LinuxApplication" resource). Since the instantiation of those resources is handled by a plugin, the researcher can use the same API between environments—as long as a plugin exists for that environment. In fact, NEPI's granular abstraction of resources (see Section 2.2) means the researcher can define a single experiment that spans multiple environments.

NEPI is under active development.

[9] http://trac.gpolab.bbn.com/gcf/wiki/Omni
[10] http://nepi.inria.fr/

MalwareLab [2]

MalwareLab is a collection of browser-based exploits and different versions of Mozilla Firefox, Microsoft Internet Explorer, Adobe Flash Player, Adobe Reader, and Java. For each exploit, 30 random configurations of the above software are automatically chosen, built in VirtualBox virtual machines, and tested with Selenium, a browser automation tool. Although MalwareLab itself does not provide a generic platform for experimentation, it does serve as a good example of how one group is performing automated experimentation.

MalwareLab was presented August 12, 2013, and it is not apparent that work has continued on it since then.

2.1.2 Tools from industry

The tools here perform similar tasks to those mentioned in Section 2.1.1, except they are aimed at developers, testers, and system administrators. Some handle resources provided by Amazon Web Services[11], OpenStack[12], and other "*-as-a-service" vendors. Others execute test suites against a number of autonomously provisioned environments.

Terraform[13]

Terraform is a tool developed by Hashicorp for "building, changing, and versioning infrastructure." From a single configuration file, a user can describe his infrastructure in terms of physical and virtual servers, DNS records, email applications—or any other resource defined by the available providers. When the user executes Terraform, an execution plan is built that dictates what resources will be destroyed, changed, or built, and in what order to respect inter-resource dependencies. For

[11] http://aws.amazon.com/
[12] http://www.openstack.org/
[13] https://terraform.io/

example, if node A is on network B, network B will be built before node A, but in parallel to some other network C.

Terraform is under active development.

Ansible[14]

Ansible is a agentless IT automation tool that allows users to describe the configuration of hosts, with the only requirement that the host supports SSH connections and has Python 2.4 installed. Ansible focuses on reaching a desired state (e.g. package `apache2` is installed) versus the blind execution of a task (e.g. install package `apache2`) as a shell script would, making it safe to rerun "playbooks" multiple times without a negative impact—this concept is referred to as "idempotence." Ansible contains hundreds of modules which facilitate the management of users, databases, packages, files, services, etc. and users are free to develop their own.

Ansible is under active development.

tox[15] and dox[16]

Virtual environments provide Python developers a way of isolating projects and their dependencies so that they don't conflict with each other (e.g. project X requires version 1.2 of a library, and project Y requires version 1.3). tox is a tool for automatically building virtual environments and running tests inside them so developers can check that their software functions properly across libraries, interpreters, and configurations. dox is a tool inspired by tox that relies on Docker[17] to build Linux containers where the tests run. This makes tests runs even further independent of the machine they are being hosted on, which provides consistency between all build

[14] http://www.ansible.com/
[15] http://tox.testrun.org/
[16] https://git.openstack.org/cgit/stackforge/dox
[17] https://www.docker.com/

environments. Both tox and dox can report test results to the developer and to a continuous integration service, such as Jenkins[18].

tox and dox are both under active development.

Travis CI[19]

Travis CI is a continuous integration service that builds code with a variety of runtimes, dependencies, and environment variables. When performing a build, Travis CI creates a build matrix from the configuration file—`.travis.yml`—that simply computes the cross product of every variation defined. For example, specifying two Python runtime versions and two versions of a library would result in four distinct builds. These builds run either in a VM or a Docker container with standard images, which the user is only able to configure via shell commands listed in `.travis.yml`.

Travis CI is under active development.

Heat[20]

Heat is a subproject of the OpenStack cloud platform that builds infrastructure based on a single description file. In OpenStack, this means the ability to create software-defined networks and virtual machines, among other resources, with one action. Unlike some other tools mentioned here, Heat does not seek a goal state; rather, it enables the user to build the template many times to facilitate elastic application stacks.

Heat is under active development.

[18] https://jenkins-ci.org/
[19] https://travis-ci.org/
[20] https://wiki.openstack.org/wiki/Heat

2.2 Abstracting cyber experimentation

Many researchers who develop experimentation tools, such as those summarized above, began their work by trying to identify a model of cyber experimentation. The developers of SEER identified the "building blocks" of a security experiment as [20]:

- Network infrastructure configuration
- Generation of background traffic and activity
- Simulation of attacks
- Deployment and management of defense mechanisms
- Instrumentation and data collection
- Data analysis and visualization

NEPI takes a more generic—albeit highly granular—approach to describing experiments which is based on "boxes and connectors." Every hardware and software component in the experiment is a box, which may have attributes, and may be connected to another box via a connector. For example, a Linux host box may be connected to a network interface box, which has attributes that describe its IPv4 configuration. Additionally, a Linux application box could be connected to the Linux host. With this level of abstraction, researchers can develop experiments that span environments, such as in the case study where the authors use elements from both the netns emulator and the ns-3 simulator to construct an emulated wireless video streaming experiment [17].

Lockheed Martin has developed the Cyber Scientific Test Language (CSTL) as part of its work on the National Cyber Range [8]. CSTL is an ontology-based language for the rigorous definition of cyber experiments. The major components of CSTL cover:

- Experiment design: a scientific description of the experiment that includes hypotheses, treatments, and variables
- Test planning and administration: scheduling and sponsorship
- Network descriptions: the topology of the experiment
- Asset descriptions: the hardware and software necessary to execute the experiment
- Test execution: execution of steps via a separate Event Execution Language, event timing infrastructure, and network traffic generation
- Data analysis and visualization: instrumentation and measurement collection
- Templates: reuse of experiment descriptions

CSTL provides a rich language for asset description, which enables the experiment designer to select the types of hardware and software his experiment requires [15]. Although CSTL enables the researcher to write very precisely defined experiments, it also supports sparse descriptions. The authors note that sparse descriptions allow for more flexibility in range scheduling and application of business rules, since the range operator is able to assign sufficient substitute resources.

Along the same lines of CSTL, Westermann et al. introduce an experiment specification language for system performance experiments, which supports automated execution and is independent of underlying technologies [23]. The language supports input and observation parameters, automated assignment of values to input parameters (e.g. given a range to choose from), exploration strategies (i.e. how to combine input parameters), and termination conditions. Like our inquiry, the authors of this language aren't trying to build tools that perform a specific task—they are solely trying to identify a model that suits a variety of applications. And, once it's defined, extensible tools can be developed around that model. Because this language is for performance testing, it assumes the existence of target nodes, and thus has no mechanism for creating arbitrary experiment resources. In other words,

the language and its corresponding tool [24] assume the experiment can be run on a set of nodes which it already knows about because it is just the configuration of the software running on that node that is of significance, as opposed to some attribute of the node itself.

Finally, the Virtual Platform for Network Experimentation (VIPE) seeks to abstract network protocol experimentation [11]. VIPE provides APIs to a virtual platform that expresses the core components of every platform's networking stack: packet control, memory management, timing, synchronization, and network devices. By specifying a target platform, such as Linux Kernel or Windows XP Userspace, the protocol can be built with platform-specific code. The virtual platform that VIPE provides enables researchers to quickly build code for multiple platform and bypass the porting effort. Although the type of experimentation that VIPE facilitates is different from what we are trying to accomplish, the authors' cross-platform flexibility is notable to us.

Chapter 3

Tools Analysis

In this chapter, we reduce the tools from Section 2.1 to a set of common features. In Chapter 4, we'll combine this analysis with the previous work on cyber experiment abstraction (section 2.2) to develop a generic framework for cyber experimentation.

3.1 Resource creation or reservation

The most common feature available in these tools is resource creation or reservation (depending on whether the resources are virtual or fixed). This is an obvious feature, since the resources provide the foundation of the experiment—it's where the experiment, whatever its nature, will be executed. In some experiments, the resource itself has some property that makes it important. For example, the researcher may require two nodes to have wireless network interfaces, or may want to select nodes in specific geographic locations. In these cases, the experimenter will specify the requirements of the resources in some format that the environment understands, such as an XML-based resource specification (RSpec) with GENI [19] or ontologically with the NCR [15]. In most cases, the resource broker in the environment understands sparse descriptions, i.e. it need not have every attribute of the resource defined. Instead, it queries available resources and selects those that

match the requirements the user specifies. This is the difference between a request like "reserve resource X" and "reserve a resource with two processors and at least 4096 MB memory."

