SAND2011-3057C

Combining HPC and Virtual Machines to understand Internet-scale phenomena

Ron Minnich
Don Rudish
John Floren
Sandia National Laboratories

Andrew Sweeney
David Fritz
Keith Vanderveen
Sandia National Laboratories

Kevin Pedretti
Kristopher Watts
Casey Deccio
Sandia National Laboratories

***Abstract*—In this paper we describe the application of supercomputing to a new area: the setup and control of hundreds of thousands to millions of virtual machines, in order to facilitate research into the behavior of internet-scale phenomena such as peer-to-peer networking, botnets, routing behavior, and new network protocols. We are currently able to run 1200 Linux virtual machines, or 200 Windows 7 virtual machines, on a single HPC node. In each case, we can boot the full complement of virtual machines in a few minutes. We are overcommitting the resources of the system to an unusual degree: the CPU by factors of several hundred, and the memory by a factor of at least 10.**

An early emphasis of our work is to run botnets in captivity and at scale, because they are a low-hanging fruit and at the same time, they are a good measure of how well the system is working. Botnets are complex distributed systems consisting of many tens of thousands of individual instances of malware which, once connected, are resilient, self-healing, controllable from a central place, and capable of autonomous behavior; botnets exhibit a wide variety of complex behavior and network structures [5]. Furthermore, many important characteristics and effects of botnets, such as DDOS attack potency and controllability, emerge only at the scale of tens to hundreds of thousands of nodes [5]. There are very few organizations with supercomputers with ten thousand sockets, much less one hundred thousand. Virtualization is required to run a botnet at scale.

For the past four years, we have been working to run large numbers of tightly connected virtual machines on HPC systems. We began with the ability to boot 100 virtual nodes on our laptops, to test clustering software; from that point, we have scaled up and, in the process, have developed new ways of managing HPC systems at this scale.

Using our software, we recently ran an IRC botnet that we captured in the wild. We ran this botnet on a 520-node cluster, hosting it on 62,000 Windows 7 virtual machines.

In this paper, we describe the cluster we have built, the software we have developed, and its recent applications. Our experience has shown that few HPC software scales to the millions. Problems discovered through our emulated environment will mirror scaling problems in the HPC world. Hence, the software we develop may well be useful for future large-scale HPC systems.

I. Introduction

As of this writing, there are no longer any 32-bit IP addresses left, and many of those IP addresses front organizations, not computers: the real "address space" of the Internet is hard to estimate, but it is certainly larger than 32 bits.

Finding out what is going on even a small piece of the Internet is a daunting task. Much of the observed behavior of the Internet is difficult to understand, in part because collecting the data is so difficult. Typically, those hoping to measure some phenomena attach probes to points on the Internet and try to extrapolate behavior from those points. Frequently, organizational, national, and technical boundaries sharply limit how many probes can be set up. It is somewhat like determining the health of a whale by examining hair follicles, subject to the rule that if one examines the flukes, one is not also allowed to examine the flippers.

The intrinsic communication behaviors of *botnets* make them both interesting and difficult to observe. Botnets are complex distributed systems consisting of many tens of thousands of individual instance of malware (called "bots") which, once connected, are resilient, self-healing, controllable from a central place, and sometimes capable of sophisticated autonomous behavior.

Botnets are an example of well-designed distributed systems software. Sophisticated botnets such as Storm, Conficker, and Waledac use or have used peer-to-peer (P2P) networks for command and control [8] and [6], which results in an "overlay" botnet topology unrelated to the underlying Internet and geography. The result is that two "nearby" nodes may be far apart in space, and hosted in two completely different subnets. This address structure makes advanced botnets resilient to outages of either countries or single organizations.

Traditionally, studies of botnet software and behavior have relied nearly exclusively on reverse engineering of captured bot binaries, dynamic analysis of bot binaries using sandboxes, and observation of botnets "in the wild" using honeynets/honeyfarms [18] or through insertion of an instrumented false bot controlled by researchers into an extant botnet [12] and [9]. As noted by studies such as Calvet [3] and Barford [1], however, the aforementioned techniques cannot provide the "big picture" of a botnet's operations.

The solution, as recognized by Calvet [3] and Barford [1], is to build a network testbed capable of holding an entire operational botnet in a "network sandbox." Like sandboxing of individual bots, a capability to sandbox an entire functioning botnet would provide the opportunity to investigate the botnet's behavior and function, test "what if" scenarios, and reliably re-run experiments to generate confidence in the researcher's conclusions, all without threatening the safety and reliability of the Internet.

