A Lightweight Drive-by Download Simulator

Matthew McDonald James Ong John Aycock* Heather Crawford
Department of Computer Science
University of Calgary
2500 University Drive NW
Calgary, Alberta, Canada T2N 1N4

Nathan Friess†
Lyryx Learning, Inc.
210 - 1422 Kensington Road NW
Calgary, Alberta, Canada T2N 3P9

Abstract

Teaching students about practical security can be a challenge in laboratories that are, by necessity, isolated from the Internet. We describe some preliminary work to address this problem for the topic of drive-by downloads. A previous system, the Spamulator, allowed students' real, non-contrived code to interact with a simulation of the Internet, simulating up to a million domains on a single machine. We extend the Spamulator to allow arbitrary drive-by downloads in a lightweight fashion, using a hybrid simulation technique to capture both local effects and global spread.

1 Introduction

Education is unquestionably a defense against some security threats. Use of the term "education" usually refers to *user* education, but it is equally important to consider the education of the next generation of defenders: computer science and engineering students.

A practical education of these students demands that they have hands-on experience with current forms of malware. (We leave this assertion general: the debate about whether students should work with existing malware or create their own malware is outside the scope of this paper.) For safety reasons, student experiments with malware should be conducted in an isolated laboratory environment with no access to the Internet or even the intranet outside the laboratory. This means that there is not a ready pool of willing victims and machines available to students. At the same time, many

*Corresponding author; email aycock@ucalgary.ca.
†Work done while at the University of Calgary.

current threats are large-scale and global in scope, like botnets. How can students be given effective, hands-on training with these threats within the confines of a secure laboratory?

The preliminary work described in this paper begins to address this problem. Building on a successful previous system, the Spamulator, we describe the system we have built to help teach students about drive-by downloads. Our system allows the students to write real drive-by download code, and uses a hybrid simulation technique that combines the effects of running the students' code with a global simulator that provides an Internet's worth of victims, all within the safety of the laboratory.

We begin with a discussion of related work in the next section. Section 3 describes the Spamulator. This is followed by details of our drive-by download simulation and the global simulator in Sections 4 and 5, respectively. Finally, we end with our conclusion and future work in Section 6.

2 Related Work

Gordon [10] makes a distinction between simulators for education and simulators for anti-virus testing, although the education referred to is clearly user education rather than the student audience our system is targeting. This distinction is echoed by Leszczyna et al. [12], whose MAlSim system is a Java-based multi-agent sytem rather than one that runs real code.

Liljenstam et al. [13] suggest modeling worm activity with a mixture of models, which from a very high level is similar to our hybrid approach. However, they are linking different models rather than real code execution and simulation. Weaver et al. take a different tack, trying to scale down the size of the Internet while still maintaining accuracy [20].

There are a number of examples of systems for sandboxing and scalable simulations. All of these are heavyweight and resource-hungry when compared to our system, though. For example, Ford and Cox [9] describe the Vx32 sandboxing system, complete with dynamic translation of code, making it anything but lightweight. Brumley et al. run code in a fully-emulated environment [6]. Potemkin's honeyfarm uses virtual machines on ten servers [19], and the Botnet Evaluation Environment uses over 100 PCs [5], orders of magnitude more computing power than our system needs. Aycock et al. [4] compares the Spamulator to various other system, including the Honeyd system that the related work by Filiol et al. [8] is based on.

General automatic detection of malware is of course within the purview of anti-virus software, and large portions of books have been written on the topic (e.g., [1, 17]). More specific analyses have been applied to detect things like time-based triggers [7] and spyware [11]. SpyProxy [14] captures a superset of what our exploit verifier can detect (or, for that matter, needs to detect) and is probably the closest work to ours.

3 Spamulator

We had previously developed a system called the Spamulator for use in a university course on spam and spyware [2]. The Spamulator and its applications in anti-spam research have been described elsewhere [4, 15], but we include a brief overview here for completeness before explaining how we have extended it to the harder problem of drive-by downloads.

The primary problem the Spamulator was meant to solve was the fact that there were few people to spam in an isolated security laboratory. The Spamulator simulates up to a million domains' worth of machines (i.e., potentially multiple machines per domain) on a single computer, effectively giving each student their own copy of the Internet. HTTP servers serve out web page hierarchies for each domain, where the web pages contain email addresses that may be harvested. A search engine and web directory, modeled after Google and Yahoo! Directory respectively, provide starting points for harvesting. SMTP servers and open proxies exist for sending spam to the harvested addresses.

