Generating Client Workloads and High-Fidelity Network Traffic for Controllable, Repeatable Experiments in Computer Security*

Charles V. Wright, Christopher Connelly, Timothy Braje,
Jesse C. Rabek**, Lee M. Rossey, and Robert K. Cunningham

Information Systems Technology Group
MIT Lincoln Laboratory
Lexington, MA 02420
{cvwright,connelly,tbraje,lee,rkc}@ll.mit.edu, jesrab@alum.mit.edu

Abstract. Rigorous scientific experimentation in system and network security remains an elusive goal. Recent work has outlined three basic requirements for experiments, namely that hypotheses must be *falsifiable*, experiments must be *controllable*, and experiments must be *repeatable* and *reproducible*. Despite their simplicity, these goals are difficult to achieve, especially when dealing with client-side threats and defenses, where often user input is required as part of the experiment. In this paper, we present techniques for making experiments involving security and client-side desktop applications like web browsers, PDF readers, or host-based firewalls or intrusion detection systems more *controllable* and more easily *repeatable*. First, we present techniques for using statistical models of user behavior to drive real, binary, GUI-enabled application programs in place of a human user. Second, we present techniques based on adaptive replay of application dialog that allow us to quickly and efficiently reproduce reasonable mock-ups of remotely-hosted applications to give the illusion of Internet connectedness on an isolated testbed. We demonstrate the utility of these techniques in an example experiment comparing the system resource consumption of a Windows machine running anti-virus protection versus an unprotected system.

Keywords: Network Testbeds, Assessment and Benchmarking, Traffic Generation.

1 Introduction

The goal of conducting disciplined, reproducible, "bench style" laboratory research in system and network security has been widely acknowledged [1,2], but remains difficult to achieve. In particular, Peisert and Bishop [2] outline three

* This work was supported by the US Air Force under Air Force contract FA8721-05-C-0002. The opinions, interpretations, conclusions, and recommendations are those of the authors and are not necessarily endorsed by the United States Government.

** Work performed as a student at MIT. The author is now with Palm, Inc.

S. Jha, R. Sommer, and C. Kreibich (Eds.): RAID 2010, LNCS 6307, pp. 218–237, 2010.

basic requirements for performing good experiments in security: (*i*) Hypotheses must be *falsifiable*—that is, it must be possible to design an experiment to either support or refute the hypothesis. Therefore, the hypothesis must pertain to properties that are both *observable* and *measurable*. (*ii*) Experiments must be *controllable*; the experimenter should be able to change only one variable at a time and measure the change in results. (*iii*) Finally, experiments should be both *repeatable*, meaning that the researcher can perform them several times and get similar results, and *reproducible*, meaning that others can recreate the experiment and obtain similar results.

Unfortunately, designing an experiment in system or network security that meets these requirements remains challenging. Current practices for measuring security properties have recently been described as "ad-hoc," "subjective," or "procedural" [3]. Experiments that deal primarily with hardware and software may be extremely controllable, and recent work [4,5,6,7,8] has explored techniques for deploying and configuring entire networks of servers, PCs, and networking equipment on isolated testbeds, disconnected from the Internet or other networks, where malicious code may safely be allowed to run with low risk of infecting the rest of the world. However, the recent shift in attacks from the server side to the client side [9,10,11,12] means that an experiment involving any one of many current threats, such as drive-by downloads [13] cross-site scripting, or techniques for detecting and mitigating such intrusions, must account for the behavior of not only the hardware and software of the computing infrastructure itself, but also the behavior of the human users of this infrastructure. Humans remain notoriously difficult to control, and experiments using human subjects are often expensive, time consuming, and may require extensive interaction with internal review boards.

Repeatability of experiments on the Internet is difficult due to the global network's scale and its constant state of change and evolution. Even on an isolated testbed, repeatability is hampered by the sheer complexity of modern computer systems. Even relatively simple components like hard disks and Ethernet switches maintain state internally (in cache buffers and ARP tables, respectively), and many components perform differently under varying environmental conditions e.g. temperature. Many recent CPUs dynamically adjust their clock frequency in reaction to changes in temperature, and studies by Google suggest that temperature plays an important role in the failure rates of hard disk drives [14]. Reproducibility is even harder. It is unclear what level of detail is sufficient for describing the hardware and software used in a test, but current practices in the community likely fall short of the standards for publication in the physical sciences.

The contributions of this paper address two of the above requirements for performing scientific experiments in security. Specifically, we describe techniques that enable *controllable*, *repeatable* experiments with client-side attacks and defenses on isolated testbed networks. First, we present techniques for using statistical models of human behavior to drive real, binary, GUI-enabled application programs running on client machines on the testbed, so that tests can be

performed without the randomness or privacy concerns inherent to using human subjects. Second, we present adaptive replay techniques for producing convincing facsimiles of remotely-hosted applications (e.g. those on the World Wide Web) that cannot themselves be installed in an isolated testbed network, so that the client-side applications have something to talk to. In doing so, we generate workloads on the hosts and traffic on the network that are both highly controllable and repeatable in a laboratory testbed setting.

On the client side, our approach is to construct a Markov chain model for the way real users interact with each application. Then, during the experiment, we use the Markov chains to generate new event streams similar in distribution to those generated by the real users, and use these to drive the applications on the testbed. This provides a realistic model of measured human behavior, offers variability from trial to trial, and provides an experimenter with the ability to change model parameters to explore new user classes. It also generates a reasonably realistic set of workloads on the host, in terms of running processes, files and directories accessed, open network ports, system call sequences, and system resource consumption (e.g. CPU, memory, disk). Many of these properties of the system are important for experiments involving defensive tools like firewalls, virus scanners, or other intrusion detection systems because they are used by such systems to detect or prevent malicious behavior. Furthermore, because we run unmodified application program binaries on the testbed hosts, we can closely replicate the attack surface of a real network and use the testbed to judge the effectiveness of various real attacks and defenses against one another.

Using real applications also allows us to generate valid traffic on the testbed network, even for complicated protocols that are proprietary, undocumented, or otherwise poorly understood. We discuss related work in more detail in the following section, but for now it suffices to say that almost all existing work on synthetically generating network traffic focuses on achieving realism at only one or two layers of the protocol stack. In contrast, our approach provides realistic traffic all the way from the link layer up to and including the contents of the application layer sessions.

