

# Virtual Playgrounds for Worm Behavior Investigation

Xuxian Jiang<sup>1</sup>, Dongyan Xu<sup>1</sup>, Helen J. Wang<sup>2</sup>, and Eugene H. Spafford<sup>1</sup>

<sup>1</sup> CERIAS and Department of Computer Science,  
Purdue University, West Lafayette, IN 47907  
{jiangx, dxu, spaf}@cs.purdue.edu

<sup>2</sup> Microsoft Research Redmond, WA 98052  
helenw@microsoft.com

**Abstract.** To detect and defend against Internet worms, researchers have long hoped to have a safe convenient environment to unleash and run real-world worms for close observation of their infection, damage, and propagation. However, major challenges exist in realizing such “worm playgrounds”, including the playgrounds’ *fidelity*, *confinement*, *scalability*, as well as *convenience* in worm experiments. In this paper, we present a *virtualization-based* platform to create virtual worm playgrounds, called *vGrounds*, on top of a physical infrastructure. A *vGround* is an all-software virtual environment dynamically created for a worm attack. It has realistic end-hosts and network entities, all realized as virtual machines (VMs) and confined in a virtual network (VN). The salient features of *vGround* include: (1) *high fidelity* supporting real worm codes exploiting real vulnerable services, (2) *strict confinement* making the real Internet totally invisible and unreachable from inside a *vGround*, (3) *high resource efficiency* achieving sufficiently large scale of worm experiments, and (4) *flexible and efficient worm experiment control* enabling fast (tens of seconds) and automatic generation, re-installation, and final tear-down of *vGrounds*. Our experiments with real-world worms (including *multi-vector worms* and *polymorphic worms*) have successfully exhibited their probing and propagation patterns, exploitation steps, and malicious payloads, demonstrating the value of *vGrounds* for worm detection and defense research.

**Keywords:** Internet Worms, Intrusion Observation and Analysis, Destructive Experiments.

## 1 Introduction

In recent worm detection and defense research, we have witnessed increasingly novel features of emerging worms [41] in their infection and propagation strategies. Examples are polymorphic appearance [34], multi-vector infection [15], self-destruction [23], and intelligent payloads such as self-organized attack networks [18] or mass-mailing capability [21]. In order to understand key aspects of worm behavior such as probing, exploitation, propagation, and malicious payloads, researchers have long hoped to have a safe and convenient environment to run and observe real-world worms. Such a “worm playground” environment is useful not only in accessing the impact of worm intrusion and propagation, but also in testing worm detection and defense mechanisms [46, 42, 35, 37].

Despite its usefulness, there are difficulties in realizing a worm playground. Major challenges include the playground’s *fidelity*, *confinement*, *scalability*, *resource efficiency*, as well as the *convenience in worm experiment setup and control*. Currently, a common practice is to deploy a dedicated testbed with a large number of physical machines, and to use these machines as nodes in the worm playground. However, this approach may not effectively address the above challenges, for the following reasons: (1) Due to the coarse granularity (one physical host) of playground entities, the scale of a worm playground is constrained by the number of physical hosts, affecting the full exhibition of worm propagation behavior; (2) By nature, worm experiments are *destructive*. With physical hosts as playground nodes, it is a time-consuming and error-prone manual task for worm researchers to re-install, re-configure, and reboot worm-infected hosts between experiment runs; and (3) Using physical hosts for worm tests may lead to security risk and impact leakage, because the hosts may connect to machines *outside* the playground. However, if we make the testbed a physically-disconnected “island”, the testbed will no longer be share-able to remote researchers.

The contribution of our work is the design, implementation, and evaluation of a *virtualization-based* platform to quickly create safe virtual worm playgrounds called *vGrounds*, on top of general-purpose infrastructures. Our vGround platform can be readily used to analyze Linux worms, which represent a non-negligible source of insecurity especially with the rise of popularity of Linux in servers’ market. Though the current prototype does not support Windows-based worms, our design principles and concepts can also be applied to build Windows-based vGrounds.

The vGround platform can conveniently turn a physical infrastructure into a base to host vGrounds. An infrastructure can be a single physical machine, a local cluster, or a multi-domain overlay infrastructure such as PlanetLab [7]. A vGround is an all-software virtual environment with realistic end-hosts and network entities, all realized as virtual machines (VMs). Furthermore, a virtual network (VN) connects these VMs and *confines* worm traffic within the vGround. The salient features of vGround include:

- *High fidelity*. By running real-world OS, application, and networking software, a vGround allows *real* worm code to propagate as in the real Internet. Our full-system virtualization approach achieves the fidelity that leads to more opportunities to capture nuances, tricks, and variations of worms, compared with simulation-based approaches [39]. For example, one of our vGround-based experiments *identified a misstatement in a well-known worm bulletin*<sup>1</sup>.
- *Strict confinement*. Under our VM and VN (virtual network) technologies, the real Internet is totally invisible (unaddressable) from inside a vGround, preventing the leakage of negative impact caused by worm infection, propagation, and malicious payloads [16, 23] into the underlying infrastructure and cascadingly, the rest of the Internet. Furthermore, the damages caused by a worm only affect the virtual entities and components in one vGround and therefore do *not* affect other vGrounds running on the same infrastructure.
- *Flexible and efficient worm experiment control*. Due to the all-software nature of vGrounds, the instantiation, re-installation, and final tear-down of a vGround are

---

<sup>1</sup> The misstatement is now fixed and the authors have agreed not to disclose the details.

both fast and automatic, saving worm researchers both time and labor. For example, in our Lion worm experiment, it only takes 60, 90, and 10 seconds, respectively, to generate, bootstrap, and tear-down the vGround with 2000 virtual nodes. Such efficiency is essential when performing *multiple runs of a destructive experiment*. These operations can take hours or even days if the same experiment is performed directly on physical hosts. More importantly, the operations can be started by the researchers *without* the administrator privilege of the underlying infrastructure.

- *High resource efficiency.* Because of the scalability of our virtualization techniques, the scale of a vGround can be magnitudes larger than the number of physical machines in the infrastructure. In our current implementation, one physical host can support several *hundred* VMs. For example, we have tested the propagation of Lion worms [16] in a vGround with 2000 virtual end hosts, based on 10 physical nodes in a Linux cluster.

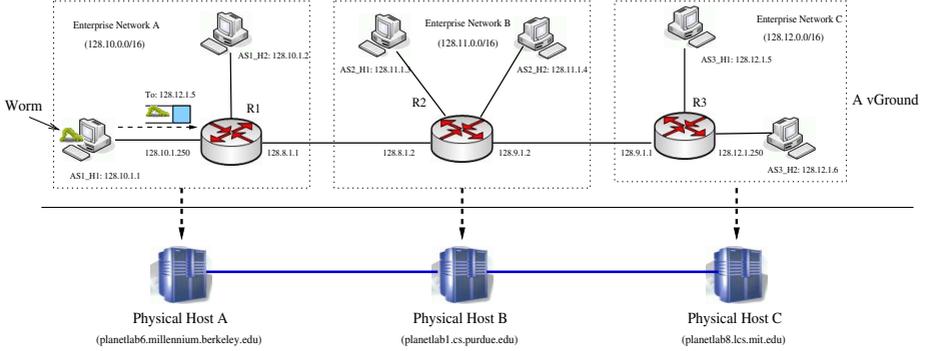
However, we would like to point out that although such scalability is effective in exposing worm propagation strategies based on our limited physical resources (Section 4), it is *not* comparable to the scale achieved by worm simulations. Having different focuses and experiment purposes, vGround is more suitable for analyzing detailed worm actions and damages, while the simulation-based approach is better for modeling worm propagation under Internet scale and topology. Also, lacking realistic background computation and traffic load, current vGrounds are *not appropriate for accurate quantitative modeling of worms*.