In other environments, primarily where the resources are virtualized, the user is capable of requesting *any* attribute value and getting it. Consider a cloud provider like Rackspace: customers can select the hardware capabilities of their virtual machines with "flavors." In general, all flavors are always available, because it makes no difference to Rackspace whether the customer requests two virtual CPUs or four—this is one of the building blocks of elastic cloud computing. When working with tools like Terraform or Heat, this type of environment enables the resource specification to be less of a *request* for resources and more of a *declaration* of what resources are necessary.

Regardless of whether the environment is made up of static or virtualized resources, or even if it's heterogeneous, the resource creation/reservation step is almost always the first step in executing an experiment. It is important that our framework is agnostic to the actual workflow of allocating resources to an experiment and instead just provides a common, compatible language for doing so. This way, the framework can be used to perform experiments across environments, including those with the allocation workflows we define here, and others.

3.2 Scripted or live shell control

With resources allocated to an experiment, the researcher must provision them in some meaningful way. The de facto option here tends to be raw control of the shell via an SSH connection—this is the case with GENI[1] and DeterLab[2], for example. The RSpec format allows startup commands to be embedded in the resource request, while tools like Gush allow the user to control the resources interactively. In addition, jFed allows shell commands to be placed on a timeline

[1] http://groups.geni.net/geni/wiki/HowTo/LoginToNodes
[2] https://trac.deterlab.net/wiki/DETERSSH

so that they will be executed automatically after the experiment has been running for the specified time. Typically, we don't see the use of industry configuration management tools like Ansible or Chef[3] to provision nodes on academic testbeds, although we have seen this recommended when performing experiments on cloud environments, where they are noted for their ability to consistently provision the experimental resources [7]. We're unsure why academics don't take advantage of these popular, stable, well-documented tools to provision their experimental nodes more frequently, but this trend will have to be investigated outside of this thesis.

3.3 Instrumentation and measurement collection

Most experiments probably require some measurement collection in order to validate some theory. This can range from a simple success/fail (MalwareLab, TestREx [6], dox/tox) to log files and process output (Ansible, NEPI, jFed) to packet-level graphs (SEER). In addition, tools are available to support the instrumentation of any application, especially those developed by the researcher (OML). There isn't much overlap in the types of measurements that each tool can collect (e.g. Ansible cannot collect packet captures), which means that collecting a different type of measurement requires learning a different tool. We see again the theme mentioned in Chapter 1 that although researchers have many tools available to them, they are faced with a significant learning curve when they want to design different types of experiments.

3.4 Experiment execution

Experiment execution means different things in different tools, but we mean the execution of experimental tasks once the resources have been built. In some cases, this encapsulates shell control and measurement, but not always. In most tools,

[3] https://www.chef.io/chef/

we only see primitive examples of execution: startup commands or interactive shell sessions. jFed gives us time-based control, which is slightly more interesting but still not very sophisticated in that it doesn't take into account any information about the experiment before executing the commands. The only tool we have identified that provides advanced execution controls is OMF's OEDL, particularly because of its integration with OML which allows the researcher to build an experiment that is feedback-dependent. It also provides native support for custom event triggering. In addition, because OEDL is a Ruby-based language, all the flow control and libraries available in Ruby are present in OEDL, which could allow an experimenter to build an experiment that reaches far beyond the typical set of controls offered by OMF. NEPI provides similar capabilities because it (1) has its "traces" that provide output and (2) is a Python library. Note that based on our observations, it appears that to build a reasonably complex experiment, the experimenter must be a confident programmer *and* have the time to program an experiment. This is unfortunate because the experiment's implementation is probably much less important to the researcher than its design.

Chapter 4

Our Framework

In this chapter, we outline our proposed cyber experimentation framework. In order to fulfill all four of our requirements from Chapter 1, the framework is organized into five loosely coupled components. As shown in Figure 4.1, each component is at most only aware of its neighboring components, meaning that the impact of changes to the experiment are localized. In addition, this separation of concerns also means that changing major components of the experiment, such as the environment where the experiment is run or the type of analysis performed, can be switched seamlessly or with relative ease. The framework *itself* does not interact with any environments or tools (requirements 2 and 3). Rather, the framework communicates with drivers or plugins which in turn communicate with the environments and tools. As mentioned briefly in Chapter 1, by setting expectations for the overall design of the experiment (requirement 1), we allow these plugins to behave internally however they see fit.

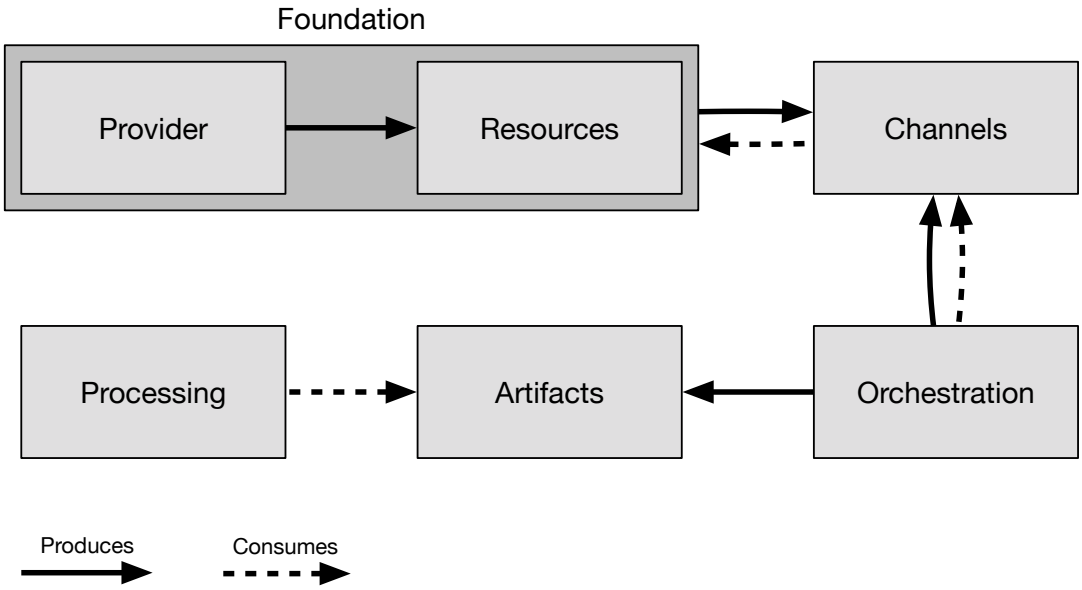

Figure 4.1: A high-level view of the framework's components and the production-consumption relationships between them. The role of an experimentation tool that implements our framework is explained in Section 4.2 and shown in Figure 4.6.