For example, by emulating a user replying to an email, with just a few mouse click events, we can generate valid application-layer traffic in open protocols like DNS, IMAP, and LDAP, proprietary protocols including SMB/CIFS, DCOM, and MAPI/RPC (Exchange mail). This is, of course, in addition to the SMTP connection used to send the actual message. Each of these connections will exhibit the correct TCP dynamics for the given operating system and will generate the proper set of interactions at lower layers of the stack, including DNS lookups, ARP requests, and possibly Ethernet collisions and exponential backoff. Moreover, if a message in the user's inbox contains an exploit for his mail client (like the mass-mailing viruses of the late 1990s and early 2000s), simply injecting a mouse click event to open the mail client may launch a wave of infections across the testbed network.

For the case where the actual applications cannot be installed on the isolated test network, we present techniques based on adaptive replay of application

dialog that allow us to quickly and efficiently reproduce reasonable mock-ups that make it appear across the network as if the real applications were actually running on the testbed. These techniques are particularly useful for creating a superficially realistic version of the modern World Wide Web, giving the illusion of connectedness on an isolated network.

To illustrate the utility of these techniques, we perform a simple experiment that would be labor intensive and time consuming to conduct without such tools. Specifically, we investigate the performance impact of open source anti-virus (AV) software on client machines. Conventional folk wisdom in the security community has been that AV products incur a significant performance penalty, and this has been used to explain the difficulty of convincing end users to employ such protection. Surprisingly, relatively little effort has been put in to quantifying the drop in performance incurred, perhaps due to the difficulty of performing such a test in a controllable and repeatable manner.

The remainder of the paper is organized as follows. In Section 2, we review related work in network testbeds, automation of GUI applications, modeling user behavior, and network traffic generation. In Section 3, we present our techniques for driving real binary applications and for crafting reasonable facsimiles of networked applications that we cannot actually install on the testbed. In Section 4, we walk through a simple experiment to demonstrate the utility of these techniques and to highlight some challenges in obtaining repeatable results. Finally, we conclude in Section 5 with some thoughts on future directions for research in this area.

2 Related Work

Several approaches for configuring, automating, and managing network laboratory testbeds have recently been proposed, including Emulab [4], FlexLab [5], ModelNet [6], and VINI [7]. Our group's LARIAT testbed platform [8] grew out of earlier work in the DARPA intrusion detection evaluations [15,16] and was designed specifically for tests of network security applications. More recently, along with others in our group, two of the current authors developed a graphical user interface for testbed management and situational awareness [17] for use with LARIAT. The DETER testbed [18] is built on Emulab [4] and, like LARIAT, is also geared toward network security experiments. The primary contribution of this paper, which is complementary to the above approaches, is to generate client-side workloads and network traffic for experiments on such testbeds. The techniques in Section 3.1 were first described in the fourth author's (unpublished) MIT Master's thesis [19]. USim, by Garg et al. [20], uses similar techniques for building profiles of user behavior, and uses scripted templates to generate data sets for testing intrusion detection systems.

Our server-side approach for emulating the Web is similar to the dynamic application layer replay techniques of Cui et al. [21,22] and Small et al. [23]. Like our client-side approach, the MITRE HoneyClient [24] and Strider HoneyMonkeys from Microsoft Research [25] drive real GUI applications, but that work

focuses narrowly on automating web browsers to discover new vulnerabilities and does not attempt to model the behavior of a real human at the controls. Software frameworks exist for the general-purpose automation of GUI applications, including `autopy` [26] and SIKULI [27], but these also require higher-level logic for deciding which commands to inject. PLUM [28] is a system for learning models of user behavior from an instrumented desktop environment. Simpson et al. [29] and Kurz et al. [30] present techniques for deriving empirical models of user behavior from network logs.

There is a large body of existing work on generating network traffic for use on testbeds or in simulations, but unfortunately most of these techniques were not designed for security experiments. Simply replaying real traffic [31,32] does not allow for controllable experiments. Other techniques for generating synthetic traffic based on models learned from real traffic [33,34,35,36,37] can match several important statistical properties of the input trace at the Network and Transport layers. However, because these approaches do not generate application layer traffic, they are not compatible with many security tools like content-based filters and intrusion detection or prevention systems, and they cannot interact with real applications on a testbed. Sommers et al. [38] present a hybrid replay-synthesis approach that may be more appropriate for some experiments in security. Mutz et al. [39], Kayacik and Zincir-Heywood [40], and other work by Sommers et al. [41] generate traffic specifically for the evaluation of defensive tools.

Commercial products from companies including Ixia, BreakingPoint, and Spirent can generate application-layer traffic, but their focus is on achieving high data rates rather than realistic models of individual user behavior, and their implementations do not necessarily exhibit the same attack surface as the real applications.

3 Traffic and Workload Generation Techniques

Although our techniques could potentially be applied using any of the existing network testbeds [4,5,6,7,8,18], our current implementation is built as an extension of own testbed platform, LARIAT [8], which provides a centralized database for experiment configuration and logging and a graphical user interface for launching automated tasks to configure the testbed and for controlling and monitoring experiments. Since the publication of [8], the scope of the project has expanded significantly. LARIAT has been used to run distributed experiments on testbeds of more than a thousand hosts. In addition to the user model-driven actuation capabilities and internet reproduction described in this paper, components have been added for automatically configuring client and server software, controlling hosts across a testbed, visualizing the configuration and logged data [17], and for distributing control across remote physical locations. The current version can drive user-model behavior on a number of different operating system and physical device platforms including smart phones and router consoles.

3.1 Client-Side Workload Generation

Our approach is to emulate a human user by injecting input events to applications via the operating system. In principle, we could use any number of possible techniques to determine what events to inject and when. One simple approach would be to simply record the sequence of events generated by a real user, and replay them verbatim to the applications on the testbed. While this "capture/replay" approach offers a level of realism that is difficult to match with synthetic workloads, it fails the requirement that experiments be *controllable*.

Our techniques strike a careful balance between realism of the workloads and controllability of the experiment. We record the inputs generated by real human users and then train a hierarchical Markov chain model for the events sent to each application. Then, during the experiment, we simulate from the Markov chains to generate new event streams similar in distribution to those generated by the real users, and use these to drive the applications on the testbed.

Application User State Machines. We call these models Application User State Machines, or AUSMs, because the Markov chain models describe a finite state machine model of a human user of the application. Formally, an AUSM is defined as a 4-tuple (n, A, M, X), where n is the number of states in the finite state machine model, $A = \{a_{ij} : i, j < n\}$ is the Markov chain state transition matrix, $M = \{m_i : i < n\}$ is a set of second-level models for the outputs produced by each state, and $X = \{X_{ij} : i, j < n\}$ is a set of models describing the interarrival time distribution when an event of type i is immediately followed by an event of type j. We describe the training and event generation processes for these models in greater detail in the following paragraphs.