We are not aware of similar worm playground platforms with all the above features that are widely deployable on general-purpose infrastructures. We have successfully run real worms, including multi-vector worms and polymorphic worms, in vGrounds on our *desktops*, *local clusters*, and *PlanetLab*. Our experiments are able to fully exhibit the worms' probing and propagation patterns, exploitation attempts, and malicious payloads, demonstrating the value of vGrounds in worm detection and defense research.

The rest of this paper is organized as follows: Section 2 provides an overview of the vGround approach. The detailed design is presented in Section 3. Section 4 demonstrates the effectiveness of vGround using our experiments with several real-world worms. A discussion on its limitations and extensions is presented in Section 5. Related works are discussed in Section 6. Finally, Section 7 concludes this paper.

## 2 The vGround Approach

A vGround is a virtualization-based self-confined worm playground where not only each entity, including an end host, a firewall, a router, and even a network cable, is fully virtualized, but also every communication traffic is strictly confined within. Due to its virtualization-based nature and associated self-confinement property, a vGround can be safely created on a wide range of general-purpose infrastructures, including regular desktops, local clusters, and even wide-area shared infrastructures such as PlanetLab. For example, Figure 1 shows a simple vGround (the vGrounds in our worm experiments are much larger in scale) which is created on top of three PlanetLab hosts A, B, and C.



**Fig. 1.** A PlanetLab-based vGround for worm experiment

The vGround includes three virtual enterprise networks connected by three virtual routers ( $R1$ ,  $R2$ , and  $R3$ ). Within the vGround, the “seed” worm node ( $AS1\_H1$  in network A 128.10.0.0/16) is starting to infect other nodes running vulnerable services. Note that a vGround essentially appears as a virtual Internet whose network address assignment can be totally orthogonal to that of the real Internet. Furthermore, multiple simultaneously running vGrounds can safely overlap their address space without affecting each other as one vGround is completely invisible to another vGround.

Using a vGround specification language, a worm researcher will be able to specify the worm experiment setup in a vGround, including software systems and services, IP addresses, and routing information of virtual nodes (i.e. virtual end hosts and routers). Given the specification, the vGround platform will perform *automatic* vGround instantiation, bootstrapping, and clean-up. In a typical worm experiment, multiple runs are often needed as each different run is configured with a different parameter setting (e.g., different worm signatures [8, 1] and different traffic throttling thresholds[46]). However, because of the worm’s destructive behavior, the vGround will be completely unusable after each run and need to be re-installed. *The vGround platform is especially efficient in supporting such an iterative worm experiment workflow.*

## 2.1 Key vGround Techniques

**Existing full-system virtualization** is adopted to achieve high *fidelity* of vGrounds. Worms infect machines by remotely exploiting certain vulnerabilities in OS or application services (e.g., BIND, Sendmail, DNS). Therefore, the vulnerabilities provided by vGrounds should be the same as those in real software systems. As such, vGround can not only be leveraged for experimenting worms propagating via known vulnerabilities, but also be useful for discovering worms exploiting *unknown vulnerabilities*, of which worm simulations are *not* capable.

There exist various VM technologies that enable full-system virtualization. Examples include Virtual PC [12], VMware [13], Denali [49], Xen [26], and User-Mode Linux (UML) [30]. The differences in their implementations lead to different levels of cost, deployability and configurability: VMware and similarly Virtual PC require

several loadable kernel modules for virtualizing underlying physical resources; Xen and Denali “paravirtualize” physical resources by running *in place of* host OS; and UML is mainly a *user-level* implementation through system call virtualization. We choose UML in the current vGround implementation so that the deployment of vGround does not require the root privilege of the shared infrastructure. As a result, current vGround prototype can be widely deployed in most Linux-based systems (including PlanetLab). However, we would like to point out that the original UML itself is *not* able to satisfy the vGround needs. As described next, *we have developed new extensions to UML.*

**New network virtualization techniques** are developed to achieve vGround *confinement*. Simply running a worm experiment in a number of VMs *will not* confine the worm traffic just within these VMs and thus prevent potential worm “leakage”. Although existing UML implementation does have some support for virtual networking, it is still not capable of organizing different VMs into an *isolated* virtual topology. In particular, when the underlying shared infrastructure spans multiple physical domains, additional VPN softwares are needed to create the illusion of the virtual Internet. However, there are two notable weaknesses: (1) a VPN does not hide the existence of the underlying physical hosts and their network connections, which fails to meet the strict confinement requirement; (2) a VPN usually needs to be statically/manually configured as it requires the root privilege to manipulate the routing table, which fails to meet the flexible experiment control requirement. As our solution, we have developed a link-layer network virtualization technique to create a VN for VMs in a vGround. The VN reliably intercepts the traffic at the link-layer and is thus able to constrain both the topology and volume of traffic generated by the VMs. Such a VN essentially enables the illusion as a “virtual Internet” (though with a smaller scale) with its own IP address space and router infrastructure. More importantly, the VN and the real Internet are, by nature of our VN implementation, *mutually un-addressable*.

**New optimization techniques** are developed to improve vGround *scalability, efficiency, and flexibility*. To increase the number of VMs that can be supported in one physical host, the resource consumption of each individual VM should be conserved. For example, a full-system image of Red-Hat 9.0/7.2 requires approximately 1G/700M disk space. For a vGround of 100 VMs, a naive approach would require at least 100G/70G disk space. Our optimization techniques exploit the fact that a large portion of the VM images is the *same* and can be shared among the VMs. Furthermore, some services, libraries, and software packages in the VM image are *not* relevant to the worm being tested, and could therefore be safely removed. We also develop a *new method* to safely and efficiently generate VM images in each physical host (Section 3.4). Finally, a *new technique is being developed to enable worm-driven vGround growth*: new virtual nodes/subnets can be added to the vGround at runtime in reaction to a worm’s infection intent.

## 2.2 Advanced vGround User Configurability

The vGround platform provides a vGround specification language to worm researchers. There are two major types of entities - *network* and *virtual node*, in the vGround

specification language. A *network* is the medium of communication among *virtual nodes*. A virtual node can be an end-host, a router, a firewall, or an IDS system and it has one or more network interface cards (NICs) - each with an IP addresses. In addition, the virtual nodes are properly connected using proper routing mechanisms. Currently, the vGround platform supports RIP, OSPF, and BGP protocols.