From the students' point of view, they are simply interacting with the Internet. They can use unaltered Internet applications like Firefox to surf the simulated web pages, and they simply write normal networking code to interact with their copy of the Internet. There are no contrived limitations on their code, no special libraries to link, no restrictions on what programming language they can use. The simulation is complete enough that we have been able to run real bulk-mailing and address harvesting software successfully within the Spamulator.

The Spamulator accomplishes its task by selectively redirecting and rewriting TCP network packets. One or more ranges of IP addresses are flagged (using `iptables` and `libipq` on Linux, and `ipfw` and divert sockets on Mac OS X and FreeBSD; the latter version will be described here), and any packets destined for those IP addresses are sent instead to the Network Rerouting Daemon, or NeRD. NeRD will start a simulated server process for each new connection, rewriting the packets to reflect the actual destination IP and port number, once known; further packets belonging to the same connection are rewritten similarly. Return packet traffic is identified by a sentinel IP address (`127.0.0.2`) and those packets are also rewritten by NeRD.

Using the abstracted TCP packet shown in Figure 1, we illustrate this process in Figure 2 with an example of a new connection:

Figure 2a. The simulated IP range in this example is `10.0.0.0/8`. A packet destined for the IP address `10.0.0.1` is thus redirected to NeRD. The packet's SYN bit indicates whether or not it is a new connection; it would be set for this example.

Figure 2b. NeRD starts a simulated server process to handle the new connection. A pipe remains open between NeRD and the simulated server for communications.

Figure 2c. The simulated server listens for new connections.

Figure 2d. NeRD receives the port number from the simulated server along the pipe, indicating what port the simulated server is listening at.

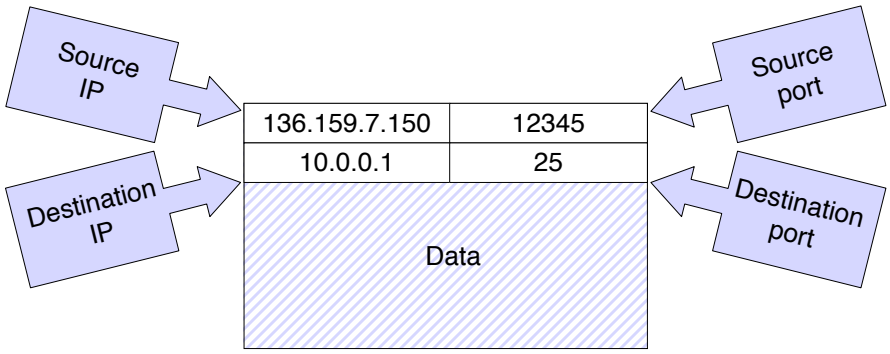

Figure 1: Abstracted TCP packet, sent from port 12345 on `136.159.7.150` to the SMTP port (port 25) on `10.0.0.1`

Figure 2e. NeRD notes the information for future packets and rewrites the initial packet accordingly, readdressing it to localhost (`127.0.0.1`), port 2112. The source IP address is changed to the sentinel address `127.0.0.2`. Finally, NeRD reinjects the packet and the kernel sends it to the listening simulated server process.

Subsequent packets from `136.159.7.150:12345` to `10.0.0.1:25` are rewritten similarly, but there is no need to start the simulated server anew. Return packets are rewritten as shown in Figure 3.

4 Drive-by Download Simulation

After its successful use in teaching spam and spyware, adapting the Spamulator for use with arbitrary malware was the obvious next step. Ideally, we would like to allow large-scale experimentation with computer worms in a safe environment. However, this was a much more challenging problem.

Recall that the Spamulator's primary use was address harvesting and spamming. It was also used for a spyware scenario, where a browser helper object steals online banking login information and posts it to a drop site. In both these applications, the students' code interacted with (simulated) network servers in a well-defined manner using established network protocols. At no time was the students' code running on the simulated servers.

In contrast, computer worms would require arbitrary code to run on the simulated machines, and on a large scale. Whether and how the worm propagates depends on the execution of the code. This negates the key elements of the Spamulator: that it be lightweight and run on a single computer. We cannot simply start up tens of thousands of virtual machines on a laptop, for instance.