Setting AUSM Parameters. To collect training data for the AUSM's, we use the DETOURS framework [42] from Microsoft Research to instrument a set of Windows desktop machines as they are driven by real human users. During the training interval, we record the event ID, process ID, and arrival time of each COM (Component Object Model) event on these instrumented systems for some length of time. We then use the sub-sequence of events corresponding to each application to set the parameters for a hierarchical Markov chain model that we then use to drive the given application on the testbed.

To create an AUSM, we begin by creating one state for each event ID. We count the number of times in the training data where event i was immediately followed by event j, and store this count as c_{ij}. We then compute the probability of a transition from state i to state j, and store this in the Markov model's state transition matrix as:

$$a_{ij} = \frac{c_{ij}}{\sum_k c_{ik}}$$

Modeling State Output Distributions. To allow for flexibility in the level of detail provided by the AUSMs, the outputs of each state are represented using a second level of models. Some states may always produce the same output, e.g. a state that generates a mouse click on the "Start" button. Others, like the state that

generates input for a text box in Internet Explorer, or the word processor input model, use an n-gram word model of English to produce blocks of text at a time.

If we have no other source of data, these output models can be trained using the values observed during the training data collection. In other cases, where we have some expert knowledge of the application, the output models can be trained using other, larger external data sources. For example, the model that generates text for the body of an email could be trained using the contents of real emails in the Enron corpus [43]. In our experiments, we use a locally-collected corpus of real emails from the authors' inboxes to train a bigram word model of English text.

Modeling Event Interarrival Times. Each state transition edge (i, j) in the AUSM also has an associated interarrival time distribution X_{ij}, which characterizes the delay between events when event i is immediately followed by event j. Typically, waiting time distributions are well described by the exponential distribution (e.g. time between buses arriving at a bus stop, time between major hurricanes, etc.). However, the data collected from our users' workstations exhibits a heavier tail than the exponential distribution, with more wait times that are much longer than the mean. Some so-called "heavy-tailed" distributions that occur as a result of user interaction have been shown to be well described by a power-law or Pareto distribution in the past [44], although the Pareto distribution also does not appear to be a good fit for our event interarrivals. Figure 1 shows the observed empirical distribution of COM event interarrival times for one state transition, together with the best-fit exponential and Pareto distributions.

Our hypothesis for the poor fit of these two distributions is that there are actually two sub-populations of event interarrival times. In the first case, the user is actively engaged with the application, generating events at shorter and more regular intervals. In the second case, the user may switch to another application or disengage from the system entirely to perform some other task, such

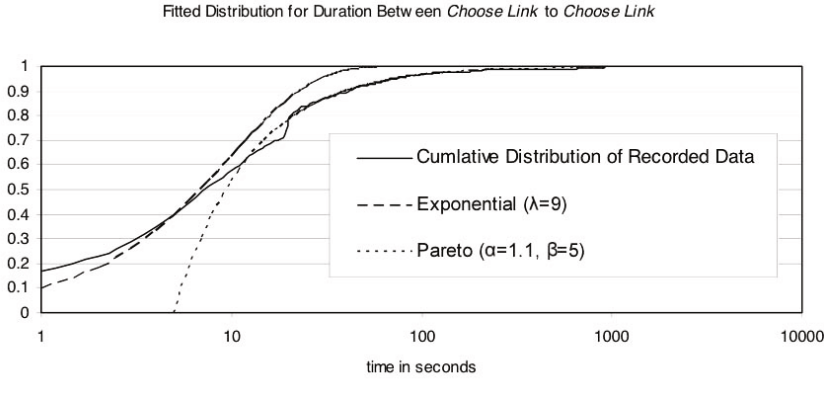

Fig. 1. Empirical distribution of event interarrival times, with best-fit Exponential and Pareto distributions

as answering the telephone, reading a paper, going to a meeting, going home for the night, or even going on vacation while leaving the system up and running. To capture this bimodal distribution, we use a mixture model with one Exponential component to represent the periods of active engagement and one Pareto component to represent the longer periods of inactivity.

Generating Client Workloads. In this section we explain how the state machine models developed above can be used to feed input to application programs on a client machine to generate workloads on the host and traffic on the testbed network. Figure 2 shows at a high level how our modules interface with the Windows OS and applications on the client-side system under test (SUT) to achieve the illusion of a human user at the controls.

Regarding Repeatability. We note that, in order to achieve repeatable experimental results, the entire testbed needs to be started from a fixed state at the beginning of each run. While we believe the approach we describe here is a necessary condition for obtaining repeatable experimental results, this alone is not sufficient. We elaborate on other techniques for improving the repeatability of an example experiment in Section 4.

To enable repeatable outputs from our state machines, we store a master random seed in the LARIAT database for each experiment. As part of setting up the testbed for the experiment, each host generates its own unique random seed as a hash of the master random seed and a unique host identifier assigned to it by the LARIAT testbed management system. At the beginning of an experiment, each host instantiates a Mersenne Twister [45] pseudo-random number generator, seeded with its host seed. This PRNG is then used to drive the state machines as explained above. Thus, by keeping the master seed unchanged for several runs of the experiment, we can repeat a test many times and get the same sequence of actions from the state machines in each run. Conversely, by varying the master seed, we can explore the space of possible user actions and the corresponding experimental outcomes.

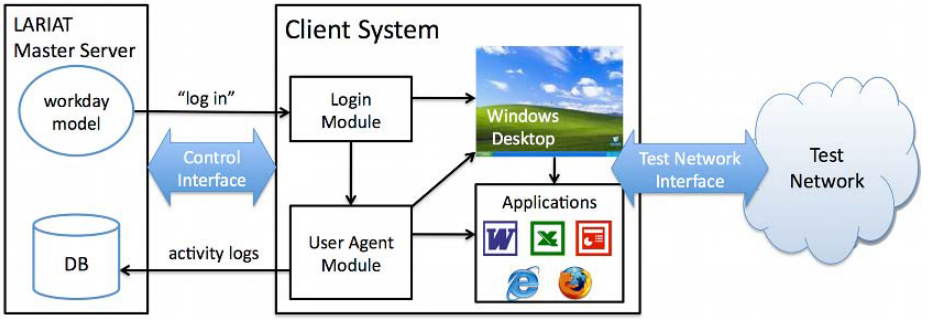

Fig. 2. Client-side traffic generation overview

To simulate the user arriving at the machine and logging in, the master LARIAT server sends a message to the client host's login module over the control interface, instructing it to log in the given user. On Windows NT, 2000, and XP systems, the login module is implemented as a GINA, a dynamic-link library used by the Windows *Winlogon* process for Graphical Identification and Authentication [46]. On Windows Vista and newer versions, it runs as a service. In either case, the module provides login credentials to the OS to start up a desktop session for the given user. It also launches the user agent module, which generates user input to drive the Windows desktop and applications from that point forward.