4.1 Components

4.1.1 Foundation: provider and resources

The most common feature that current tools give researchers is the ability to create raw resources in the environment. We use the terms *resource* and *provider* to refer to individual experimental entities and the environments that allow us to create them, respectively (see Figure 4.2). In addition, we group providers and resources into the *foundation* component, so called because it represents the most essential building blocks of an experiment. OpenStack is an example of a provider, as well as DeterLab, GENI, VirtualBox, and ns-2[1]. The resources OpenStack makes available to a user include images, virtual machines, and virtual networks. Other providers may make the same resources available, add others, or lack some. We

[1] http://nsnam.isi.edu/nsnam/index.php/Main_Page

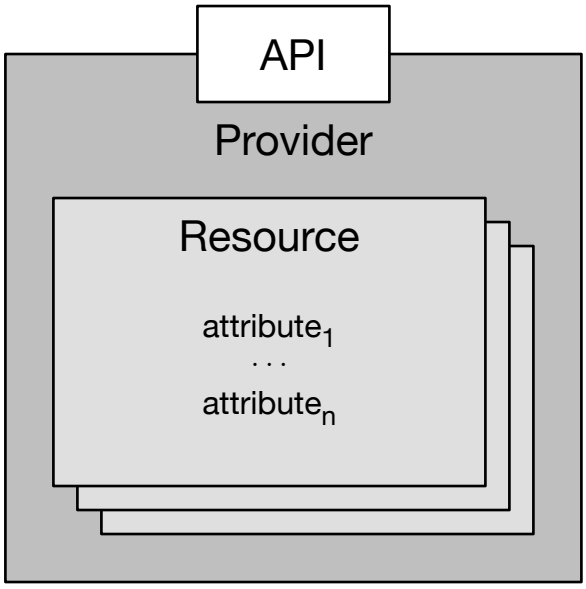

Figure 4.2: The experiment foundation consists of a provider who manages many resources. Every resource can have a number of attributes, and the provider exposes an API for communication.

don't mandate the existence of certain resources, such as a node, because doing so would make assumptions about the environment, violating Requirement 2 (environment agnosticism). However, there would be clear advantages with regard to Requirement 4 (repeatability) if providers shared resource types.

Continuing with support for Requirement 2, we acknowledge that certain providers will require certain information about resources in order to build them. For example, a virtual machine will require a disk image, a network will require an IP address space, etc. We allow all components of our framework to have arbitrary attributes specified so that we don't promote or restrict the use of any one environment, tool, or method. For example, even though OpenStack and VirtualBox may both offer a virtual machine, OpenStack will require an image, flavor, and network while VirtualBox requires only an image.

Rather than including a special segment of the foundation for network topologies, we let resources implicitly support the creation of network topologies to the extent

that the provider does. That is, it's up to the provider to make resources like subnets and links available. Since network topology support varies greatly across providers, we cannot build any assumptions about topology into the framework. As is the case with all resources, network resources can have additional properties set that help the provider build the appropriate devices, such as link speed or packet loss rate. Regardless of provider support, designating network topologies as an inherent part of the experiment would violate Requirements 2 and 3, so we do not include it for the same reason we do not specify any resource types. (For further justification, see Section 4.1.6.)

4.1.2 Channels

On their own, experimental resources don't serve much purpose—they are black boxes with no entry or exit point. The foundation is responsible for specifying *channels* that allow communication with the resources, as shown in Figure 4.3. The most common channel for a Linux-based resource would be SSH, but there could be other dedicated control channels such as the APIs offered by applications that live on the resource. For example, SEER's traffic generation agents expose an API which would constitute a channel. In addition, channels may build on top of each other: consider Ansible, which is capable of performing advanced provisioning when an SSH connection is available. The only requirement for a channel is that it can be invoked, potentially with some parameters (e.g. invoking SSH with a command to execute or invoking a REST API with a URL and request body). This is the simple interface that is exposed to the user so that new tools can be plugged in in a familiar way, rather than the researcher having to learn a new tool and figure out how to implement it in his experiment. With such a design, it's possible to rapidly build experiments using new tools. Ansible has seen great success with this pattern, where a user invokes some module with a set of parameters, because the learning curve is reduced to a 60-second process of reading a one-page document that describes each parameter[2].

[2]For an example, see http://docs.ansible.com/file_module.html.

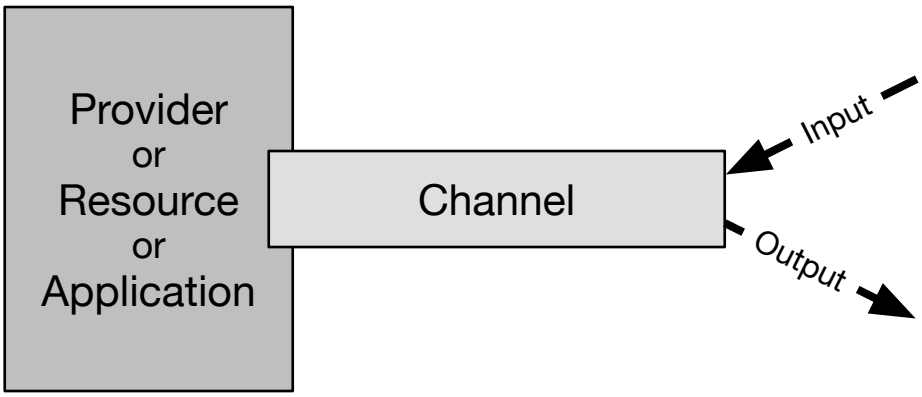

Figure 4.3: Channels provide a medium of communication between the provider, a resource, or an application living on a resource. Channels are invoked, possibly with input parameters, and may provide output.

The foundation as a whole is responsible for specifying which channels are available. This means that both the provider and the resources may accept communication from the experiment. We've already mentioned some resource channels, but provider channels may be useful for performing operations outside the resource. These channels support some "god-like" control over the experiment which the researcher can use to emulate experimental conditions, such as a node unexpectedly losing power (a power control channel) or limiting access to network and internet resources (a firewall channel). In addition, provider channels can be used to instrument the experiment in ways that the resources themselves can't facilitate: a snapshotting channel could be used to capture the state of a node's disk during experimentation, and a monitoring channel could be used to report a resource's CPU, memory, or bandwidth consumption.

Note that the foundation is only responsible for *listing* which channels are available, not *implementing* them; this speaks to the separation of concerns that grounds this framework. In general, channels should be developed independently of the providers, so that we keep the loose coupling between components and support switching providers as easily as possible. In fact, using specialized or provider-specific channels negatively affects the framework's loose coupling goal, and means

that changes to the foundation will traverse the framework more deeply than if the channels used were generic. We advocate the use of abstract provider channels even where it may seem tempting to build a provider-specific channel; doing so allows the experimenter to use the same channels even while changing provider, because it's only necessary that the provider implements a common interface for performing these actions. For example, rather than creating a single VirtualBox channel that exposes every feature that VirtualBox offers, it would be better for the VirtualBox driver to interpret generic power and snapshotting channels. Then, if the user moves to VMware and the VMware driver also supports the generic channels, there are no changes to make to the experiment.

In addition to foundation channels, the experimentation tool that implements this framework may offer one or more meta channels which provide experimental controls. This channel could provide for example the ability to stop an experiment or, in the case of automated experimentation, add some values to be tested. Automated experimentation is discussed in more detail in Chapter 7. These meta channels do not need to be generic like foundation channels, since by their nature they are specific to the experimentation tool.

4.1.3 Orchestration

Channels just represent a line of communication where the user can control the experiment foundation (or the experiment itself with meta channels); we need *orchestration* to take advantage of these channels. Orchestration encompasses setup, execution, and results collection—essentially anything that happens while the resources are instantiated. The tasks executed during orchestration could be fairly primitive, such as deploying a program and executing it as soon as the resources become available; or performing some type of modifications as the experiment is running. In addition, the orchestration component is allowed to add more channels to the experiment; this allows relatively unconfigured resources to be bootstrapped with control software, then registered with the appropriate channel. Such would be

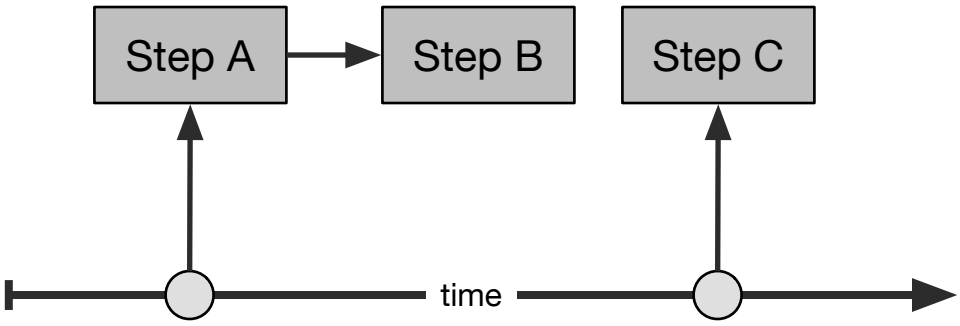

Figure 4.4: Orchestration steps may be fired by a number of methods. Here, we show time-based triggering (steps A and C fired at a specific time) and dependency-based triggering (step B fired by step A).

the case if the Ansible channel was added to a node whose only channel by default is SSH.