Instead we turned our attention to arbitrary code that has a known spread and a known infection vector, drive-by downloads. The conceptual model we considered is

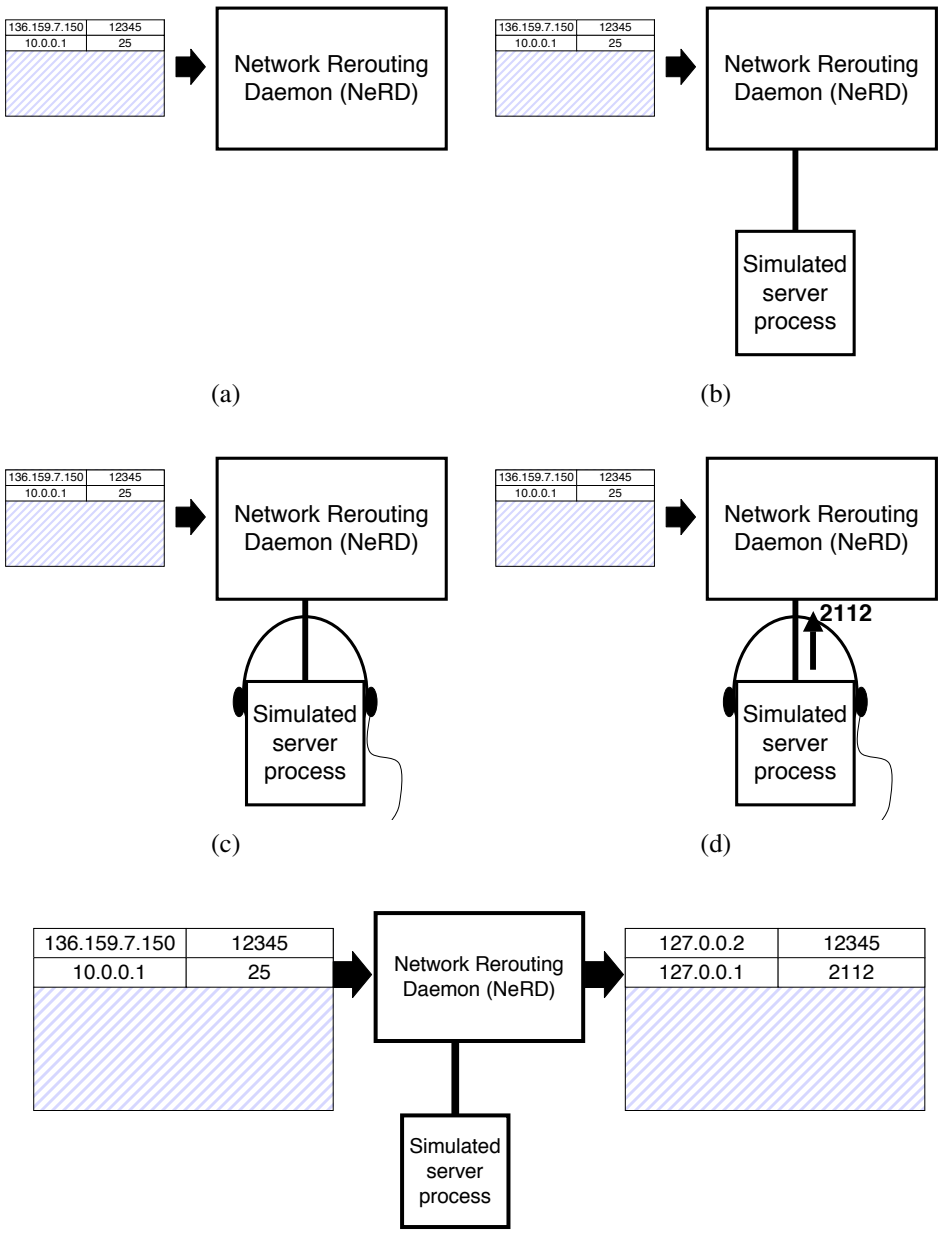

Figure 2: Packet rewriting for new connection

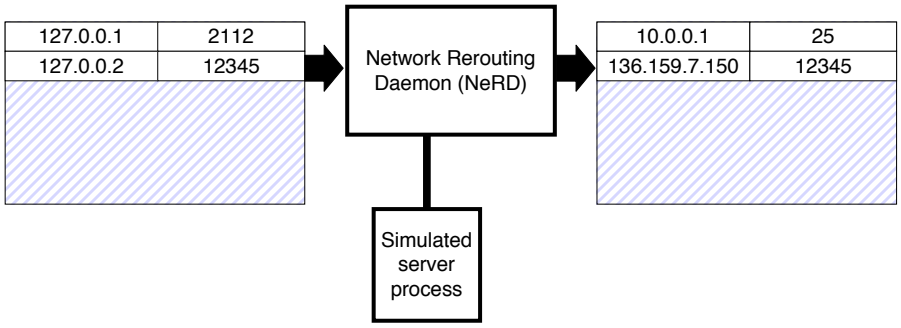

Figure 3: Return packet traffic rewriting

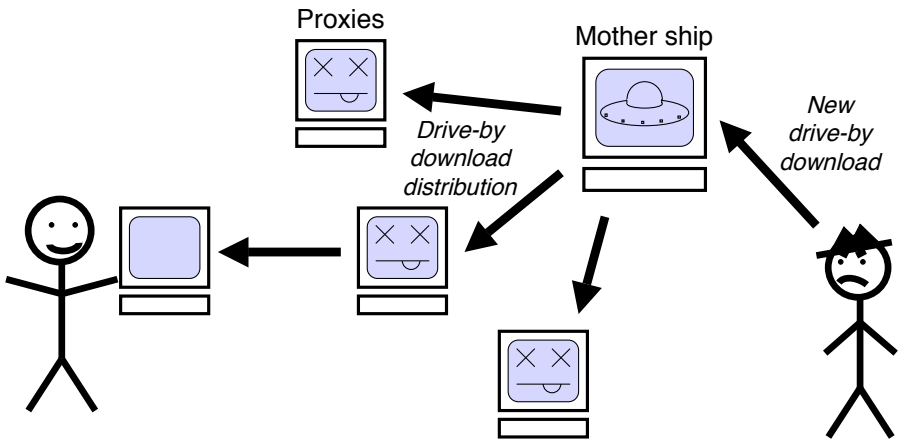

Figure 4: Drive-by download model

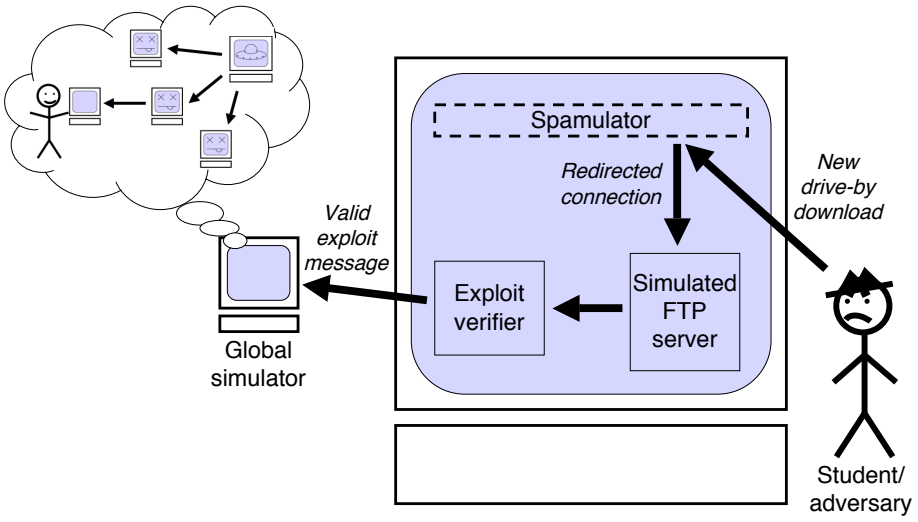

Figure 5: Drive-by download model, as implemented in the Spamulator

shown in Figure 4. An adversary has an established architecture for distribution of drive-by downloads, with a "mother ship" containing the exploit *du jour*, served out via proxies on compromised machines. A new drive-by download is uploaded to the mother ship by the adversary, then infects victims over time through proxies.

In our Spamulator implementation of this conceptual model (Figure 5), the student plays the role of the adversary. They develop a drive-by download exploit for Firefox, and infect victims in the following way:

1. The student/adversary uploads their drive-by download to the mother ship via FTP. The mother ship is actually a simulated FTP server.

2. The mother ship invokes a program, the exploit verifier, to automatically verify that the purported drive-by download is in fact a *bona fide* exploit. The exploit verifier is discussed in detail below. Because our lab has a physical limit on the number of users, as discussed in Section 5, one could argue that there are few enough exploits that they could be manually verified. However, automatic verification keeps a human out of the loop so that our system can run constantly and scale as necessary.