Upon login, the user agent module starts with a pseudorandomly-selected AUSM and, if necessary, launches the corresponding application. Then, until the user agent process receives a signal instructing it to log the user out, it generates input for the applications by driving the state machines as follows.

In state i, the user agent first samples from state i's output model m_i to generate an input to the application. It injects the input events using the Microsoft COM APIs or as keyboard events so that, from the applications' point of view, these events are delivered by the operating system just as if they had been generated by a real human user. Then, the user agent selects the next state j by pseudorandomly sampling from row i of the Markov model's state transition matrix A. The user agent samples a pseudorandom delay x from the AUSM's event interarrival time distribution X_{ij}. It then sleeps for x seconds, resets the current state to j, and repeats the process. In some cases, the output of state j may be to launch a new application or switch to another running application. In such cases, the user agent also switches to using the new application's AUSM in the next iteration.

3.2 Server Side Techniques

For our client-side workload generation techniques to truly be useful on an isolated testbed network, there must be something for the client side applications to talk to. Sometimes this is relatively straightforward. For example, simply installing and configuring a Microsoft Exchange email and calendaring server on the client's local area network is mostly sufficient to enable the MS Outlook AUSM to function normally. Our previous work [8] presents techniques for generating emails for the virtual users to receive, and of course the Domain Name Service and IP layer routing must be properly configured on the testbed so that clients can discover one another's SMTP servers and transmit the actual mail. Some testbed management systems [47,8,18] handle part or all of this setup process.

For some other network applications, most notably the world-wide web, setting up a realistic environment on an isolated network is much more challenging. Although installing a server for the underlying HTTP protocol is not especially difficult, getting realistic content is. In the early days of the web, most pages consisted solely of static content, which could easily be downloaded and "mirrored" on another server to easily replicate the page. While some web pages

still use this model, for example many researchers' profile pages, the majority of the most popular web sites are currently powered by special-purpose, proprietary programs that dynamically generate page content and are only accessible as a service. Some web applications for dynamically generating page content are available for installation on the testbed, either as software packages, or as a hardware appliance such as the Google Search Appliance [48], and we do make use of several such products, including the open source *osCommerce* [49] e-commerce engine, the *GreyMatter* weblog software, and Microsoft Exchange's webmail interface.

However, to make it appear on the surface as if the isolated testbed network is actually connected to the Internet, more sophisticated techniques are required. Our approach is to use dynamic application-layer replay techniques like those developed by Cui et al. [22,21] and Small et al. [23] for creating lightweight server-side honeypots. We elaborate on our approach in the following sections.

Collecting Data. We begin by downloading a large number of web pages using what is essentially a client-side honeypot [25,24]. That is, we run a web browser (in our case Microsoft Internet Explorer) on a Windows operating system in a virtual machine, and we script it to automatically download a list of URLs via a consumer-grade cable modem connection to the Internet. For each URL in the list, we retrieve the page using the honeyclient and record the full contents of each packet that is generated. We revert the VM to a clean state after each page retrieval. For broad coverage of the Web, we begin with a list of over ten thousand URLs from a categorized directory of links such as Mozilla's Open Directory Project [50] or Yahoo!'s directory [51], as well as lists of the most popular URLs from the Alexa.com rankings [52]. For increased realism, we can script the honey client to "crawl" more links to provide increased depth for a given interactive site.

Then, we perform TCP stream reassembly on the captured packets to reconstruct the application-layer conversations for each URL and all of the associated HTTP sessions. For each HTTP request in the collected traces, we store the full text of the resulting HTTP response, including both the headers and the body, in a database, keyed based on the hostname used in the request and the URL requested, as well as some meta-information from the headers such as transport and content encoding parameters.

Emulating the Web. On the testbed network, we deploy a very simple web server program to emulate the Web using the data collected above. Upon receiving an HTTP request, it first parses out the hostname, URL, and other parameters, then looks up the corresponding HTTP response text in the database, and finally sends this response back to the client. The content from the database can be distributed across several web servers to provide the ability to handle large traffic loads as well as provide more realistic network characteristics for the traffic.

To give the impression of a much larger network than can be realistically installed on a testbed, we typically deploy this web server on several Linux

machines, each configured with hundreds or even thousands of virtual network interfaces for each physical interface. Each machine can thus respond to HTTP requests sent to any one of thousands of IP addresses. Each instance of the web server application listens on a designated subset of the host's IP addresses and serves content for a designated set of web sites. This flexibility enables us to emulate both very simple sites hosted at a single IP address as well as dynamic, world-wide content distribution networks. We store the mapping from hostnames to IP addresses and Linux hosts in the testbed's central LARIAT database. There, this information is also used to configure the testbed's DNS servers, so that client nodes can resolve hostnames to the proper virtual IP addresses. We also provide artificial Root DNS servers as well as a core BGP routing infrastructure to redistribute all of the routing information for these IP addresses.

Discussion and Limitations. This combination of lightweight application-level replay on the server side with automation of heavyweight GUI applications on the client side allows us to generate very high-fidelity network traffic for many use cases. It requires no parsing or understanding of JavaScript, but many JavaScript-heavy sites can be emulated using this method and appear fully functional from the client's perspective, limited only by the extent of the data collection. One notable example of such a site is Google Maps.

However, the focus on light weight and efficiency in our server-side replay techniques leads to some important limitations of the current implementation. First, because the server is stateless, it cannot do HTTP authorization or any customization of page content based on cookies. Second, because it only looks for exact matches in the URLs, some pages that dynamically generate links may fail to find a matching page when run on the testbed. Pages that dynamically choose IP addresses or hostnames for links may need to be fetched tens or even hundreds of times during the data collection step in order to find all IP addresses or hostnames that should occur in the linked pages' URLs. Otherwise, any client-side JavaScript code that uses random numbers to control its actions (e.g. client-side load balancing) will fail to function given that previously unrequested URLs will not be found in the new closed environment. Finally, while our approach could be almost trivially extended to support the concurrent use of multiple browsers or multiple operating systems, it does not do so currently.