As with the other components, the exact implementation is completely plugin-based and completely opaque to the framework, so the experiment can be run in whatever fashion makes sense to the researcher. All orchestration plugins interact with this component by firing orchestration *steps*. When a step fires, it invokes a channel with parameters specific to that channel, provided by the user (see previous section for examples). An experiment can be orchestrated by mixing multiple plugins, as shown in Figure 4.4.

It's worth reiterating that orchestration can take any form. On one hand, this allows the developer of a large, well-defined experiment to set up a variety of tasks to automatically execute on a variety of queues. A non-interactive orchestration layer enables the experiment to be run repeatedly, either using an identical configuration or by varying some parameter, and without any knowledge of what inputs the experiment expects. On the other hand, a fully-interactive orchestration layer gives the experimenter full control and visibility of the experiment execution and is useful when the task at hand is exploratory in nature. It would even be possible to build a hybrid orchestration layer, where a number of steps are automated but a human is still in the loop.

The orchestration steps know which resources exist, but they don't know anything about those resources, except which channels are available on those resources. Consider the benefit that this has to the experimenter when changing environment vertically (e.g. virtual machines to physical machines) or horizontally (e.g. GENI to DeterLab). Provided that the environments offer the same channel, such as SSH, the remaining components of the experiment will remain intact, including the orchestration layer which, based on our experience, has the potential to be one of the most complex components. This has major implications on reproducibility because it enables peers to recreate the experiment (requirement 4) even if they don't have access to the specific environment that the original experimenter used.

4.1.4 Artifacts

As mentioned briefly before, the orchestration layer is also responsible for collecting results. We use the term *artifacts* to refer to any information that the orchestration steps collect and preserve. Artifacts are simply information collected and saved from or about the resources or provider. We are immediately aware of four types of artifacts, although more might exist: (1) Resource artifacts are those that we collect from the resource that are independent of the experimentation framework. Log files or packet capture files fall under this category. (2) Provider artifacts are those generated by the provider. These artifacts will vary greatly between providers, but consider a provider that's capable of returning the entire virtual disk of a resource. (3) Channel artifacts are the response of a resource when a channel is invoked on it. The standard output stream for an SSH channel or the webpage downloaded from an HTTP channel are channel artifacts. (4) Meta artifacts are those generated by the channel or the framework that describe the invocation of the channel, but not the response from the resource. The time elapsed during the invocation of a channel and whether that invocation was successful are meta artifacts. Meta artifacts are useful for debugging and also provide ground truth data to experiments that need such information (e.g. when an attack was started versus when it was detected).

In reality, it's the implementation of the channel that dictates what kinds of artifacts can be procured from the experiment. These could be flat text files (logs), binary files (packet captures or executables), or some other structured result. The experimentation tool should be capable of storing artifacts in a number of formats, whether this is accomplished in the filesystem, a database, or some other store. In this work, we don't try to enumerate all of the types of artifacts that channels may produce; keeping with our theme, we don't dictate any requirements about the internal implementation of a framework component. The only requirement for artifacts is based on the expectations of the processing plugins that will operate on them, shown in Figure 4.5 and discussed further in the following section.

4.1.5 Processing

Processing represents any type of operation on the artifacts. The types of processing conducted are completely up to the researcher, and only include those steps that make sense given the goals of the experiment. For example, the extent of processing in one experiment may be to forward log files to a Logstash[3] and Elasticsearch[4] stack, where they can be indexed, searched, and visualized more easily than they could from flat files. Another experiment may perform some statistical analyses on measurements collected from the resources, while another evaluates a model against the experimental data. In an implementation of automated experimentation, processing may schedule more experiments (see Chapter 7). Finally, processing could also be "manual," where the researcher would be free to browse the collection of artifacts as they were stored by the tool.

Processing plugins execute on artifacts with no knowledge of their origin (unless the artifact is accompanied by some metadata). Specifically, the plugin wouldn't know the provider, resource, channel, or orchestration step that produced this artifact. Again, we see the framework's loose coupling at play: the processing component is fully content with other components' implementations being changed. In addition

[3]https://www.elastic.co/products/logstash/
[4]https://www.elastic.co/products/elasticsearch/

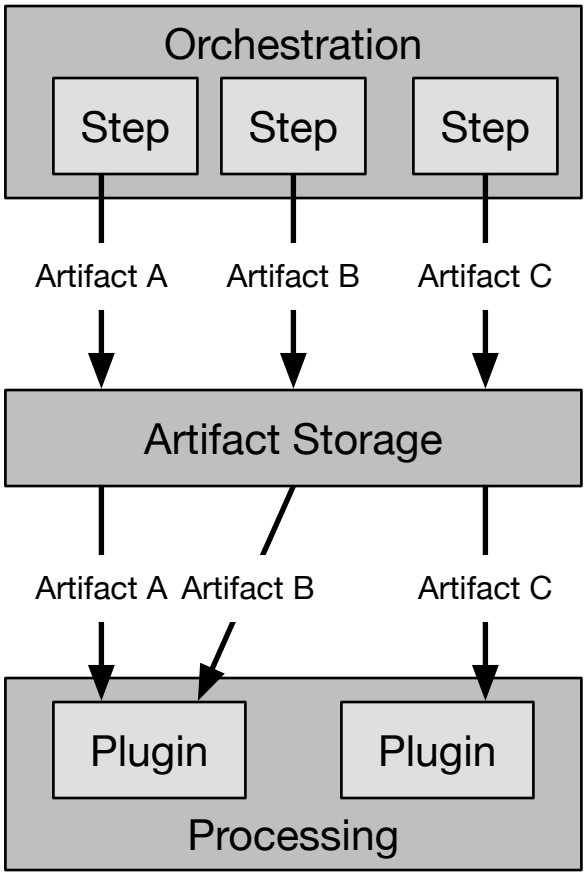

Figure 4.5: Orchestration steps produce various artifacts, which are stored by the experimentation tool during the execution of the experiment. Once orchestration ends, processing plugins can request the artifacts from storage.

to the ability to change the provider or resources that we've already described, the channels and orchestration used in the experiment can also be changed without impact on processing, as long as the same artifacts are produced. In an experiment that emulates attack traffic, the attacks could be launched on a simple interval basis in one experiment, then by a more advanced method in another. With the same artifacts being collected, the processing would be identical, and the two experiments could be compared immediately.

We considered allowing plugins of the processing component to produce artifacts, but ultimately decided that this decision would degrade our framework in two ways:

(1) Most importantly, this link would violate Requirement 2 because it would make assumptions about the types of processing being conducted. Namely, it would imply that processing necessarily includes some sort of artifact transformation, and we've already stated that this is *not* the limit of processing in our framework. (2) This link would make artifacts more difficult to define because it would segregate the artifact collection into two classes, raw and transformed, and we preferred to keep artifacts strictly driven by the execution of the experiment. For now, storage of processing results must be handled by the plugin.

Processing doesn't have to take place after the experiment concludes, although this is probably the most common approach. In some cases, it may be desirable to perform processing as the experiment is running, for example if the experiment is long-running or if it is feedback-driven. We discuss experiment lifecycle further in Section 4.2.

4.1.6 Other considerations

Some readers may be wondering where certain aspects of cyber experimentation that they are familiar with fit in our framework. We identify some of these below.

In shared environments, it is often necessary to reserve resources for use. We don't include this in our framework as it would mean we are making assumptions about the environment. If the environment requires scheduling, it is up to the provider driver to perform the proper checks. The implementing tool should allow the provider driver to postpone an experiment—or simply pause it—until the resources are available.

Software does not have a special place in our framework, as it does in others like CSTL [8]. Citing Requirements 2 and 3, we claim that this would represent an unwarranted assumption about the capabilities of the resources, the tools being used to manage them, and the domain of the experiment being conducted. Potentially, software resources could be modeled as part of the experiment foundation, but the details of such an implementation are outside the scope of this thesis. Without classifying software as part of the foundation, it would be up to the experimenter

to organize software in a way that suits his experiment, for example on the local filesystem so that it can be copied to the remote resource, or on a server where the remote resources can download it.