In order to conveniently specify and efficiently generate various system images, the language defines the following notions: (1) A *system template* contains the basic VM system image which is *common* among multiple virtual nodes. If a virtual node is derived from a system template, the node will inherit all the capabilities specified in the system template. The definition of system template is motivated by the observation that most end-hosts to be victimized by a certain worm look quite similar from the worm’s perspective. (2) A *cluster* of nodes is the group of nodes located in the same subnet. The user may specify that they inherit from the same system template, with their IP addresses sharing the same subnet prefix.

```

project Planetlab-Worm
template slapper {
  image slapper.ext2
  cow enabled
  startup {
    /etc/rc.d/init.d/httpd start
  }
}
template router {
  image router.ext2
  routing ospf
  startup {
    /etc/rc.d/init.d/ospfd start
  }
}
router R1 {
  superclass router
  network eth0 {
    switch AS1_lan1
    address 128.10.1.250/24
  }
  network eth1 {
    switch AS1_AS2
    address 128.8.1.1/24
  }
}
switch AS1_lan1 {
  unix_sock sock/as1_lan1
  host planetlab6.millennium.berkeley.edu
}
switch AS1_AS2 {
  udp_sock 1500
  host planetlab6.millennium.berkeley.edu
}
node AS1_H1 {
  superclass slapper
  network eth0 {
    switch AS1_lan1
    address 128.10.1.1/24
    gateway 128.10.1.250
  }
}
node AS1_H2 {
  superclass slapper
  network eth0 {
    switch AS1_lan1
    address 128.10.1.2/24
    gateway 128.10.1.250
  }
}
switch AS2_lan1 {
  unix_sock sock/as2_lan1
  host planetlab1.cs.purdue.edu
}
switch AS2_AS3 {
  udp_sock 1500
  host planetlab1.cs.purdue.edu
}
node AS2_H1 {
  superclass slapper
  network eth0 {
    switch AS2_lan1
    address 128.11.1.3/24
    gateway 128.11.1.250
  }
}
node AS2_H2 {
  superclass slapper
  network eth0 {
    switch AS2_lan1
    address 128.11.1.4/24
    gateway 128.11.1.250
  }
}
switch AS3_lan1 {
  unix_sock sock/as3_lan1
  host planetlab8.ics.mit.edu
}
router R2 {
  superclass router
  network eth0 {
    switch AS2_lan1
    address 128.11.1.250/24
  }
  network eth1 {
    switch AS1_AS2
    address 128.8.1.2/24
  }
  network eth2 {
    switch AS2_AS3
    address 128.9.1.2/24
  }
}
router R3 {
  superclass router
  network eth0 {
    switch AS3_lan1
    address 128.12.1.250/24
  }
  network eth1 {
    switch AS2_AS3
    address 128.9.1.1/24
  }
}
node AS3_H1 {
  superclass slapper
  network eth0 {
    switch AS3_lan1
    address 128.12.1.5/24
    gateway 128.12.1.250
  }
}
node AS3_H2 {
  superclass slapper
  network eth0 {
    switch AS3_lan1
    address 128.12.1.6/24
    gateway 128.12.1.250
  }
}

```

Fig. 2. A sample vGround specification

As an example, Figure 2 shows the specification for the vGround in Figure 1. The keyword *template* indicates the system template used to generate other images files. For example, the image *slapper.ext2* is used to generate the images of the following end-hosts: *AS1\_H1*, *AS1\_H2*, *AS2\_H1*, *AS2\_H2*, *AS3\_H1*, and *AS3\_H2*; while the image *router.ext2* is used to generate the images of routers *R1*, *R2*, and *R3*. The keyword *switch* indicates the creation of a *network* connecting various virtual nodes. The internal keywords *unix\_sock* and *udp\_sock* indicate different network virtualization techniques based on UNIX and INET-4 sockets, respectively. Note that the keyword *cluster* is not used in this example. However, for a large-scale vGround, it is more convenient to use *cluster* to specify a subnet, which has a large number of end-hosts of similar configuration.

After a vGround is created, the vGround platform also provides a collection of toolkits to unleash the worm, collect worm infection traces, monitor worm propagation status, and re-install or tear-down the vGround. More details will be described in Sections 3 and 4.

### 3 Design Details

#### 3.1 Full-System Virtualization

The vGround platform leverages UML, an open-source VM implementation where the guest OS runs directly in the unmodified *user space* of the host OS. Processes within a UML-based VM are executed in the VM in exactly the same way as they are executed in a native Linux machine. Leveraging the capability of *ptrace*, a special process is created to intercept the system calls made by any process in the UML VM, and redirect them to the guest OS kernel. Through system call interception, UML is able to virtualize various resources such as memory, networks, and other “physical” peripheral devices. An in-depth analysis of UML is beyond the scope of this paper and interested readers are referred to [30].

For worm experiments, it is interesting to note that in earlier implementation of UML termed as the “tt mode”[30], the UML guest-OS kernel needs to be present at the last 0.5G of ptraced process address space and is *writable* by default. Such placement *prevents* certain worms from exploiting stack-based overflows and therefore limits applicability of vGrounds. In addition, the “write” permission incurs security risk. The recent version of UML implements the “skas mode” [30], by which the tracing process acts as a kernel-level thread, and does not impose such restriction or risk. In fact, this explains why certain worms like *Lion* cannot successfully propagate in vGrounds on top of PlanetLab, as the OS kernels of PlanetLab hosts do not usually support the “skas” mode.

#### 3.2 Link-Layer Network Virtualization

Figure 3 illustrates the link-layer network virtualization technique (marked within a dotted rectangle) developed for the vGround purpose. It involves three different entities: *virtual NIC*, *virtual switch*, and *virtual cable*, reflecting the corresponding physical counterparts. The virtual switch, implemented as a regular server daemon, will receive the connection requests from other virtual NICs. Each successful connection essentially acts as a virtual cable. The virtual NIC is largely based on the original UML implementation with certain extensions to communicate with remote virtual switch daemons. We would like to point out that these entities are link-layer “devices”, which

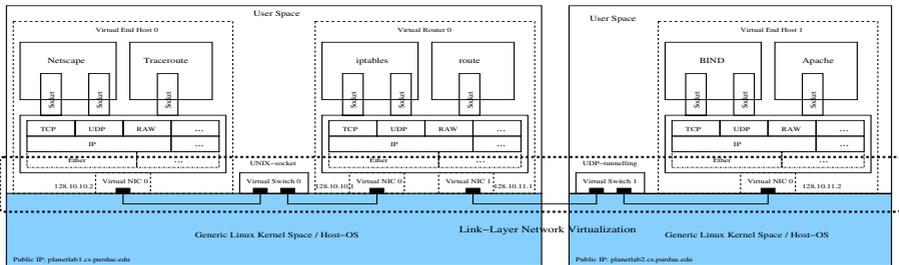


Fig. 3. Illustration of link-layer network virtualization in vGround

```
[root@AS1_H1 /root]#traceroute -n AS3_H2
traceroute to AS3_H2 (128.12.1.6), 30 hops max, 40 byte packets
 1  128.10.1.250  2.342 ms  3.694 ms  2.054 ms
 2  128.8.1.2  69.29 ms  68.943 ms  68.57 ms
 3  128.9.1.1  104.556 ms  107.078 ms  109.224 ms
 4  128.12.1.6  116.237 ms  172.488 ms  108.982 ms
[root@AS1_H1 /root]#
```

**Fig. 4.** Running *traceroute* inside a vGround

are un-tamperable from inside a VM. *This new design differentiates our technique from other virtual networking techniques* [45, 43] and is critical to the strict confinement feature of vGrounds. Also, the *user-level* implementation of our network virtualization methods brings significant deployability and topology flexibility to vGrounds.