Despite these limitations, the current techniques are sufficient for many kinds of experiments involving network traffic. They are also valuable for tests that focus primarily on host behavior, as they enable a wider range of applications to be run on the host, most notably the web browser. In the next section, we walk through a simple experiment with a host-based security system where use of the browser makes up a significant fraction of the client machine's workload.

4 An Example Experiment

In this section we walk through a simple experiment as an example of the kind of test our system enables a researcher to perform. Although the underlying

LARIAT test range automation tool and the AUSM-based workload generators are capable of scaling to hundreds or even thousands of nodes, for ease of exposition, we will limit ourselves to a much more limited test scenario in this paper. Despite its small scale and relative simplicity, we believe this experiment is still complex enough to illustrate the difficulties in applying the scientific method to problems in computer security.

Specifically, the goal of our example experiment is to measure and quantify the performance penalty incurred by running anti-virus protection on desktop computers. In principle, very nearly the same experiment could be used to measure the performance impact of many other security tools, including other kinds of intrusion detection systems such as content-based filters, or many types of malware like rootkits, adware, spyware, or key loggers. We chose to measure the impact of AV systems because (1) they are ubiquitous on Internet-connected machines, and (2) because although users have long complained that AV negatively impacts system performance, very little hard data has been presented to either refute or support this claim.

Hypothesis. We begin by defining a falsifiable hypothesis. A simple statement such as *"anti-virus makes the system slow"* is not a good hypothesis because slowness is subjective and is therefore not *measurable* without a substantial user study; it depends not only on the performance of the system but also on the perception of the user. Instead, we use a similar hypothesis that we hope will be a good predictor of perceived slowness, namely that "*anti-virus increases the system's resource consumption.*" In related work, others have tested a similar hypothesis, namely that "*anti-virus increases the time required to complete a suite of computational tasks*" [53,54].

4.1 Testbed Setup

We use a very simple experimental testbed comprised of two physical machines. One of these machines is a server-class machine (HOST) which we use to provide the LARIAT infrastructure. HOST is a Dell PowerEdge 2650 with dual Intel Xeon 2.60GHz processors and 2GB of RAM. On HOST, we deploy two virtual servers using VMWare Server. One of these is the LARIAT control server (CONTROL) and the other (INTERNET) provides email, DNS, and world-wide web services (Section 3.2) on the testbed for use by the system under test. The second machine in our setup is a Dell Latitude D610 laptop (SUT, for "system under test") with a 1.7GHz Pentium M processor and 1GB of RAM. We partition the hard disk of the SUT into two pieces. On one partition, we install Ubuntu Linux 9.10. On the other, we install Windows XP with Service Pack 2, Microsoft Office XP, and our client-side workload generation tools, including the login module, the user agent, and the AUSMs for Internet Explorer, Word, Excel, PowerPoint, and Outlook. To enable the collection of performance measurements from the system under test, we install components of SGI's *Performance Co-Pilot* (PCP) software [55] on the Windows partition of the SUT,

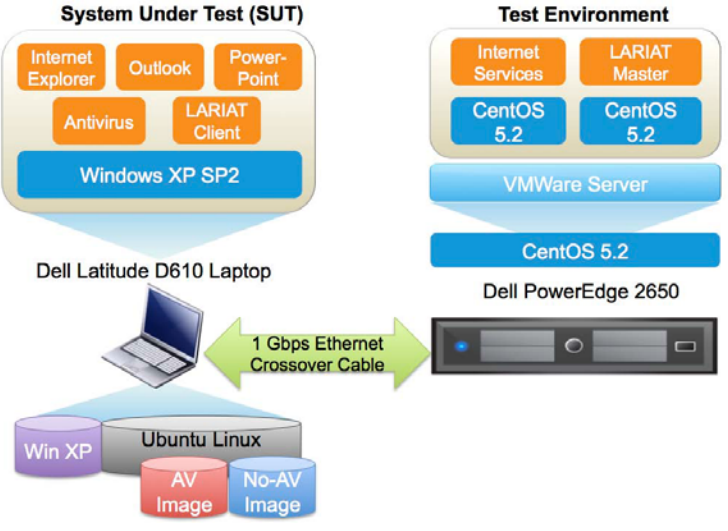

Fig. 3. Test Network Physical Topology

where it can collect performance information, and on HOST, where it can log these measurements for future analysis. We also install the `winexe` remote administration tool on HOST so that we can automatically reboot the laptop when it is in Windows.

Then, from the SUT's Linux installation, we use the Unix tool `dd` to make a byte-level image of the Windows partition and store this as a file in the Linux system. We then re-boot into the Windows system and install an open source anti-virus product, reboot back into the Linux system and make another byte-level copy of the Windows partition with the open source AV product installed. At the completion of this process, we have two Windows disk images on the Linux filesystem: (1) a clean snapshot of the Windows system, with no AV software installed and (2) a snapshot of the clean image with the open source product installed.

4.2 Experimental Methods

In Section 3.1, we explained how we use pseudorandom number generators to drive the AUSMs to deliver repeatable user inputs to the application programs on the testbed. However, more steps are necessary to achieve repeatable results from a system as complex as a modern computer or network. In this section, we discuss the steps we take to improve the repeatability of our very small, simple example experiment. First, we note that we intentionally deploy the SUT on a physical machine instead of using virtualization because of difficulty in obtaining repeatable timing measurements on current VM platforms [56]. Our experimental procedure is outlined in Figure 4; we describe the process in more detail below.

1. **Prepare Systems for Test Run**
   (a) Revert disk images on SUT and INTERNET
   (b) Revert system clocks on SUT and INTERNET
   (c) Reboot SUT laptop into Windows environment
   (d) Seed PRNGs using master experiment seed
   (e) Start PCP performance logging service
2. **Execute Test Run**
   (a) Start AUSM-based client workload generation
   (b) Let workload generation run for 2 hours
   (c) Stop AUSM-based client workload generation
3. **Collect Results**
   (a) Stop PCP performance logging service
   (b) Archive performance logs
   (c) Reboot SUT laptop into Linux environment

Fig. 4. Experimental Procedure

When running a test, we begin with the SUT booted into its Linux environment, HOST powered up, and the INTERNET and CONTROL virtual machines running. We revert the disk images on INTERNET and SUT's Windows partition to clean states, using VMWare's snapshot feature and the Unix `dd` tool, respectively. This is necessary to ensure that SUT's filesystem is unchanged from run to run, and that its view of the Internet, including queued email in the user's inbox, is consistent in each run. Next, we set the system clocks on SUT and INTERNET to a constant value. Setting the clocks before each run is important because many application programs also use pseudorandom number generators, and many of them seed their PRNG with the current value of the clock when they start up. Finally, we set the `GRUB` boot loader on the SUT to load Windows on its next invocation, and we reboot the laptop. While the laptop performs its power-on self test and boots into Windows, we start the PCP performance logger on HOST so that it is ready to begin recording performance metrics from SUT when it comes on line. To maximize repeatability, all of these actions are performed automatically by a set of three shell scripts. One script, the master, runs on HOST and executes the other two scripts to re-set the SUT and the INTERNET VM before launching each run of the experiment.