3. If the exploit is verified, the verifier sends a message to that effect to the global simulator.

4. The global simulator simulates the spread of the drive-by download to victims worldwide. The global simulator is the topic of the next section.

The conceptual model is thus implemented with a hybrid simulation technique, using both real code execution (during exploit verification) and traditional simulation (in the

```
<validcalls>
<syscall>
    <name>read</name>
    <invalidatt>
        <att>42</att>
    </invalidatt>
</syscall>
</validcalls>
```

(a)

```
def verifySyscall(sys_call, params):
    if sys_call == 'read':
        if params[0] == '42':
            return False
    return True
```

(b)

Figure 6: XML description file and generated (pseudo)code

global simulator). The Spamulator allows FTP connections to the mother ship and HTTP connections from a victim's browser to be redirected appropriately.

Technically, the simulated FTP server is a skeletal implementation that supports the `put` and `quit` commands, which is sufficient for the mother ship. The exploit verifier is the more comparatively sophisticated part of our system.

4.1 Exploit Verifier

The exploit verifier's design has arisen in the process of addressing three practical engineering constraints. First, we only need to run a single process – Firefox – to verify the exploit; a full operating system emulation would be overkill and involve too much overhead. Second, we wanted a lightweight solution that would not involve massive amounts of code. Third, the exploit verifier needed to be low maintenance and not require delicate changes to some third-party emulator that would need to be redone whenever a new version of the emulator appeared.

The only existing package we found that had the ability to run single processes was QEMU. However, this ability is currently restricted to Linux and Mac OS X [16], and our system had to run on FreeBSD due to laboratory constraints. Furthermore, having to modify the third-party QEMU software would violate our third design constraint, maintenance.

Because we are interested in whether or not the exploit actually runs, we are concerned with the dynamic behavior of Firefox. For this reason, we have chosen an appropriate lightweight mechanism: the exploit verifier monitors the execution of Firefox by tracing the system calls that Firefox makes; this can be done in user mode with the `ptrace` system call. Any appearance of a "bad" behavior from the exploit causes the exploit verifier to halt Firefox and signal to the global simulator that the exploit is valid. (Thinking in terms of "good" and "bad" behaviors is intuitively appealing, but somewhat misleading, as the exploit verifier is really just looking for specified behaviors.)

Behaviors to watch for are specified using an XML description file that lists system calls and their arguments; combinations of the two are flagged as either valid or invalid. The XML descriptions are used to generate code that performs the matching against the system call traces, allowing us to gradually improve the matching and its speed

over time. Figure 6a gives a simple XML description that disallows the `read` system call with a first argument of `42`, and the corresponding generated code is in Figure 6b. (We have shown pseudocode for clarity.) Wildcards and negation can also be specified in the XML descriptions.

In addition to being able to look for specific system calls as Firefox executes, the exploit verifier is also able to monitor system call frequency and memory usage for anomalous activity. Despite this, the exploit verifier is the part of the system that could benefit the most from enhancement. Some obvious extensions would allow specification of sequences of system calls, and to skip the system calls at startup that occur before an exploit could possibly affect Firefox – we conjecture that this latter extension will reduce the risk of false positives.

Another option available to us, being a tool for teaching, is to steer students in a direction that is pedagocially acceptable in that it teaches students the right concepts, but is easier to detect by the exploit verifier. For example, we could deliberately add an exploitable bug into Firefox, and monitor for activity at that address only. A related idea is to monitor for a specific class of attacks, like checking the address of the system call to see if it is located in the stack, and direct students to mount a stack-smashing attack for their drive-by download. A less desirable alternative would be to supply the students with partial code containing a system call that would act as a sentinel to the exploit verifier, but this option would be both contrived and limiting. Letting shellcode progress far enough to produce some externally-visible effect, like connecting to a well-known IP address and port, is another option.

Regardless of the method used by the exploit verifier, however, it passes on a message to the global simulator once a valid exploit has been detected.

5 Global Simulator and Visualization

The global simulator simulates the worldwide effect of a new drive-by download being distributed from the mother ship through proxies on compromised machines, as it infects new victims' machines. A related component, the visualizer, displays the ongoing simulation to give students feedback about their drive-by download. In theory the global simulator and the visualizer could run on each student's machine, but by moving them onto a different physical machine we can show all students' aggregate activity in the laboratory on a common display.