To demonstrate its effect, we again use the PlanetLab example shown in Figure 1. In particular, we run the command *traceroute in the VM AS1\_H1* to find the route to *AS3\_H2*. The result is shown in Figure 4. As we can see, the route is totally orthogonal to the real Internet. More details can be found in [32].

### 3.3 Virtual Node Optimization and Customization

A virtual node in vGround can be one of the following: (1) an end-host exposing certain software vulnerabilities that can be exploited by worms; (2) a router forwarding packets according to routing and topology specification; (3) a firewall monitoring and filtering packets based on firewall rules; or (4) a network/host-based intrusion detection system (IDS) sniffing and analyzing network traffic. We have applied and developed techniques to customize VMs into different types of virtual nodes and to optimize VM space requirement for better scalability.

The system template is a useful facility to share the common part of virtual node images. As shown in Section 2.1, the images of the same type of virtual nodes have a lot in common though they might have different network configuration. Every image file in vGround is composed of two parts: one is a shared system template and the other part is node-specific. In the example in Figure 2, the Apache service started by the script `/etc/rc.d/init.d/httpd start` is common among all end-host images, while the OSPF service started by the script `/etc/rc.d/init.d/ospfd start` is common among all router images. On the other hand, every virtual node has its unique networking configuration (e.g., IP address and routing table), which is specified in the node-specific portion. To execute such specification, we leverage the Copy-On-Write (COW) support in UML. The COW support also helps to achieve high image generation efficiency.

Another optimization is to strip down system templates. When a vGround contains hundreds or thousands of virtual nodes, the templates need to be tailored to remove unneeded services. In worm experiments, this seems feasible because most worms infect and spread via one or only a few vulnerabilities. For example, for the lion worm experiment, a tailored system image of only *7MB* (with BIND-8.2.1 service) can be built. Since the system templates are just regular *ext2/ext3* file systems, it is possible to build customized system templates from scratch. However, available packaging tools such as *rpm* greatly simplify this process.

### 3.4 Worm Experiment Services

To provide users with worm experiment convenience, the vGround platform provides a number of efficient worm experiment services.

**VM Image Generation (by VM).** Every virtual node is created from its corresponding image file containing a regular file system. However, image generation using direct file manipulation operations such as *mount* and *umount* usually requires the *root* privilege of the underlying physical host. To efficiently generate image files *without* the root privilege, an interesting “*VM generating VMs*” approach is developed: the vGround platform first boots a *special* UML-based VM in each physical host, which takes less than 10 seconds. With the support of *hostfs* [30], this special VM is able to access files in the physical host’s file system with regular user privilege. Inside the special VM, image generation will then be performed *using the VM’s own root privilege*. It only takes tens of seconds for the special VM to generate hundreds of system images. We note that the special VM will *not* be part of the vGround being created. Therefore, there is no possibility of worm accessing files in the physical host.

**vGround Bootstrapping and Tear-Down.** The vGround platform also creates scripts for automatic boot-up and tear-down of virtual nodes, to be triggered remotely by the worm researcher. In particular, the sequence of virtual node boot-up/tear-down is carefully arranged. For example, a virtual switch should be ready before the virtual nodes it connects. In the current implementation, each virtual node is associated with a *boot-order/tear-order* number to reflect such a sequence.

**Generation and Collection of Worm Traces.** Each virtual node in vGround has an embedded logging module (included in its VM image). The logger generates worm traces, which will be collected for analyzing different aspects of worms. The vGround platform supports different types of logging modules. In fact, a Linux-based monitoring or intrusion detection system, such as *tcpdump* [9], *snort* [8], and *bro* [1], can be readily packaged into vGround. In addition, we have designed and implemented a *kernelized* version of *snort* called *kernort* [33] that operates in the guest OS kernel of virtual nodes. Kernort generates logs and pushes them down from the VM domain to the physical host domain at runtime.

To collect traces generated by the hundreds and thousands of virtual nodes, manual operation is certainly impractical, especially when the traces need to be collected “live” at runtime. vGround automates the collection process via a toolkit that collects traces generated by different loggers (e.g., *tcpdump*, *kernort*). Furthermore, after an experiment, the worm’s “crime scene” in the vGround can also be inspected and “evidence” be collected, in a way similar to VM image generation: a *special VM* is quickly instantiated to mount the image file to be inspected (an *ext2/ext3* file), and “evidence” collection will be performed via the special VM.

## 4 Worm Experiments in vGrounds

To demonstrate the capability of vGrounds, we present in this section a number of worm experiments we have conducted in vGround using the following real-world worms: the

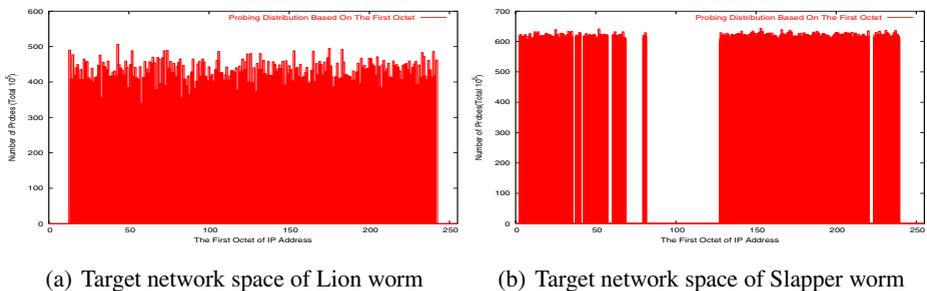
Lion worm [16], Slapper worm [18], and Ramen worm [3]. The experiments span from individual stages for worm infections (e.g., target network space selection (Section 4.1), propagation pattern and strategy (Section 4.2), exploitation steps (Section 4.3), and malicious payloads (Section 4.4)) to more advanced schemes such as intelligent payloads (Section 4.4), multi-vector infections (Section 4.5), and polymorphic appearances (Section 4.5). Throughout this section, we will highlight the new benefits vGrounds bring to a worm researcher, as well as interesting worm analysis results obtained during our experiments. In fact, the worm bulletin misstatement mentioned in Section 1 was identified during these experiments. We discuss the limitations and extensions in Section 5.

The infrastructure in our experiments is a Linux cluster, which belongs to the Computing Center of Purdue University (ITaP). Neither do we have root privilege nor do we obtain special assistance from the cluster administrator, indicating vGround’s good deployability. Each physical node in the cluster has two AMD Athlon processors (each with 64K L1 I-cache, 64K D-cache, and 256KB L2 cache), 1GB memory, and 10GB disk space.