Our framework doesn't provide any built-in language for experiment instrumentation (NEPI's "traces," SEER's "collectors," OML's "measurement points") for the same reasons it doesn't include a designated place for software. It is up to the experimenter to use whatever tools he likes to instrument the experiment, then use the proper channels to collect the results.

We do not include any core resource types, such as a node or network, although it may seem senseless to do so right now. While our discussions thus far have been focused solely on experiments consisting of emulated computer networks, this is only one application of our framework. Consider that we could also use the framework to describe an experiment where the provider is a multi-agent system and each resource is a software agent.

4.2 Lifecycle

Although we hope that the lifecycle of an experiment designed within our framework is intuitive, we explicitly state our expectations in this section. We assume the experiment has been specified using some interface that the framework-implementing tool defines, whether that be a graphical user interface or simply an API. Besides the discussion of our prototype in Chapter 6, the nature of this interface is outside the scope of this thesis.

The first step of the experiment is building the resources on the provider. The provider driver is passed the resource types and attributes that the user defines and is responsible for building them as specified. The driver may block here if it is waiting for resources to become available, or it may fail depending on implementation and configuration. When a resource is built, the provider informs the tool that the

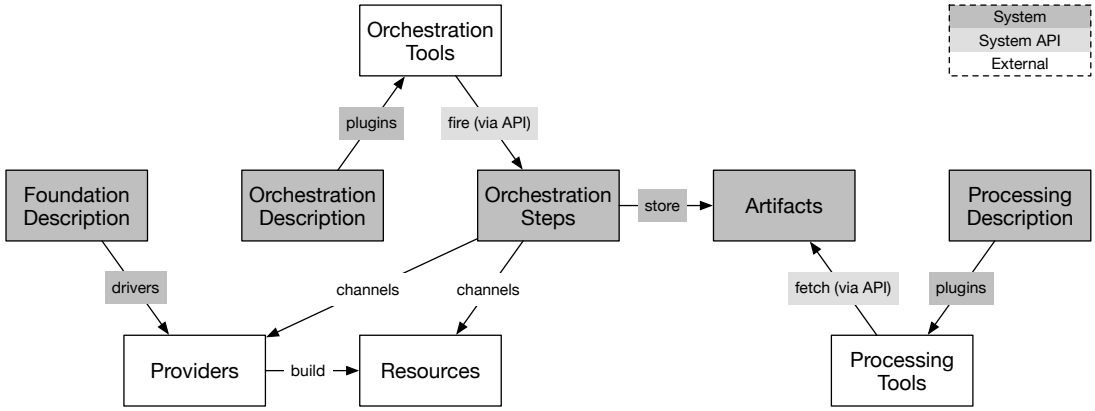

Figure 4.6: Some aspects of experimentation are inherently part of the experimentation system (dark shading), while others are completely independent of it (no shading). External components can interact with the system via public APIs (light shading). In this diagram, we have collapsed channels and their plugins, since these plugins will often only be lightweight wrappers.

resource is instantiated and which channels are capable of interacting with it. The provider must also inform the tool which provider channels it makes available.

Once all resources are instantiated, the orchestration component is invoked[5]. The exact execution of the experiment will vary depending on which orchestration plugins are used. As the orchestration steps are performed, artifacts will be generated and stored. The tool may also provide mechanisms to invoke the processing component so that processing can be performed as the experiment is executing. Such a mechanism would be useful in long-lived experiments where the researcher doesn't want to wait until the experiment concludes to get feedback.

After some time, the experiment will end due to a request from the orchestration component (or by a timeout, if the tool chooses to implement such a feature). At this time, the resources will be destroyed and the processing component will be invoked. Some experimenters may be uncomfortable with the idea of losing their

[5]It would be a reasonable expectation that the tool is capable of performing some orchestration before *all* resources become active. For example when performing basic configuration on resource R, there is no reason to wait for resource S to come up. We don't explore in this thesis how an experimentation tool could implement such a workflow.

resources after the experiment concludes, but we argue that everything of importance should be collected as artifacts. In addition, it's unlikely that there are an infinite number of resources available, especially on a shared environment—at some point, the resources *will* have to be destroyed. The tool may provide configuration options to alter this behavior, but doing so diverges from our expectations.

Once the experiment concludes, it's up to the experimenter to take the steps that make sense to him. This likely includes looking at the results of the processing steps and/or looking at the artifacts. After this, the experimenter may repeat the experiment, either identically or with some variation. Requirement 4 states that this capability *must* be available to the experimenter, whether he needs it or not. Also note that the experimenter may not be human; another system built on top of the basic experimentation tool may make additional experimentation plans automatically after this experiment completes.

Chapter 5

Illustrative Scenario

In this chapter, we describe an experiment and outline the workflow required to execute it, first with the current state of the art, and next with the tool that we are proposing.

5.1 Experiment design

We hope to solidify our framework to the reader by describing how a realistic experiment might be constructed. Please note that the resource types, attributes, channels, and plugins we reference in this example are fictitious; we do feel however that they represent reasonable possibilities as we move forward with implementing this framework. We use the language of the framework in this section to describe the design of the experiment at a high level, then define how those components can be realized in the following two sections.

Consider an experiment that measures the effectiveness of a moving target defense that varies the webserver that handles HTTP requests: a router will forward the requests to either a server running Apache[1] or one running nginx[2]. Doing so alters

[1] http://httpd.apache.org/
[2] http://nginx.org/

the attack surface exposed to an adversary and should thwart some types of attacks (e.g. an attack that works on Apache probably won't work on nginx). In this experiment, an attacker program is written that generates attack traffic on a fixed interval, say every 30 seconds. A benign client makes HTTP requests to the router every 10 seconds. The router switches the backing server it uses on a fixed interval, 90 seconds.

Resources

This experiment requires two network resources: one private where the webservers will live, and one public where the attacker and client will live. Next, this experiment requires two nodes for the webservers, one for the router, one for the attacker, and one for the client. These resources have the following properties:

resource-name: public-net (resource-type: network) cidr=10.10.0.0/16

private-net (network) cidr=192.168.1.0/24

apache (node) image=ubuntu-server, networks=private-net

nginx (node) image=ubuntu-server, networks=private-net

router (node) image=ubuntu-server, networks=private-net,public-net

attacker (node) image=ubuntu-server, networks=public-net

client (node) image=ubuntu-server, networks=public-net

We don't define IP addresses for the nodes here because we assume our provider does this automatically. These IP addresses, and potentially other properties of the resources, will be discoverable during orchestration so that the nodes can be configured properly. Terraform refers to these types of properties as "computed properties[3]."

[3]https://terraform.io/intro/getting-started/build.html

Channels

Assume the provider offers the SSH and SCP channels to node resources. In this example, we'll assume public-private key authentication is used, that the provider configures the nodes automatically with the public key, and that the private key resides on the host where the experimentation tool is running. Since the nodes support SSH, we can declare that they also support Ansible. The network resources offer no channels.

Orchestration

The following steps will be performed once the resources are instantiated:

1. Using Ansible, install the appropriate software (webserver, moving target defense, attacker, or client) on each resource.
2. Every 30 seconds, using SSH, run a command that generates attack traffic. Every 10 seconds, using SSH, run a command that generates benign HTTP traffic. Every 90 seconds, using SSH, run a command that switches the backing webserver.
3. After 30 minutes, using SCP, copy the attacker's and client's logs and store them.
4. After the logs are collected, end the experiment.

Artifacts and processing

We are collecting the attacker's and client's log files, which contain every attack and request executed and their outcome. With these artifacts, we can determine the attacker's performance and the client's quality of service. We may also want produce visualizations of this experiment with Kibana, so we will have to send the

logs to Logstash. Given the results of the experiment, the experimenter (human or software) may decide to run variations on the experiment.