As mentioned, the global simulator receives messages from the exploit verifier. These messages signal the upload of a new drive-by download, and also contain the username to whom the drive-by download should be attributed. We assume that each user has at most one drive-by download active at any given time, and that there are a maximum of eight users. This latter constraint is due to the fact that our physical laboratory has eight computers for students [3], but this limit turns out to be very helpful for visualization as we describe below.

The simulator has a fixed set of proxies (currently three), and a larger fixed set of 126 cities where infections may occur. Each city is mapped into its world map coordinates, and also its time zone. Because the drive-by download infections are the result of human activity, and human activity tends to be greater during daylight

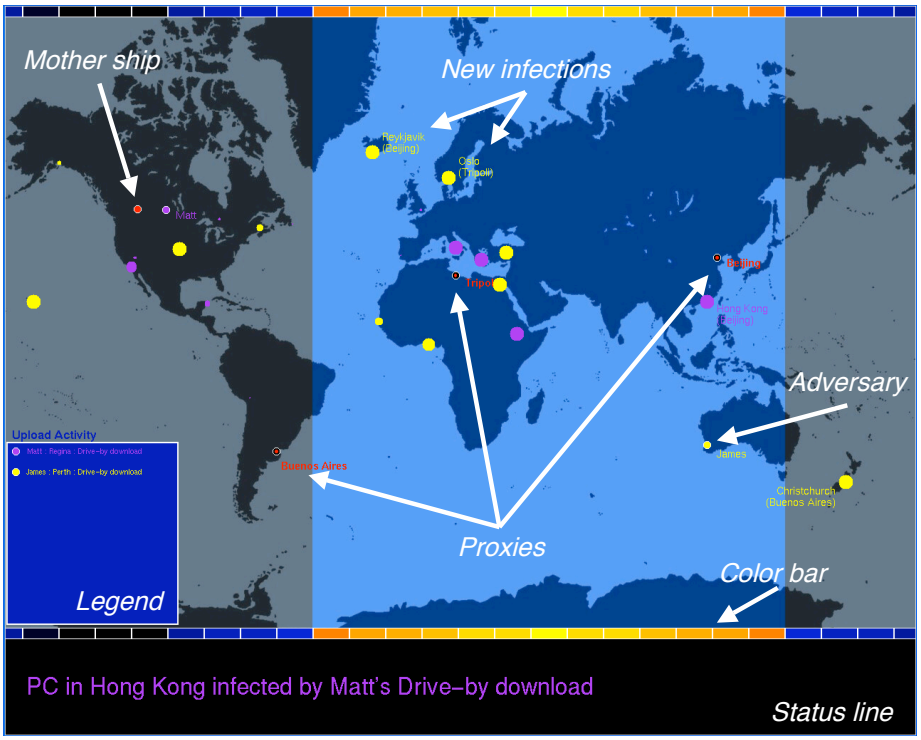

Figure 7: Visualizer screenshot

hours, we simulate the effect of the diurnal cycle. The global simulator biases the number of new infections such that cities whose time zones are in daylight have more infections. Other elements of the simulation, like the number of new infections per time step, are driven by a pseudorandom number generator operating within various tunable parameters.

The visualizer (Figure 7) displays the global simulator's activity on a Mercator projection [18] of the world. Daylight is shown using two indicators that move right to left appropriately as daylight cycles around the globe. First, a shaded, semi-transparent area darkens nighttime areas. Second, color bars across the top and bottom show gradations of light with different shades of yellow and blue: yellow for daytime (7am to 6pm) with lighter shades being closer to noon, and blue for nighttime (7pm to 10pm and 3am to 6am) with darker shades closer to midnight. Black bars are used between 11pm to 2am. The linear daylight pattern is an approximation of actual daylight, of course, but it is sufficient for our simulation's purposes.

The display continually shows the mother ship (located in Calgary) and the proxies. Students who have an active drive-by download are shown in a fixed location; because there are at most eight of them, they are each assigned a distinct color that is shown in the legend at the lower left of the display. Infections from a particular drive-by download are shown using the corresponding color, which significantly reduces the

clutter from labels. New infections are temporarily labeled with the city name and (in parentheses) the proxy through which they became infected. Finally, a status line at the bottom gives details of the last simulation event.

6 Conclusion and Future Work

While our system needs to be refined slightly before a full deployment in the laboratory, our preliminary running system suggests that it is possible to field a training tool that allows students to use real drive-by download code and see a global impact, all from within the safety of a secure laboratory environment.