#### 4.1 Target Network Space

Using vGrounds, we first examine the target network space of Lion worms and Slapper worms. We are especially interested in the address blocks that a worm *tries to avoid*. This information not only exposes the worm author’s knowledge about unallocated Internet address blocks [2], but also reveals the address blocks that have been “black-listed” by the black-hat community (for example, the address blocks used for sinkhole networking [51]).

**Lion Worm.** The Lion worm “spreads by scanning random class B IP networks for hosts that are vulnerable to a remote exploit in the BIND name service daemon. Once it has found a candidate for infection, it attacks the remote machine and, if successful, downloads and installs a package...” [4]. To create a vGround for the Lion worm, a system template *lion.ext2* is built, containing the vulnerable version of BIND service. Thanks to vGround’s virtual node optimization techniques, the size of the image is only 7M. A vGround with more than 1500 virtual nodes (1500 virtual end-hosts in ten subnets connected by OSPF routers) is deployed on ten physical hosts each supporting



**Fig. 5.** Target network space of Lion worm and Slapper worm

about 150 virtual nodes. The image files are efficiently generated within 60 seconds and the vGround is boot-up in less than 90 seconds. In this experiment, we deploy “seed” Lion worms in ten virtual end-hosts. Over a one-week period, the vGround automatically collects the traces generated by the *kernort* logging module embedded in the 10 infected end hosts. We then extract and aggregate the IP addresses of attempted targets to show the distribution of Lion worm victims.

Figure 5(a) shows the network distribution of targets probed by the Lion worm, based on the first octet of their IP addresses. The probes are evenly distributed over the range of [13, 243]. It seems that the Lion worm does not skip private or reserved address blocks [2]. To verify this observation, we also perform reverse engineering using *objdump* [6] on the Lion worm binary, which confirms our observation in vGround.

**Slapper Worm.** The Slapper worm exploits a buffer overflow vulnerability in the OpenSSL component of SSL-enabled Apache web servers. If successful, the worm can be used as a back-door to start up a range of Denial-of-Service attacks [5]. The Slapper worm was captured and thoroughly analyzed by researchers at Symantec [38].

A system template *slapper.ext2* contains the vulnerable version of *Apache* server. The size of the image is approximately 32M. A vGround of about 1500 virtual nodes is deployed on 20 physical hosts of the Linux cluster, with each hosting about 75 virtual nodes. Similar to the Lion worm experiment, we extract the probing traffic from the Slapper-infected nodes and then plot the target address distribution in Figure 5(b).

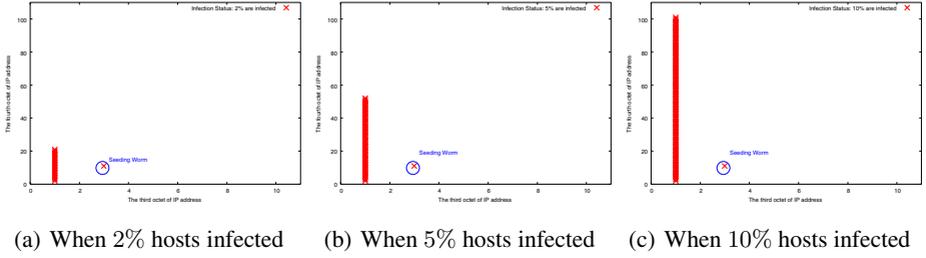
Unlike the Lion worm which ignores the reserved IP address ranges, the Slapper worm deliberately skips certain reserved IP address ranges. The address blocks skipped reflect the global address assignment *at the time when the Slapper worm was released*. For example, back then, the address blocks of 082/8 - 088/8 are reserved by IANA (Internet Assigned Numbers Authority) and therefore skipped by the Slapper worm, as shown in Figure 5(b). As of today, however, these address blocks are no longer reserved by IANA [2].

## 4.2 Propagation Pattern

Understanding a worm’s propagation pattern is important to the design of worm containment mechanisms. In this experiment, we demonstrate that vGrounds achieve sufficiently large scale to observe a worm’s propagation pattern.

We create a vGround with 1000 vulnerable end-hosts running in 10 networks each with 100 end hosts ( $192.168.x.y$ ,  $x = 1 \dots 10$ ,  $y = 1 \dots 100$ ). At the beginning, there is *one* Slapper-infected “seeding” node (192.168.3.11) in the vGround. We allow the Slapper worm to propagate in the vGround and the propagation progress is recorded. Based on the vGround traces, the propagation pattern of Slapper worm can be visualized in Figure 6. The three sub-figures show the status of the vGround at three different time instances: when 2%, 5%, and 10% of the end-hosts in the vGround are infected, respectively. The x-axis is the third octet of an end-host’s IP, while the y-axis is the fourth octet. An “X” indicates that the corresponding end-host is infected. The figure shows the progress and victim distribution of Slapper worm propagation.

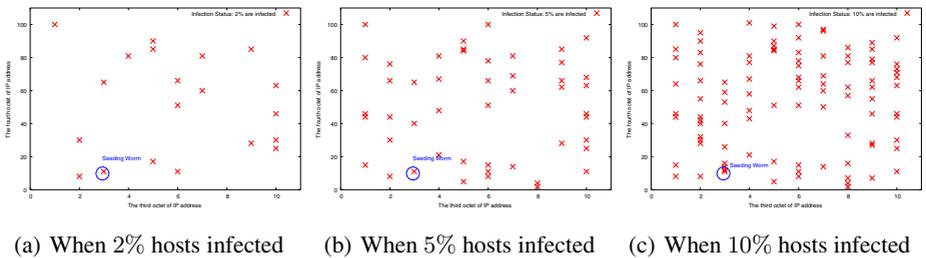
From Figure 6, it can be conjectured that the Slapper worm is using the *address-sweeping* strategy when selecting victims: once an address range such as 192.168.0.0/16



**Fig. 6.** Propagation of Slapper worm w/ *address-sweeping* (total: 1000 hosts)

is chosen, hosts within the address range will then be *sequentially scanned*. Figure 6 shows that all the infected nodes are so far in the same subnet. A closer look at the detailed vGround traces reveals the reason: it takes some time for the seed worm to “hit” the 192.168.0.0/16 range and start infecting the hosts. The newly spawned worms will do the same as the seed worm. If one of them hits the same range, it will “sweep” the IP addresses again *in the same sequence* (i.e. from 192.168.0.1 to 192.168.254.254). An analysis of the Slapper worm source code confirms our conjecture.