5.2 Traditional workflow

The first step in running this experiment *without* a unifying tool would be to set up the foundation. Let's say the experiment will be conducted on the experimenter's workstation with VirtualBox. He would have to do this manually through the VirtualBox GUI, unless he created a program or script to do this for him. Automating the creation requires the experimenter to take some time—perhaps a day or two—to learn the VirtualBox API, write the software, and test it. Even if this program is effective, it is specific to VirtualBox, and its utility is lost if the experiment is to be run on a different provider.

With the resources created, the experimenter needs to install the appropriate software. Note first that even if the creation of resources was automated, the experimenter must still watch the process and wait for it to finish before manually triggering resource configuration. The experimenter may have the configuration steps prepared as Ansible playbooks or scripts, both of which he has to run interactively. In a more painful version of this scenario, the experimenter has elected to perform this configuration by hand, which is infeasible at scale, inconsistent, and only documented as well as the experimenter chooses to write.

Considering now that the resources have been provisioned with all the necessary software, it's time to execute the experiment. Since we have shown in Chapter 3 that there are virtually no experiment orchestration tools available, the experimenter has three options: watch a timer and perform each task manually; rely on a system tool like `cron` or `at` to perform each step; or build the capability into each tool. Although the latter approach seems reasonable, it's unachievable if the tool is not written by the experimenter (for example, if using a suite of attacks). In addition, basic timer-based execution may not satisfy future, more advanced experiments.

All options in this phase seem insufficient for one reason or another, and in our experience this *is* the most difficult part of experimentation.

After the experiment concludes, the experimenter must collect the attacker and client log files. Like the setup step, this could be accomplished with Ansible, a script, or manually. To find the success rate of the attacker and client, the experimenter would have to write a tool that counts the number of lines containing each outcome and execute it himself, passing in the location where he has stored the logs (which he hopefully does not forget if performing the experiment over several days). To graph these results in Kibana, the researcher would have to learn how to pipe them to Logstash.

If the experimenter decides to change a parameter of the experiment, he must make changes to configuration files via Ansible or by hand, and repeat each step again. Even if the experimenter has created custom automation scripts, he must watch each step execute so that he can run the next. If the experimenter didn't write automation scripts and there were manually performed steps in the experiment, repetition becomes a non-trivial task.

Consider the effect of wanting to scale the experiment up on an OpenStack cloud. First, the experimenter must determine how to build resources on this provider. If the experimenter has resource creation scripts and wants to reuse them, he will have to learn a brand new API, and again write and test his tools. If the experimenter does not have such scripts, building the experiment by hand will quickly become infeasible at scale. The same is true of any other steps that are performed manually (e.g. installing software or collecting logs). On OpenStack, the experimenter may use Heat to describe his resources, but this again requires taking the time to learn a provider-specific tool.

Overall, the existing experimentation process is time-consuming and error-prone. Experimenters waste time building ad-hoc capabilities like resource automation and timing controls instead of focusing on their research and the system under test. Given this description of the steps required to produce a very simple experiment, it's clear why the state of cyber science is so grim.

5.3 Proposed workflow

The situation is better with our proposed tool from the very beginning. Instead of setting up the resources manually or learning a new API, the experimenter describes the resources in a format defined by the tool that is consistent across providers. The experimenter is only responsible for choosing appropriate resource types and attribute values, which would be documented with the driver for that provider. Then, the tool works with the driver to realize those resources. Once the resources are available, setup is mostly the same with or without our tool: Ansible playbooks or scripts are executed that target the resources. The difference though is that these tasks are run automatically after the resources are available—no human is required in the loop.

Perhaps the largest improvement that the tool delivers is in the experiment execution. Instead of writing custom tools to schedule experiment steps, the capabilities are already present in the form of orchestration plugins. Like the description of the foundation, all of these orchestration capabilities are available via a common interface that the tool provides. This enables the researcher to build the experiment he has designed quickly, rather than spending time designing an orchestration infrastructure. The same is true of processing, where existing tools (for instance, counting lines that match a regular expression, or forwarding logs to Logstash) can be applied to an experiment by simply including them in the experiment description.

When it comes time to vary the experiment, the process is as simple as modifying the experiment description. This could be changing some variable passed to a channel (e.g. Ansible) or orchestration plugin (e.g. event timing), the attribute of a resource (e.g. operating system), or something else. The tool doesn't care what has been changed and will happily execute the modified experiment. In fact, if the infrastructure is available, the tool should be happy to execute multiple experiments in parallel. This stems from the fact that no human interaction is necessary to run an experiment once it has been described.

To scale to OpenStack, the impact would be as small as changing the foundation description to align with OpenStack's resource types and attributes. Since SSH and Ansible are available in both providers, orchestration and the rest of the experiment remain unchanged.

Our proposed approach provides three main advantages over the current state of the art:

1. An experiment can be run without human interaction. This saves the researcher from having to babysit his experiment, and also enables automated experimentation.

2. The experimenter avoids designing and building capabilities that others have already implemented. The experimenter then only needs to focus on the design of the experiment and the system under test, rather than the supporting infrastructure.

3. The experiment can be faithfully repeated with "one click." This enables a more scientific approach to cyber experimentation in that both the researcher and his peers are able to repeat the experiment and ideally reproduce the results. This also has applications in automation.

Chapter 6

Prototype

In this chapter, we describe a prototype implementation of our framework. Although the prototype is based on a simplified interpretation of this framework and only offers a subset of the features we described in Chapter 4, it does present a good basis for discussion about the advantages of this framework.

6.1 Features

The Virtual Infrastructure for Network Emulation (VINE) is a web-based application that allows experimenters to create large and complex network topologies using virtual machines and software-defined networks [21]. VINE doesn't implement a build-execute-destroy architecture like we propose in the framework; rather, the user creates resources and is able to modify and destroy them on his own accord. Users are able to interact with the VMs through an emulated console or via SSH, the latter of which VINE is also capable of leveraging without user interaction. We have developed VINE for daily use in our laboratory to develop cyber scenarios for internal research as well as collaborator requirements. Although VINE's driving purpose (construction of virtual cyber environments) doesn't overlap completely

with the framework's (flexible end-to-end specification of cyber experiments), it does follow many of the concepts that the framework defines.

Beginning with the foundation, VINE abstracts the underlying provider and its implementation. However, where VINE diverges from the framework is its assumption that the provider is capable of managing three specific types of resources: machines, networks, and network connections. Given its exact domain, that isn't an unreasonable expectation. In fact, we have successfully adapted VINE from its original provider, VirtualBox, to a cloud platform, OpenStack, with relatively low effort. Moreover, the change had a trivial impact on the VINE's users. That is, moving from one environment to a completely different one didn't mean learning new tools, languages, or operating procedures; it only included one small change to the workflow that users are accustomed to, since VirtualBox creates VMs powered off while OpenStack creates them powered on. Given the resource types VINE expects, we expect that it would be easy to adapt it to other providers, such as VMware or Linux Containers[1], and potentially GENI and PlanetLab. This speaks to the tool's ability to satisfy Requirement 2, at least when we restrict the set of environments to those that support a notion of machines, networks, and network connections.

VINE implements channels by abstracting mediums of communication, but again making a simplified assessment of what capabilities that medium should possess. *Communicators* are like the framework's channels, except that every communicator must support uploading a file to a VM, downloading a file from the VM, and executing a command on the VM. In addition, every communicator must be able to report if it can work with a specific VM. Communicators we have implemented include SSH and VirtualBox Guest Additions[2] (VBGA). When a communicator is needed, VINE is capable of automatically selecting one that works with the VM by iterating over all the communicators and checking their compatibility. This is a different approach than the framework takes, where the foundation is responsible for specifying which channels are available, and the orchestration steps request a specific channel. In future work, we may consider a way to combine these approaches,

[1] https://linuxcontainers.org/

[2] https://www.virtualbox.org/manual/ch04.html

such that orchestration steps need not select a specific channel, but rather specify a capability that a number of channels may implement. Doing so could make it easier to move between environments because the channels would be allowed to change while the orchestration remained constant, and allow for some robustness during experiment execution with regard to the ability to fall back to similar channels.