We are currently looking at making the exploit verifier more powerful in terms of its ability to detect exploits automatically, while still retaining its lightweight aspect. We are also considering a number of ways to extend the Spamulator for both teaching and research. Botnets are an obvious extension, as are certain web-based attacks like cross-site scripting exploits. Finally, we continue to pursue the Holy Grail of single-machine simulation: large-scale computer worm spread using real code.

Acknowledgment

Some of the authors were supported in part by grants from the Natural Sciences and Engineering Research Council of Canada. This work was supported in part by NSERC ISSNet, the Internetworked Systems Security Network.

References

[1] J. Aycock. *Computer Viruses and Malware*. Springer, 2006.

[2] J. Aycock. Teaching spam and spyware at the University of C@1g4ry. In *Third Conference on Email and Anti-Spam*, pages 137–141, 2006. Short paper.

[3] J. Aycock and K. Barker. Creating a secure computer virus laboratory. In *13th Annual EICAR Conference*, 2004. 13pp.

[4] J. Aycock, H. Crawford, and R. deGraaf. Spamulator: The Internet on a laptop. In *13th Annual ACM SIGCSE Conference on Innovation and Technology in Computer Science Education*, pages 142–147, 2008.

[5] P. Barford and M. Blodgett. Toward botnet mesocosms. In *First Workshop on Hot Topics in Understanding Botnets (HotBots)*, 2007.

[6] D. Brumley, J. Newsome, and D. Song. *Sting: An End-to-End Self-Healing System for Defending against Internet Worms*, chapter 7, pages 147–170. 2007.

[7] J. R. Crandall, G. Wassermann, D. A. S. de Oliveira, Z. Su, S. F. Wu, and F. T. Chong. Temporal search: Detecting hidden malware timebombs with virtual machines. In *Proceedings of the 12th International Conference on Architectural*

Support for Programming Languages and Operating Systems (ASPLOS), pages 25–36, 2006.

[8] E. Filiol, E. Franc, A. Gubbioli, B. Moquet, and G. Roblot. Combinatorial optimisation of worm propagation on an unknown network. *Proceedings of World Academy of Science, Engineering and Technology*, 23:373–379, 2007.

[9] B. Ford and R. Cox. Vx32: Lightweight user-level sandboxing on the x86. In *USENIX Annual Technical Conference*, pages 293–306, 2008.

[10] S. Gordon. Are good virus simulators still a bad idea? *Network Security*, pages 7–13, Sept. 1996.

[11] E. Kirda, C. Kruegel, G. Banks, G. Vigna, and R. A. Kemmerer. Behavior-based spyware detection. In *15th USENIX Security Symposium*, pages 273–288, 2006.

[12] R. Leszczyna, I. N. Fovino, and M. Masera. Simulating malware with MAlSim. *Journal in Computer Virology*. To appear; was in EICAR 2008.

[13] M. Liljenstam, Y. Yuan, B. J. Premore, and D. Nicol. A mixed abstraction level simulation model of large-scale Internet worm infestations. In *Proceedings of the 10th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems (MASCOTS)*, pages 109–116, 2002.

[14] A. Moshchuk, T. Bragin, D. Deville, S. D. Gribble, and H. M. Levy. SpyProxy: Execution-based detection of malicious web content. In *16th USENIX Security Symposium*, pages 27–42, 2007.

[15] M. Nielsen, D. Bertram, S. Pun, J. Aycock, and N. Friess. Global-scale anti-spam testing in your own back yard. In *Fifth Conference on Email and Anti-Spam*, 2008. 8pp.

[16] QEMU. QEMU emulator user documentation. http://bellard.org/qemu/qemu-doc.html. Last retrieved 17 December 2008.

[17] P. Szor. *The Art of Computer Virus Research and Defense*. Addison-Wesley, 2005.

[18] United States Geological Survey. Map projections – a working manual. U.S. Geological Survey Professional Paper 1395, 1987.

[19] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. C. Snoeren, G. M. Voelker, and S. Savage. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, pages 148–162, 2005.

[20] N. Weaver, I. Hamadeh, G. Kesidis, and V. Paxson. Preliminary results using scale-down to explore worm dynamics. In *Proceedings of the 2004 ACM Workshop on Rapid Malcode (WORM)*, pages 65–72, 2004.