We note that the scale of the above vGround may *not* be large enough to observe other propagation patterns. For example, we synthesize a *Slapper worm variant* using the *island-hopping* strategy [36]. Under this strategy, the seed worm targets the hosts in *its own* /16 range with high probability (0.75), and hosts outside the range with low probability (0.25). The same vGround for the original Slapper is used to run the Slapper variant. The propagation pattern is visualized in Figure 7. It is clear that the hosts in the worm’s local range (192.168.0.0/16) are infected *randomly* instead of *sequentially* as in address sweeping. Our vGround traces also indicate that the seed worm as well as the newly spawned worms will *immediately* start to infect local hosts, without the delay (caused by random range selection) observed in address sweeping. Unfortunately, the “hopping away” behavior (i.e. worms infecting hosts outside the local range) cannot be observed in the vGround, due to the limited address space of the vGround. As our solution, we develop a new technique called *worm-driven vGround growth*: when a worm’s probing target is generated and the target is not in the vGround, a new subnet with at least the target host will be dynamically generated and added to the vGround within *seconds*. Other techniques such as NAT/reverse-NAT,



**Fig. 7.** Propagation of Slapper worm variant w/ *island-hopping* (Total: 1000 hosts)



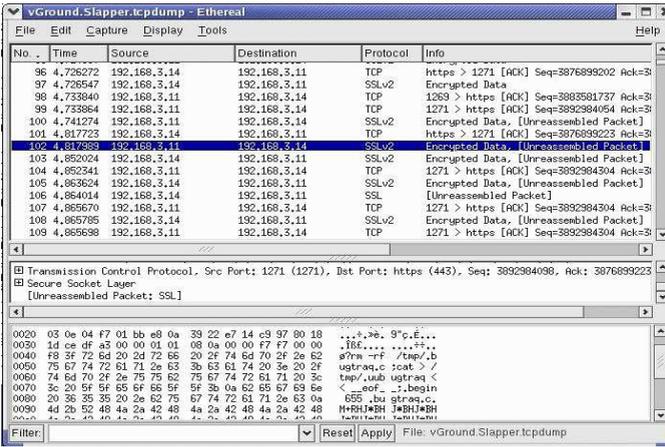


Fig. 9. Exploitation details of the Slapper worm

Apache’s pool of server and thus forces the creation of two fresh processes when serving the next two SSL connections. The purpose of “forking” two fresh processes is to have the same heap structures within them and thus prepare for the final two SSL handshake exploitations. The first SSL connection exploits the vulnerability to obtain the exact location of affected heap allocation, and it is used in the second SSL connection to correctly patch attack buffer. The second SSL connection re-triggers the heap-based buffer overflow which transfers the control of the just-patched attack buffer.

Due to space constraint, we do not show the full vGround traces during the above exploitation process. Instead, the trace in the final stage of the attack is shown in Figure 9. From the decoded area of Figure 9, it is interesting to see that the worm source is transferred in the *uencoded*<sup>2</sup> format.

#### 4.4 Malicious Payload

A worm’s payload reveals the intention of the worm author and often leads to destructive impact. The vGround is an ideal venue to invoke the malicious payload, because the consequent damage will be confined within the vGround. Moreover, the vGround will be easily recoverable due to the all-software user-level implementation.

The following string is found in the Lion worm trace in Figure 8: `find / -name "index.html" -exec /bin/cp index.html {} \;`. The Lion worm recursively searches for all *index.html* files starting from the “/” root directory and replaces them with a built-in web page. This malicious payload is confirmed by our forensic analysis enabled by the vGround post-infection trace collection service (Section 3.4). We also run an *earlier version* of the Lion worm in a separate vGround. We observe that the Lion worm carries and installs an infamous rootkit - *tor*n [28], which will destroy the infected

<sup>2</sup> Uencode, or the full name “*Unix to Unix Encoding*”, represents a method or tool for converting files from binary to ASCII(text) so that they can be sent across the Internet via email.

```

[root@cl_2 /root]#pudclient 127.0.0.1
PUD Client version 11092002Ready, type in the
commands as follows, or type help for a list:

help
The commands are:
* kill      kills the daemon

* log       log output to file

* bounce    adds a bounce
* close     closes a bounce

* info      requests info
* list      lists the current servers
* sh        execs a command

* udpflood  send a udp flood
* tcpflood  send a tcp flood
* dnspflood send a dns flood

* escan     scans hard drive for emails

```

**Fig. 10.** Payloads of the Slapper worm

host. Without full-system virtualization, such *kernel-level damage* cannot be easily reproduced. Furthermore, the vGround contains the damage and makes the system re-installation fast and easy.

The Slapper worm does not destroy local disk content like the Lion worm. It is more *advanced* in self-organizing worm-infected hosts into a *P2P attack network*. In the vGround for the Slapper worm, we are able to observe the operations of this P2P network. More specifically, we deploy a special client [19] in one of the end hosts. The special client will issue commands (listed in Figure 10) to the infected hosts. Meanwhile, each Slapper worm carries a DDoS payload component [19]. In the vGround, we are able to issue commands such as *list*, *udpflood*, and *tcpflood* via the special client. The vGround traces indicate that a command is propagated among the infected hosts in a P2P fashion, rather than being sent directly from the special client. The vGround provides a convenient environment to further investigate such advanced attack strategy.

#### 4.5 Advanced Worm Experiments

In this section, we present a number of more advanced experiments where vGrounds demonstrate unique advantages over other worm experiment environments.

**Multi-vector Worms.** Multi-vector worms are able to infect via *multiple infection vectors (IVs)*. In this experiment, we run the Ramen worm [3, 17], which carries three different IVs in three different services, including LPRng (CVE-2000-0917), wu-ftpd (CVE-2000-0573), and rpc.statd (CVE-2000-0666). A vGround with 1000 virtual nodes running these services is created and only one seed Ramen worm is planted. Over the time, however, we notice *different infection attempts based on all three IVs*.

Interestingly, our vGround experiments reveal that *the Ramen exploitation code for the vulnerable wu-ftpd server is flawed* - a result *not mentioned* in popular bulletins [3] and [17]. To confirm, we also use the same exploitation code against a real machine running a vulnerable FTP server (wu-ftpd-2.6.0-3). The result agrees with the vGround result.

**Stealthy/Polymorphic Worms.** Using various polymorphic engines [34], worms can become extremely stealthy. The modeling and detection of stealthy behavior or

polymorphic appearances require much longer time and larger playground scale. Furthermore, it is hard, if not impossible, for worm simulators [39] to experiment polymorphic worms.

We have synthesized a polymorphic worm based on the original Slapper worm. We use it to evaluate the effectiveness of signature-based worm detection schemes. As shown in Section 4.3, the Slapper worm will transfer an *uuencoded* version of the worm source code after a successful exploitation. Our polymorphic Slapper first attempts to encrypt the source using the *OpenSSL* tool before transmission. The encryption password is *randomly generated* and is then XOR'ed with a shared key. Finally, the resultant value is prepended to the encrypted worm source file for transmission. Our vGround experiments show that snort [8] is no longer able to detect the worm<sup>3</sup>. The same worm could also be used to test the signatures generated by various signature extraction algorithms [42, 35, 37].

**Routing Worms.** The vGround can also be used to study the relation between worm propagations and the underlying routing infrastructure. We have recently synthesized the routing worm introduced in [52]. The routing worm takes advantage of the information in BGP routing tables to reduce its scanning space, without missing any potential target. With its network virtualization and real-world routing protocol support, the vGround provides a new venue to study (at least qualitatively) such an infrastructure-aware worm and the corresponding defense mechanisms.