In terms of orchestration, VINE offers two options: roles (scripts that execute on startup) and assets (scripts and files that can be deployed on demand). When a user starts a VM, VINE waits for it to be available at an accessible IP address[3]. This feature is implemented as an ICMP echo request ("ping") to that host, so it succeeds as soon as the VM's networking is up. Once the machine is reachable, the process of finding a communicator begins. VINE allows up to 10 minutes to establish a medium of communication, since the backing service such as SSH or VBGA may take some time to become available after the machine is considered "reachable." After a compatible communicator is found, that communicator is used to apply all of the role commands that are associated with that machine. VINE stores the output of these commands and exposes them to the user.

Roles are designed for short commands that need to be executed every time a machine starts, such as starting a service that was intentionally not configured to start automatically[4]. There was also a need to perform more sophisticated one-off configuration steps, like copying a file archive, unarchiving it, and running the software it contained. For this, we designed assets which consist of one or more of the following: a file or directory on the VINE controller that should be copied to the VM, a pre-deployment script that is executed on the VM before the file is copied, and a post-deployment script that is run after the file is copied. With these three components the user is able to make a quick configuration change or software

[3]With the VirtualBox implementation, VINE was part of the virtual network infrastructure and as such was able to reach the VM on its user-defined experimental address. With OpenStack, VINE reaches the VM at an address on a public network that OpenStack forwards via static NAT to the experimental address with a mechanism called "floating IP addresses."

[4]In VINE, it's common for the user to create one master disk image that contains all the software and services he needs, but to leave those items dormant by default.

deployment to many VMs with a single click. As with roles, VINE saves the output of the commands executed by assets.

In addition to roles and assets, we have ongoing work in user modeling which represents a more advanced way to orchestrate traffic generation in the experiment. In brief, the agents in this system are adaptive and goal-based, and as such generate human-like network behaviors. This system isn't integrated into VINE per se, but roles and assets can be used to start behaviors on the VM—this aligns with the plugin-centric architecture of our framework. In the future, this behavior system could expose a channel to the framework so that it can be controlled more precisely from the experiment specification.

VINE's primary artifact is the output of roles and assets. However, in the VirtualBox-based version of VINE, additional ground truth data was available. The design of the networking infrastructure provided VINE with a virtual interface on every experimental network, which we leveraged in two ways: First, not only could VINE communicate with the VMs, but the VMs could communicate with VINE. We set up a centralized logging server on VINE which the behavior models published their results to. With this configured, we had close monitoring of each modeled user executing in an experiment. Second, VINE was capable of performing packet capture on the infrastructure, preserving a history of low-level information about the communication of an entire experimental network.

6.2 Community response

VINE has seen significant use both inside and outside our laboratory for several years. Our research assistants use VINE for quick, self-service access to our virtualization infrastructure when they need to create cyber scenarios too large to be instantiated on their local machines. We have found that the layer of abstraction that VINE places above our virtualization provider makes it much easier for users to realize their scenarios than if they were interacting directly with the provider. This was especially true when we switched our provider from VirtualBox to OpenStack,

as we've already mentioned. We have received the same positive feedback from our collaborators in industry, government, and academia. We have received additional industry reinforcement by placing second at the National Homeland Defense Foundation's 2014 National Security Innovation Conference.

Chapter 7

Automated Experimentation

In previous chapters, we have seen how our framework can be used to specify and build individual experiments. Here, we explore how the framework supports *automated experimentation*, which we broadly define as the ability of a system to autonomously run multiple experiments, whether these experiments be the same, similar, or different. In this first part of this chapter, we discuss the types of variations that can be made to a specified experiment, namely in changes to the foundation and orchestration, and how a system could make these variations autonomously. In the second part, we consider how a system that understands and/or generates hypotheses could integrate with a tool that implements our framework.

7.1 Varying an experiment

Authors have noted that what many computer scientists, especially security researchers, refer to as an "experiment" is actually a "one-off evaluation" [10]. What researchers should be performing instead are *comparative experiments* where the effect of multiple treatments are studied together. We can realize this requirement with our framework by building an additional capability[1] that understands the dif-

[1]This capability may be built into the tool that implements our framework, or it may be built as a separate tool. We leave the discussion of precise implementation to future work.

ference between a literal value and a variable, and how to substitute those variables with values from a list. Such a feature allows an experimenter to define an experiment and also a set of treatments to apply to that experiment. The system is then capable of fully executing the experiment with each treatment. Remember, this means building resources, orchestrating the experiment, collecting artifacts, and processing them—all without any input from the user. This alone should be noteworthy, especially to readers who have spent long nights in their laboratory running multiple trials of an experiment, but we are particularly interested in the ability to parallelize these trials in the interest of time. Regardless of the execution order, having a system faithfully execute an experiment greatly reduces the likelihood of human error, thereby boosting the repeatability of the experiment [7].

The most common place a researcher will introduce variation to produce comparative experiments is in the orchestration component, since researchers are often focused on developing new software and orchestration is where software is installed and configured. Recall the example experiment we presented in Chapter 5, where a simple moving target defense was used to stave off attacks on a webserver. In this experiment, two examples of orchestration-based variation are immediately clear to us. First, the intervals used by the defense and the attacker were chosen arbitrarily, but could just as well be tested with a collection of different combinations (e.g. attack interval = 30 seconds, 60 seconds, 120 seconds, ...). Second, we could change the the actual software in the experiment by using different attacks or webservers. Similarly, we could keep the same family of software, but change the version being used to evaluate different sets of patches. Both of these approaches provide more robust results than the traditional "one-off evaluation."

We could also make variations to the experiment foundation. While keeping the type of resource constant, we could vary its attributes. For example, we may run an experiment on different operating systems or with different virtual hardware. It would also be possible to change the number of resources instantiated between trials, i.e. to determine how the system under test responds to changes in scale. In addition to changing the configuration of a resource, we could also change the *type* of resource (e.g. physical machine, virtual machine, Linux Container) or even the

provider who is managing those resources. This requires some care while designing the experiment though because changing the provider and/or resource type may change the channels available, affecting orchestration steps. The DETER Project has identified the relationship between scale and fidelity when running network experiments; that is, increasing the scale of an experiment on the same hardware requires reducing the fidelity of the resources involved [5]. Used in parallel with a variation in the number of resources, it would be possible to automatically run the same experiment at different orders of magnitude, for example on 10 physical machines, then 100 virtual machines, and finally 1000 containers.

Up to this point, we've operated under the assumption that variation takes the form of a variable and a list of values to substitute for that variable. In addition, we've assumed that varying multiple attributes of the experiment would simply require running an experiment for every combination of those attribute values. These are the approaches that software testing tools like Travis CI and tox take, but they may not be sufficient in a cyber experimentation environment. *Exploration strategies* define strategies for automatically and intelligently exploring the parameter space by analyzing previous results [25]. Feedback-based experimentation like this would be useful when seeking the optimal value of a parameter, or when a certain amount of confidence is needed to validate a result. Note that one requirement for dynamic exploration is the definition of a *termination condition* that indicates when the search should halt. Although a termination condition could be based on any metric, common examples are the number of iterations, the time elapsed, or a confidence level [23]. The metric may be something that the tool itself holds, like number of iterations, or an artifact collected from the experiment itself.

7.2 Working with hypotheses

Above, we discussed strategies for automating the execution of multiple related experiments, given that the experiment had already been specified. We are also interested in the process of experiment generation based on hypotheses. (Such a

hypothesis may be, "increasing the switching frequency for a moving target defense decreases the number of successful attacks, with no more than 10% degradation in quality of service.") The experiments generated would take advantage of the types of variation we've already defined in order to gather evidence of a hypothesis, and in this way we are building a layered approach to experimentation in which our framework is the foundation. The ability to generate experiments frees the researcher to focus more on the science of his domain than the design and implementation of experiments. Of course, this capability requires a structured hypothesis and a structured description of how a variety of assets—hardware, software, etc.—can be combined to form a coherent experiment. This topic clearly lies outside the scope of this thesis, but we may explore it in future work.