In each run, the master script sends a command to CONTROL to start the experiment, then sleeps for a specified length of time in order to let the test run. Meanwhile, CONTROL sends the command to SUT, which logs in the virtual user and starts the AUSMs to drive the client-side applications on the testbed. When the master script wakes up, it sends another command to CONTROL, stopping the test and logging the user out. It archives the PCP performance logs and then uses `winexe` to reboot the laptop into its Linux environment in preparation for the next run.

4.3 Experimental Results

Using the above procedure, we select a master experiment seed at random and repeatedly execute a 2-hour test on our testbed with this seed. We run the test

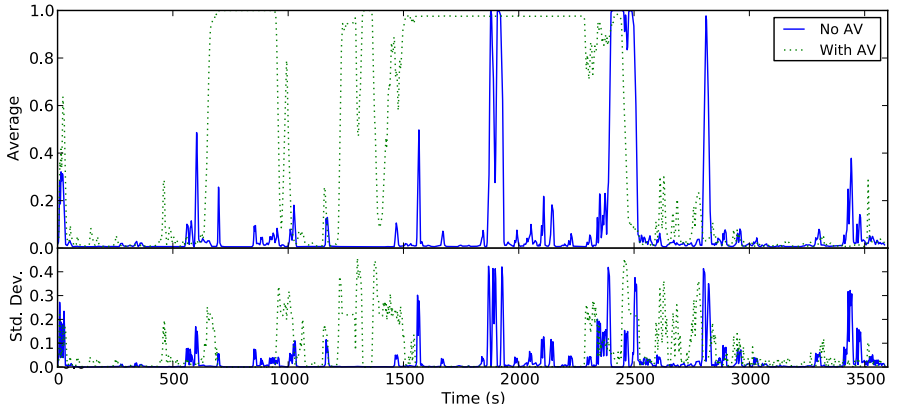

Fig. 5. CPU Utilization

under two distinct scenarios: (1) the baseline, with no anti-virus protection on the SUT and (2) with the open source anti-virus product. These experiments serve two purposes. First, they allow us to test whether there is any measurable difference in system resource consumption between the two scenarios. They also serve to demonstrate the repeatability of results enabled by our techniques.

With the experiment seed that we selected, the user agent launches both Outlook and Word soon after logging in. It spends several minutes reading and writing email and writing Word documents, then launches Internet Explorer and browses to several sites on the emulated web. It keeps these three applications open for the duration of the experiment and frequently switches back and forth between them every few minutes.

For each of the two scenarios, we take all of the runs, and show the average and standard deviation of the CPU utilization (Fig. 5), memory consumption

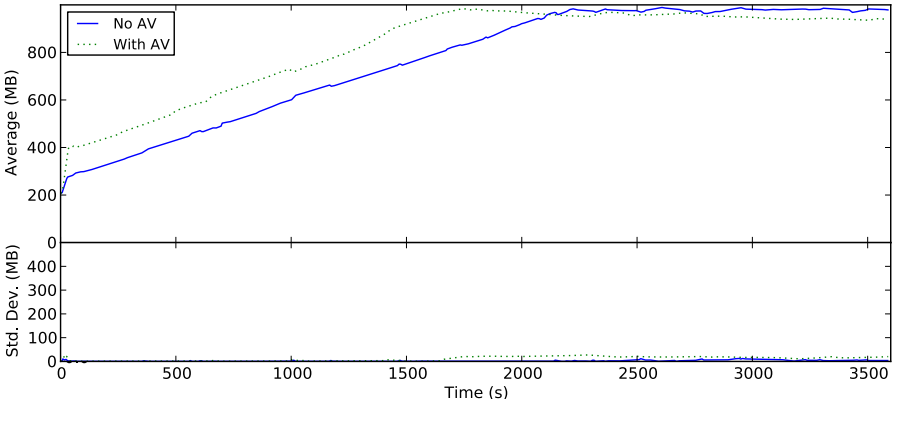

Fig. 6. Memory Consumption

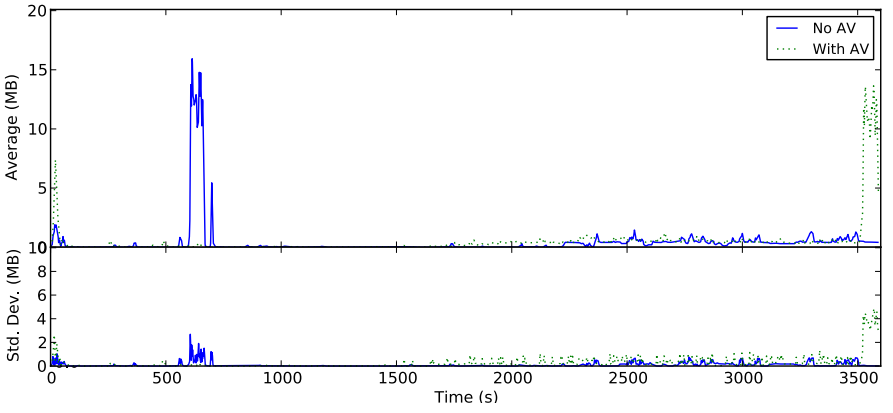

Fig. 7. Disk I/O

(Fig. 6), and disk input/output volume (Fig. 7) for the first hour of the experimental runs. The data was gathered in one second intervals using PCP. In order to make these plots more readable, we have performed a Gaussian smoothing over the data with a 5 second radius. In effect, this smooths out some of the jagged edges in the plot, making them more readable without changing them in any significant way.

Discussion. We see consistent spikes in both CPU load and disk I/O starting near 0 seconds when the user agent launches Outlook and Word, and again near 600 seconds when Internet Explorer is started. In Fig. 5, we see that the open source AV product consistently causes near 100% CPU use for a period of nearly 10 minutes. During this same period, the standard deviation of the CPU utilization is near zero, indicating that this behavior is very repeatable. Throughout Fig. 5 and Fig. 7, spikes in the average measurements are typically accompanied by smaller spikes in the standard deviations. However, we note that during periods of sustained activity, the standard deviation only spikes at the beginning and the end of the plateau in the mean. This pattern occurs in Fig. 5 at 600-1000 seconds, 1200-1400 seconds, and 1500-2500 seconds for the open source product and to a lesser extent from 1800-2000 and 2300-2500 seconds for the baseline case. This leads us to believe that much of the variance in these graphs is due to the inexact timing of our automation scripts.