## 5 Limitations and Extensions

It has been noted [11] that a UML-based VM exposes certain system-wide footprints. For example, the content in */proc/cmdline* can reveal the command parameters when a UML VM is started and the command parameters contain some UML-specific information (e.g., the special root device *ubd0*). Such deficiency may undesirably disclose the existence of vGround. As a counter-measure, methods have been proposed [27] to minimize such VM-specific footprints. However, this is not the end of the problem. Instead, it may lead to another round of “arms race”. From another perspective, an interesting trend is that VMs, including UML VMs, are increasingly used for *general computing purposes* such as web hosting, education, and Grid computing [30, 43]. If such trend prevails, the arms race tension may be *mitigated* because a worm might as well infect a VM in such a “mixed-reality” cyberspace.

In addition, the confined nature of vGround may turn out *disabling* some worm experiments where the worm has to communicate with hosts outside the vGround to “succeed”. For example, the Santy worm [22] relies on the *Google* search engine to locate targets for infection and it can be effectively mitigated by filtering the worm-related queries [20]. However, the vGround cannot be readily used to safely observe the dynamics of such worms<sup>4</sup>. Although the vGround platform does have the capability to intercept an external connection attempt and forge a corresponding re-

<sup>3</sup> The Slapper signature used in snort is the string “TERM=xterm”.

<sup>4</sup> In fact, due to the strict confinement requirement, even a dedicated worm testbed is *not* able to support such study.

sponse, it remains an open question whether such technique can survive the subsequent counter-measures taken by the worms.

Another limitation is that current prototype is only applicable to Linux worms, even though the design principles and concepts can be generally applied to build vGrounds for Windows worms. One notable hard challenge in extending current vGround implementation for Windows worms is to develop highly scalable system virtualization and customization techniques for Windows systems. However, it is encouraging to note that recent advances in system virtualization technologies such as the VMware ESX server [13] and hardware virtualization support such as the Intel’s Vanderpool technology [24] have shown great promises in addressing this challenge. Once these technologies become available, they can be naturally leveraged to support Windows-based vGrounds.

## 6 Related Work

**Testbeds for Destructive Experiments.** The DETER project [10] provides a shared testbed to researchers to conduct a wide variety of security experiments. With a pool of physical machines in a number of sites, the DETER testbed is able to provide each researcher with a virtually dedicated experiment environment in an efficient on-demand fashion. In the current practice, the granularity of resource allocation is often one physical node. The vGround software platform can be deployed in the DETER testbed as a *value-added worm experiment service*. As a result, worm researchers will benefit not only from the testbed’s general services (e.g., topology generation, result visualization), but also from the new features brought by vGround (i.e. easy recovery, larger scale, and confinement).

Netbed [50], Modelnet [47], and PlanetLab [7] are highly valuable and accessible testbeds/environments for general networking and distributed system experiments. On the other hand, the vGround platform is an enabling software system that can potentially (“already” in the case of PlanetLab) be deployed in these testbeds to enhance their support for *destruction-oriented* worm experiments. For example, PlanetLab and Modelnet currently do not support worm experiments, especially when kernel-level damages (e.g., kernel-level rootkit installation) are incurred.

The anti-virus industry has long been building worm testbeds (including virtualization-based testbeds) for timely capture and analysis of worms. Such testbeds are mainly for *in-house* exclusive use by highly skillful and specially trained experts. As a result, *wide deployability*, *infrastructure sharing*, and *user convenience* are *not* their primary design concerns. One of the pioneering industry testbeds is Internet-inna-Box [48] originally built at IBM. It involves virtual machines and virtual networks, both enabled by an “emulation package” that supports virtual *Win9x* environments. The testbed is based on one or more physical machines, each with *two physical* network connections - one dedicated to traffic between the VMs. While sharing the same principle of system and network virtualization, vGrounds *do not* require dedicated network connections and administrator privileges. Also, the vGround platform imposes lower requirement of user skills by performing automatic vGround generation and deployment. Further, vGrounds support virtual routers and user-specified network topology. However, vGround currently does not support Windows worms.

**VM-Based Worm Investigation.** Virtual machines provide an isolated virtualization layer for running and observing untrusted services and applications. Among the notable VM technologies are VMware [13], User-Mode Linux (UML) [30], Denali[49], and Xen[26]. VM technologies have been heavily leveraged to study worms. In current practice, various VM technologies including VMware [13] and User-Mode Linux (UML) [30] have been actively deployed as honeypots to *capture worms*, especially during the early stage of their propagation. To *analyze a worm*, VM-based technologies have also been developed. One advanced VM-based forensic platform is ReVirt[31]. ReVirt enhances individual VMs with efficient logging and replay capabilities for intrusion analysis purpose, making it possible for a worm researcher to replay the worm exploitation process in an instruction-by-instruction fashion. Finally, to study *how worms propagate*, we have argued that only VMs are not enough, leading to our development of new network virtualization techniques.

**Virtual Networks.** Recently, network virtualization attracts increasing research attention. In [25], research efforts are called for to create “virtual testbeds” on top of shared distributed infrastructures - the vGround platform is *a step towards this vision*. Different virtual networks have been developed such as X-bone [45], VNET [43], and VIOLIN [32]. Both X-bone and VNET create a “virtual Internet” which does not hide the existence of the underlying physical hosts and their network connections. If used in vGround, they would not be able to confine worm traffic within the virtual Internet. VIOLIN is our previous effort in network virtualization and it *does not* provide automatic virtual network generation and bootstrapping.

**Honeypot Systems.** We first note that *a vGround itself is not a honeypot system*. Recently, there have been significant advances in honeypot systems and their applications [40, 29, 51]. For example, Honeyd [40] is a highly scalable and efficient framework for *low-interaction* virtual honeypots. The vGround platform and honeypot systems are different in nature: Honeypot systems are *connected to and interact with* the real Internet, while the vGround is an *isolated virtual environment to replay worm behavior*. As a result, they perfectly complement each other. In fact, a promising integration will be to use honeypot systems to “capture” real-world worms, and then use vGrounds to run the captured worms in a realistic but isolated environment. Such an integration has great potential in *automatic* capture and characterization of 0-day worms.

## 7 Conclusion

The vGround platform enables impact-confined and resource-efficient experiments with Internet worms. The main features of vGround are supported by a suite of virtualization-based new techniques. Using real-world worms, we have demonstrated that vGrounds are high-fidelity confined playgrounds to run worms and observe key aspects of their behavior, including network space targeting, propagation pattern, exploitation steps, and malicious payload. These results are critical to the development of worm detection and defense mechanisms, which can also be tested in vGrounds. For worm researchers, the vGround platform accommodates their *iterative* experiment workflows with great

efficiency and convenience. The vGround platform makes a timely contribution to worm detection and defense research.

## Acknowledgments

We thank Aaron Walters, David Evans, Sonia Fahmy, Wenke Lee, Ninghui Li, Peng Ning, and Yi-Min Wang for providing insightful comments on early versions of this paper. The final version of this paper benefits from valuable suggestions from the anonymous reviewers and the guidance of our shepherd, George Mohay. This work was supported in part by NSF Grants SCI-0504261 and SCI-0438246, and a gift from Microsoft Research. Some of this effort was also supported by the sponsors of CERIAS, and that support is gratefully acknowledged.