Hypothesis generation represents a large domain in computer science and has applications to our framework and experimentation in general. For example, the ability to determine dependencies between system components [12] and network services [3] can provide a researcher with information that is directly related to his work, or it can provide supplemental information to his experiment. In other words, consider research where the topic of investigation is not in the foundation or orchestration at all, but is instead completely focused on processing artifacts, specifically to generate hypotheses. If this is the case, the experimenter could perform automated variation in processing in much the same way that it would be conducted in foundation or orchestration: our framework doesn't have any notion of variation, so it doesn't matter where the variation is made. On the other hand, if processing is *not* the focus of the research, automated hypothesis generation can still provide the researcher with insight about his experiment. Regardless, such features can be implemented by creating processing plugins and collecting the necessary artifacts.

Finally, if both of the capabilities above are realized, there is potential to chain them: an experiment is performed, hypotheses are generated, new experiments are created to test those hypotheses, hypotheses are refined, etc. As we've already mentioned, these topics are well beyond the scope of this thesis, but we find the potential applications of our framework exciting. We will continue our work in hopes of realizing these visions.

Chapter 8

Conclusion

In this thesis, we developed a new framework to serve as the foundation of future work in cyber experimentation. First, we surveyed a collection of tools from both academia and industry, and identified a common set of features. Next, we combined this knowledge with a survey of existing work in experimentation frameworks and abstractions to determine the five components of our framework: foundation, channels, orchestration, artifacts, and processing. Unlike existing solutions, this framework is agnostic to environment and domain, and accommodates tools from every part of the experimentation workflow. We showed how a unifying tool based on our framework would improve the experimentation process, and we presented a prototype instantiation of this framework which operates on a restricted interpretation of the framework's components.

Finally, we considered the role our framework can play in future work in automated experimentation. It is well-suited to execute repeated trials of the same or similar experiment, supporting the production of comparative experiments that provide more sound results than one-off evaluations. In addition, it should be possible to run these trials without any human intervention. In future work, we hope to realize this capability and combine it with a system that understands hypotheses in order to automatically generate experiments that can support or disprove those hypotheses.

Bibliography

[1] Jeannie Albrecht, Christopher Tuttle, Ryan Braud, Darren Dao, Nikolay Topilski, Alex C. Snoeren, and Amin Vahdat. Distributed application configuration, management, and visualization with Plush. *ACM Trans. Internet Technol.*, 11(2):6:1–6:41, December 2011.

[2] Luca Allodi, Vadim Kotov, and Fabio Massacci. MalwareLab: Experimentation with cybercrime attack tools. In *Proceedings of the 6th Workshop on Cyber Security Experimentation and Test (CSET)*, 2013.

[3] Paramvir Bahl, Ranveer Chandra, Albert Greenberg, Srikanth Kandula, David A. Maltz, and Ming Zhang. Towards highly reliable enterprise network services via inference of multi-level dependencies. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 13–24, New York, NY, USA, 2007. ACM.

[4] Terry Benzel. The science of cyber security experimentation: The DETER Project. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, pages 137–148, New York, NY, USA, 2011. ACM.

[5] Terry Benzel and John Wroclawski. The DETER Project: Towards structural advances in experimental cybersecurity research and evaluation. *Journal of Information Processing*, 20(4):824–834, 2012.

[6] Stanislav Dashevskyi, Daniel Ricardo Dos Santos, Fabio Massacci, and Antonino Sabetta. TestREx: a testbed for repeatable exploits. In *Proceedings of the 7th USENIX conference on Cyber Security Experimentation and Test*, pages 1–8. USENIX Association, 2014.

[7] Sarah Edwards, Xuan Liu, and Niky Riga. Creating repeatable computer science and networking experiments on shared, public testbeds. *SIGOPS Oper. Syst. Rev.*, 49(1):90–99, January 2015.

[8] Peter Haglich, Robert Grimshaw, Steven Wilder, Marian Nodine, and Bryan Lyles. Cyber scientific test language. In Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Noy, and Eva Blomqvist, editors, *The Semantic Web – ISWC 2011*, volume 7032 of *Lecture Notes in Computer Science*, pages 97–111. Springer Berlin Heidelberg, 2011.

[9] Guillaume Jourjon, Thierry Rakotoarivelo, and Max Ott. A portal to support rigorous experimental methodology in networking research. In Thanasis Korakis, Hongbin Li, Phuoc Tran-Gia, and Hong-Shik Park, editors, *Testbeds and Research Infrastructure. Development of Networks and Communities*, volume 90 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 223–238. Springer Berlin Heidelberg, 2012.

[10] Kevin S Killourhy and Roy A Maxion. Should security researchers experiment more and draw more inferences? In *Proceedings of the 4th Workshop on Cyber Security Experimentation and Test (CSET)*, 2011.

[11] Olaf Landsiedel, Georg Kunz, Stefan Götz, and Klaus Wehrle. A virtual platform for network experimentation. In *Proceedings of the 1st ACM Workshop on Virtualized Infrastructure Systems and Architectures*, VISA '09, pages 45–52, New York, NY, USA, 2009. ACM.

[12] Jian-Guang Lou, Qiang Fu, Yi Wang, and Jiang Li. Mining dependency in distributed systems through unstructured logs analysis. *SIGOPS Oper. Syst. Rev.*, 44(1):91–96, March 2010.

[13] Roy A. Maxion, Thomas A. Longstaff, and John McHugh. Why is there no science in cyber science?: A panel discussion at NSPW 2010. In *Proceedings of the 2010 Workshop on New Security Paradigms*, NSPW '10, pages 1–6, New York, NY, USA, 2010. ACM.

[14] Olivier Mehani, Guillaume Jourjon, Thierry Rakotoarivelo, and Max Ott. An instrumentation framework for the critical task of measurement collection in the future internet. *Computer Networks*, 63(0):68 – 83, 2014. Special issue on Future Internet Testbeds – Part II.

[15] Marian Nodine, Robert Grimshaw, Peter Haglich, Steven Wilder, and J Bryan Lyles. Computational asset description for cyber experiment support using OWL. In *Proceedings of the 5th IEEE International Conference on Semantic Computing (ICSC)*, pages 110–117. IEEE, 2011.

[16] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the internet. *SIGCOMM Comput. Commun. Rev.*, 33(1):59–64, January 2003.

[17] Alina Quereilhac, Mathieu Lacage, Claudio Freire, Thierry Turletti, and Walid Dabbous. NEPI: An integration framework for network experimentation. *19th International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 1–5, 2011.

[18] Thierry Rakotoarivelo, Maximilian Ott, Guillaume Jourjon, and Ivan Seskar. OMF: A control and management framework for networking testbeds. *SIGOPS Oper. Syst. Rev.*, 43(4):54–59, January 2010.

[19] Niky Riga. GENIExperimenter/RSpecs, May 2012. `http://groups.geni.net/geni/wiki/GENIExperimenter/RSpecs` (accessed March 28, 2015).

[20] Stephen Schwab, Brett Wilson, C. Ko, and A. Hussain. SEER: a security experimentation environment for DETER. In *Proceedings of the DETER Community Workshop on Cyber Security Experimentation and Test*, page 6, 2007.

[21] Evan Stoner, Marco Carvalho, and Thomas Eskridge. VINE: A cyber emulation environment for advanced experimentation and training. In *2014 National Security Innovation Competition Proceedings Report*, pages 14–17, Colorado Springs, CO, USA, 2014. National Homeland Defense Foundation.

[22] Jonathan S. Turner. A proposed architecture for the GENI backbone platform. In *Proceedings of the 2006 ACM/IEEE Symposium on Architecture for Networking and Communications Systems*, ANCS '06, pages 1–10, New York, NY, USA, 2006. ACM.

[23] Dennis Westermann, Jens Happe, and Roozbeh Farahbod. An experiment specification language for goal-driven, automated performance evaluations. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, pages 1043–1048, New York, NY, USA, 2013. ACM.

[24] Dennis Westermann, Jens Happe, Michael Hauck, and Christian Heupel. The performance cockpit approach: A framework for systematic performance evaluations. In *Proceedings of the 36th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 31–38. IEEE, 2010.

[25] Dennis Westermann, Jens Happe, Rouven Krebs, and Roozbeh Farahbod. Automated inference of goal-oriented performance prediction functions. *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, page 190, 2012.