Figures 5 and 6 show clear evidence of the system with open source anti-virus protection consuming more resources than the same system with no anti-virus, and formal statistical tests confirm our hypothesis with high confidence for these data series. In Fig. 7, overall, the anti-virus system does not appear to cause a statistically significantly increase in disk I/O loads relative to the baseline system. We are interested in whether these same results would hold for commercial anti-virus products, which may be developed with a greater focus on efficiency and performance. In the near future, we may expand the coverage of our test to include one or more commercial tools as well.

5 Conclusions and Future Work

We presented new techniques for driving ubiquitous, commercial-off-the-shelf Windows GUI applications in place of a human user on an isolated testbed network. Using statistical models of user behavior to generate workloads on the testbed hosts, together with dynamic application-level protocol replay techniques to emulate remote services like the World Wide Web, we can generate traffic on the testbed network that resembles real traffic to a very high degree of fidelity.

We demonstrated a small-scale experiment to show how these techniques help to enable configurable, repeatable tests involving client-side security tools, and we highlighted some challenges in achieving repeatable experimental results with such complex systems.

In the future, we plan to improve on our current techniques in a number of ways. First, we will collect data from a larger set of users and develop techniques for validating that the workloads and traffic induced on a testbed faithfully represent the environments they were modeled after. Second, we will develop actuators that require a smaller software footprint on the system under test, to further reduce the risk of test artifacts in experimental results. Finally, we plan to develop more robust techniques for dynamic application-level replay of web sites that make heavy use of JavaScript or other active content generation techniques.

Acknowledgments

The authors extend our sincerest thanks to the members of the MIT-LL Cyber Testing team who implemented much of the software described here and provided much helpful feedback on the experiments and the paper.

References

1. Barford, P., Landweber, L.: Bench-style network research in an Internet Instance Laboratory. ACM SIGCOMM Computer Communication Review 33(3), 21–26 (2003)
2. Peisert, S., Bishop, M.: How to Design Computer Security Experiments. In: Proceedings of the 5th World Conference on Information Security Education (WISE), pp. 141–148 (2007)
3. US Department of Homeland Security: A Roadmap for Cybersecurity Research. Technical report (November 2009), `www.cyber.st.dhs.gov/docs/DHS-Cybersecurity-Roadmap.pdf`
4. White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., Joglekar, A.: An integrated experimental environment for distributed systems and networks. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (December 2002)
5. Ricci, R., Duerig, J., Sanaga, P., Gebhardt, D., Hibler, M., Atkinson, K., Zhang, J., Kasera, S., Lepreau, J.: The Flexlab approach to realistic evaluation of networked systems. In: Proceedings of the 4th USENIX Symposium on Networked Systems Design & Implementation, pp. 201–214 (April 2007)

6. Vahdat, A., Yocum, K., Walsh, K., Mahadevan, P., Kostic, D., Chase, J., Becker, D.: Scalability and Accuracy in a Large-Scale Network Emulator. In: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (December 2002)
7. Bavier, A., Feamster, N., Huang, M., Peterson, L., Rexford, J.: VINI veritas: Realistic and controlled network experimentation. In: Proceedings of ACM SIGCOMM (September 2006)
8. Rossey, L.M., Cunningham, R.K., Fried, D.J., Rabek, J.C., Lippmann, R.P., Haines, J.W., Zissman, M.A.: LARIAT: Lincoln Adaptable Real-time Information Assurance Testbed. In: Proceedings of the IEEE Aerospace Conference (2002)
9. Provos, N., McNamee, D., Mavrommatis, P., Wang, K., Modadugu, N.: The Ghost in the Browser: Analysis of Web-based Malware. In: Proceedings of the First Workshop on Hot Topics in Understanding Botnets (HotBots 2007) (April 2007)
10. Fossi, M.: Symantec Internet Security Threat Report: Trends for 2008 (April 2009)
11. Deibert, R., Rohozinski, R.: Tracking GhostNet: Investigating a Cyber Espionage Network. Technical Report JR02-2009, Information Warfare Monitor (March 2009)
12. Nagaraja, S., Anderson, R.: The Snooping Dragon: Social-Malware Surveillance of the Tibetan Movement. Technical Report UCAM-CL-TR-746, University of Cambridge Computer Laboratory (March 2009)
13. Provos, N., Mavrommatis, P., Rajab, M., Monrose, F.: All Your iFrames Point to Us. In: Proceedings of the 17th USENIX Security Symposium (July 2008)
14. Pinheiro, E., Weber, W.D., Barroso, L.A.: Failure Trends in a Large Disk Drive Population. In: Proceedings of the 5th USENIX Conference on File and Storage Technologies (February 2007)
15. Lippmann, R.P., Fried, D.J., Graf, I., Haines, J.W., Kendall, K.R., McClung, D., Weber, D., Webster, S.E., Wyschogrod, D., Cunningham, R.K., Zissman, M.A.: Evaluating Intrusion Detection Systems: The 1998 DARPA Off-Line Intrusion Detection Evaluation. In: Proceedings of the 2000 DARPA Information Survivability Conference and Exposition (2000)
16. Lippmann, R., Haines, J.W., Fried, D.J., Korba, J., Das, K.: The 1999 DARPA Off-line Intrusion Detection Evaluation. Computer Networks 34(4), 279–595 (2000)
17. Yu, T., Fuller, B., Bannick, J., Rossey, L., Cunningham, R.: Integrated Environment Management for Information Operations Testbeds. In: Proceedings of the 2007 Workshop on Visualization for Computer Security (October 2007)
18. Benzel, T., Braden, R., Kim, D., Neuman, C., Joseph, A., Sklower, K., Ostrenga, R., Schwab, S.: Experience with DETER: A Testbed for Security Research. In: Proceedings of the 2nd International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TRIDENTCOM) (March 2006)
19. Boothe-Rabek, J.C.: WinNTGen: Creation of a Windows NT 5.0+ network traffic generator. Master's thesis, Massachusetts Institute of Technology (2003)
20. Garg, A., Vidyaraman, S., Upadhyaya, S., Kwiat, K.: USim: A User Behavior Simulation Framework for Training and Testing IDSes in GUI Based Systems. In: ANSS 2006: Proceedings of the 39th Annual Symposium on Simulation, Washington, DC, USA, pp. 196–203. IEEE Computer Society, Los Alamitos (2006)
21. Cui, W., Paxson, V., Weaver, N.C.: GQ: Realizing a System to Catch Worms in a Quarter Million Places. Technical Report TR-06-004, International Computer Science Institute (September 2006)
22. Cui, W., Paxson, V., Weaver, N.C., Katz, R.H.: Protocol-Independent Adaptive Replay of Application Dialog. In: Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS 2006) (February 2006)