## References

- [1] Bro. <http://bro-ids.org>.
- [2] Internet Protocol V4 Address Space. <http://www.iana.org/assignments/ipv4-address-space>.
- [3] Linux Ramen Worm. <http://service1.symantec.com/sarc/sarc.nsf/html/pf/linux.ramen.worm.html>.
- [4] Linux/Lion Worms. <http://www.sophos.com/virusinfo/analyses/linuxlion.html>.
- [5] Linux/Slapper Worms. <http://www.sophos.com/virusinfo/analyses/linuxslapper.html>.
- [6] objdump. [http://www.gnu.org/software/binutils/manual/html\\_chapter/binutils\\_4.html](http://www.gnu.org/software/binutils/manual/html_chapter/binutils_4.html).
- [7] PlanetLab. <http://www.planet-lab.org>.
- [8] Snort. <http://www.snort.org>.
- [9] Tcpdump. <http://www.tcpdump.org>.
- [10] The DETER Project. <http://www.isi.edu/deter/>.
- [11] The HoneyNet Project. <http://www.honeynet.org>.
- [12] Virtual PC. <http://www.microsoft.com/windows/virtualpc/default.msp>.
- [13] VMware. <http://www.vmware.com/>.
- [14] ISC Bind 8 Transaction Signatures Buffer Overflow Vulnerability. <http://www.securityfocus.com/bid/2302>, 2001.
- [15] Linux Adore Worms. <http://securityresponse.symantec.com/avcenter/venc/data/linux.adore.worm.html>, 2001.
- [16] Linux Lion Worms. <http://www.whitehats.com/library/worms/lion/>, 2001.
- [17] Ramen Worm. <http://www.sans.org/y2k/ramen.htm>, Feb. 2001.
- [18] CERT Advisory CA-2002-27 Apache/mod\_ssl Worm. <http://www.cert.org/advisories/CA-2002-27.html>, 2002.
- [19] PUD: Peer-To-Peer UDP Distributed Denial of Service. <http://www.packetstormsecurity.org/distributed/pud.tgz>, 2002.
- [20] Google Smacks Down Santy Worm. <http://www.pcworld.com/news/article/0,aid,119029,00.asp>, Dec. 2004.
- [21] MyDoom Worms. <http://us.mcafee.com/virusInfo/default.asp?id=mydoom>, 2004.
- [22] Santy Worms. [http://www.f-secure.com/v-descs/santy\\_a.shtml](http://www.f-secure.com/v-descs/santy_a.shtml), Dec. 2004.
- [23] Witty Worms. <http://securityresponse.symantec.com/avcenter/venc/data/w32.witty.worm.html>, Mar. 2004.
- [24] Vanderpool Technology. <http://www.intel.com/technology/computing/vptech/>, 2005.
- [25] T. Anderson, L. Peterson, S. Shenker, and J. Turner. A Global Communications Infrastructure: A Way Forward. [http://www.arl.wustl.edu/netv/contrib/nsf\\_Dec2.ppt](http://www.arl.wustl.edu/netv/contrib/nsf_Dec2.ppt), Dec. 2004.

- [26] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, R. N. Alex Ho, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. *SOSP 2003*.
- [27] C. Carella, J. Dike, N. Fox, and M. Ryan. UML Extensions for Honeypots in the ISTS Distributed Honeypot Project. *Proceedings of the 2004 IEEE Workshop on Information Assurance United States Military Academy, West Point, NY, June 2004*.
- [28] P. Craveiro. SANS Malware FAQ: What is t0rn rootkit? [http://www.sans.org/resources/malwarefaq/t0rn\\_rootkit.php](http://www.sans.org/resources/malwarefaq/t0rn_rootkit.php).
- [29] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. HoneyStat: Local Worm Detection Using Honeypots. *Proceedings of the 7th RAID*, Sept. 2004.
- [30] J. Dike. User Mode Linux. <http://user-mode-linux.sourceforge.net>.
- [31] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *OSDI 2002*.
- [32] X. Jiang and D. Xu. VIOLIN: Virtual Internetworking on Overlay Infrastructure. *Technical Report CSD-TR-03-027, Purdue University*, July 2003.
- [33] X. Jiang, D. Xu, and R. Eigenmann. Protection Mechanisms for Application Service Hosting Platforms. *CCGrid 2004*, Apr. 2004.
- [34] K2. ADMmutate. *CanSecWest/Core01 Conference, Vancouver* <http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz>, Mar. 2001.
- [35] H. A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. *Proceedings of the 13th Usenix Security Symposium*, Aug. 2004.
- [36] J. Nazario. *Defense and Detection Strategies against Internet Worms*. Artech House Publishers, ISBN: 1-58053-537-2, 2004.
- [37] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. *Proceedings of Oakland 2005*, May 2005.
- [38] F. Perriot and P. Szor. An Analysis of the Slapper Worm Exploit. *Symantec White Paper* <http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf>.
- [39] K. S. Perumalla and S. Sundaragopalan. High-Fidelity Modeling of Computer Network Worms. *Proceedings of 20th ACSAC*, Dec. 2004.
- [40] N. Provos. A Virtual Honeypot Framework. *Proceedings of the USENIX 13th Security Symposium, San Diego, USA*, Aug. 2004.
- [41] T. Ptacek and J. Nazario. Exploit Virulence: Deriving Worm Trends From Vulnerability Data. *CanSecWest/Core04 Conference, Vancouver*, Apr. 2004.
- [42] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. *Proceedings of the ACM/USENIX OSDI*, Dec. 2004.
- [43] A. Sundararaj and P. Dinda. Towards Virtual Networks for Virtual Machine Grid Computing. *Proceedings of the Third USENIX Virtual Machine Technology Symposium (VM 2004)*, Aug. 2004.
- [44] P. Szor. Fighting Computer Virus Attacks. *Invited Talk, the 13th Usenix Security Symposium (Security 2004), San Diego, CA*, Aug. 2004.
- [45] J. Touch. Dynamic Internet Overlay Deployment and Management Using the X-Bone. *Proc. of IEEE ICNP 2000*, Nov. 2000.
- [46] J. Twycross and M. M. Williamson. Implementing and Testing a Virus Throttle. *Proceedings of the USENIX 12th Security Symposium, Washington, DC*, Aug. 2003.
- [47] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *OSDI 2002*.
- [48] I. Whalley, B. Arnold, D. Chess, J. Morar, and A. Segal. An Environment for Controlled Worm Replication & Analysis (Internet-inna-Box). *Proceedings of Virus Bulletin Conference*, Sept. 2000.
- [49] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. *Proceedings of USENIX OSDI 2002*, Dec. 2002.

- [50] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *Proceedings of 5th OSDI*, Dec. 2002.
- [51] V. Yegneswaran, P. Barford, and D. Plonka. On the Design and Use of Internet Sinks for Network Abuse Monitoring. *Proc. of 7th RAID*, Sept. 2004.
- [52] C. C. Zou, D. Towsley, W. Gong, and S. Cai. Routing Worm: A Fast, Selective Attack Worm based on IP Address Information. *Umass ECE Technical Report TR-03-CSE-06*, Nov. 2003.