23. Small, S., Mason, J., Monrose, F., Provos, N., Stubblefield, A.: To catch a predator: A natural language approach for eliciting malicious payloads. In: Proceedings of the 17th USENIX Security Symposium (August 2008)
24. Wang, K.: Using HoneyClients to Detect New Attacks. In: RECON Conference (June 2005)
25. Wang, Y.M., Beck, D., Jiang, X., Roussev, R., Verbowski, C., Chen, S., King, S.: Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In: Proceedings of the 13th Annual Symposium on Network and Distributed System Security (NDSS 2006) (February 2006)
26. Sanders, M.: autopy: A simple, cross-platform GUI automation toolkit for Python, `http://github.com/msanders/autopy`
27. Yeh, T., Chang, T.H., Miller, R.C.: Sikuli: Using GUI Screenshots for Search and Automation. In: Proceedings of the 22nd Symposium on User Interface Software and Technology (October 2009)
28. Kleek, M.V., Bernstein, M., Karger, D., Schraefel, M.C.: Getting to Know You Gradually: Personal Lifetime User Modeling (PLUM). Technical report, MIT CSAIL (April 2007)
29. Simpson, C.R., Reddy, D., Riley, G.F.: Empirical Models of TCP and UDP EndUser Network Trafc from NETI@home Data Analysis. In: 20th International Workshop on Principles of Advanced and Distributed Simulation (May 2006)
30. Kurz, C., Hlavacs, H., Kotsis, G.: Workload Generation by Modelling User Behavior in an ISP Subnet. In: Proceedings of the International Symposium on Telecommunications (August 2001)
31. tcpreplay by Aaron Turner, `http://tcpreplay.synfin.net/`
32. Hong, S.S., Wu, S.F.: On Interactive Internet Traffic Replay. In: Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (September 2006)
33. Sommers, J., Barford, P.: Self-configuring network traffic generation. In: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, pp. 68–81 (2004)
34. Cao, J., Cleveland, W.S., Gao, Y., Jeffay, K., Smith, F.D., Weigle, M.C.: Stochastic models for generating synthetic HTTP source traffic. In: INFOCOM (2004)
35. Weigle, M.C., Adurthi, P., Hernández-Campos, F., Jeffay, K., Smith, F.D.: Tmix: a tool for generating realistic TCP application workloads in ns-2. ACM SIGCOMM Computer Communication Review 36(3), 65–76 (2006)
36. Lan, K.C., Heidemann, J.: Rapid model parameterization from traffic measurements. ACM Transactions on Modeling and Computer Simulation (TOMACS) 12(3), 201–229 (2002)
37. Vishwanath, K.V., Vahdat, A.: Realistic and Responsive Network Traffic Generation. In: Proceedings of ACM SIGCOMM (September 2006)
38. Sommers, J., Yegneswaran, V., Barford, P.: Toward Comprehensive Trafc Generation for Online IDS Evaluation. Technical report, University of Wisconsin (2005)
39. Mutz, D., Vigna, G., Kemmerer, R.: An Experience Developing an IDS Stimulator for the Black-Box Testing of Network Intrusion Detection Systems. In: Proceedings of the Annual Computer Security Applications Conference (December 2003)
40. Kayacik, H.G., Zincir-Heywood, N.: Generating Representative Traffic for Intrusion Detection System Benchmarking. In: Proceedings of the 3rd Annual Communication Networks and Services Research Conference, pp. 112–117 (May 2005)
41. Sommers, J., Yegneswaran, V., Barford, P.: A framework for malicious workload generation. In: Proceedings of the 4th ACM SIGCOMM Conference on Internet Measurement, pp. 82–87 (2004)

42. Hunt, G., Brubacher, D.: Detours: Binary Interception of Win32 Functions. In: Third USENIX Windows NT Symposium (July 1999)
43. Klimt, B., Yang, Y.: Introducing the Enron Corpus. In: Proceedings of the First Conference on Email and Anti-Spam (CEAS) (July 2004)
44. Paxson, V., Floyd, S.: Wide Area Traffic: The Failure of Poisson Modeling. IEEE/ACM Transactions on Networking 3(3) (June 1995)
45. Matsumoto, M., Nishimura, T.: Mersenne Twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. ACM Transactions on Modelling and Computer Simulation 8(1), 3–30 (1998)
46. GINA: MSDN Windows Developer Center, http://msdn.microsoft.com/en-us/library/aa375457VS.85.aspx
47. Hibler, M., Ricci, R., Stoller, L., Duerig, J., Guruprasad, S., Stack, T., Webb, K., Lepreau, J.: Large-scale Virtualization in the Emulab Network Testbed. In: Proceedings of the 2008 USENIX Annual Technical Conference (June 2008)
48. Google, Inc.: Google search appliance, http://www.google.com/enterprise/search/gsa.html
49. osCommerce: Open Source E-Commerce Solutions, http://www.oscommerce.com/
50. DMOZ Open Directory Project, http://www.dmoz.org/
51. Yahoo! Directory, http://dir.yahoo.com/
52. Alexa Top Sites, http://www.alexa.com/topsites
53. AV-Comparatives e.V.: Anti-Virus Comparative Performance Test: Impact of Anti-Virus Software on System Performance (December 2009), http://www.av-comparatives.org/comparativesreviews/performance-tests
54. Warner, O.: What Really Slows Windows Down (September 2006), http://www.thepcspy.com/read/what_really_slows_windows_down
55. Chatterton, D., Gigante, M., Goodwin, M., Kavadias, T., Keronen, S., Knispel, J., McDonell, K., Matveev, M., Milewska, A., Moore, D., Muehlebach, H., Rayner, I., Scott, N., Shimmin, T., Schultz, T., Tuthill, B.: Performance Co-Pilot for IRIX Advanced User's and Administrator's Guide. 2.3 edn. SGI Technical Publications (2002), http://oss.sgi.com/projects/pcp/index.html
56. Timekeeping in VMware Virtual Machines, http://www.vmware.com/pdf/vmware_timekeeping.pdf