Both Calvet [3] and Barford [1] demonstrated testbeds capable of holding a few hundred to a few thousand nodes of a botnet, and both correctly highlight the importance of scale in understanding botnet behavior. However, we believe that even greater scale is needed, because actual botnets can consist of hundreds of thousands to millions of nodes [19]. Trying to understand botnets and their interactions with (and effects on) networks by extrapolating from studies done at two to three orders of magnitude smaller scale risks missing crucial effects that manifest themselves in the real botnet running on the real Internet.

Further, we can anticipate building useful software systems based on botnet concepts. Many botnet properties are desirable for HPC systems and application software. We intend to investigate the benefits of using the structures of million-scale botnets for monitoring, diagnosing, and controlling HPC systems.

In the remainder of this paper, we discuss the creation of a network testbed capable of running the largest botnets discovered to date on the Internet, and initial experiments conducted on this testbed with a real botnet. We have developed a prototype testbed consisting of a 520-node cluster, capable of hosting 62000 Windows 7 virtual machines or 600000 Linux virtual machines, each of which can in turn host application software and malware. We attacked the problem of achieving a larger scale botnet testbed through several different approaches: use of high performance computing hardware, development of scalable cluster management software, and development of custom-built lightweight OS kernels.

Sections references are incorrect.

The rest of this paper is organized as follows. In section 2, we discuss related work in the context of harnessing large numbers of virtual machines to study botnets. In section 3, we discuss our research into building and running lightweight Linux virtual machines, and the management software we have developed which allows us to control the process of booting and configuring these machines by the millions. In section 4, we describe the special-purpose cluster which we built to carry out this research, and discuss its similarities and differences with more conventional computing clusters. In section 5, we describe the application of the lightweight Linux VM research to the task of making a lightweight Windows VM, and we also discuss reverse engineering of botnets, which is crucial to getting bot instances to run at scale in virtual machines. In section 6, we discuss how we monitor an experiment on our testbed, and the challenges in collecting and analyzing data at the scale and degree of oversubscription of resources exhibited by our system. In section 7, we present our findings, and finally we conclude and present some limitations of our work as well as possible areas for future research in section 8.

II. Related Work

Previous research into understanding the behavior of botnets has followed the approaches of reverse engineering and static analysis of captured bot code [16], breaking into an extant botnet[12], running actual bot code at a small scale[15], or creating a simulation of the botnet's network behavior and observing that [6]. While all of these approaches have increased our understanding of botnets, what has been missing is an experimental platform capable of running a full-sized botnet in a controlled environment.

A. Scalable Platforms to Host Bots

Some prior research has sought to achieve scalable tools for better understanding botnets and other malware. The Potemkin project [18] used virtualization, oversubscription of physical resources, and late binding of resources to requests to achieve a high fidelity honeyfarm capable of scaling to tens of thousands of emulated hosts. Unlike our project, the goal of Potemkin was not to actually run a botnet in its entirety. Potemkin aimed to present a large number of vulnerable systems to elicit attempts from the Internet to compromise the systems, and thereby learn about the exploits used by malware, understand the behavior of malware after it has compromised a new host, and capture samples of malware. While some of the techniques used by Potemkin are similar to our project (lots of VMs, oversubscription of resources), Potemkin mostly focused on and facilitated interactions between its VMs and the rest of a botnet residing on the Internet, as opposed to our project, in which bots on different VMs interact with each other in a closed environment.

Barford et al. [1] demonstrated a botnet testbed with similar goals to our project and designed to scale to thousands of bots. The system, called the Botnet Evaluation Environment, was built to run on Emulab [20] enabled network testbeds such as DETER [2], and contained essential services such as DNS and IRC to provide a closed environment within which a fully formed botnet could function, albeit at the scale of hundreds to thousands of bots.

More recently, Calvet et al. hosted a captive Waledac botnet with 3000 bot instances [3]. They achieved this using a 98-node server farm and roughly, 30 VMs per physical node, with each VM presumably containing an instance of Windows infected with the Waledac bot. Calvet et al. give convincing reasons why emulation of a botnet at scale is a necessary adjunct to understanding and observations of botnets in the wild.

Our project has similar goals to the Barford and Calvet emulation testbeds. However, we were able to improve on the scale of experiments reported by both groups by more than an order of magnitude. The improvement was due to our use of lighter weight virtual machines, substantial efforts to decrease the memory footprints of Linux and Windows instances, use of scalable cluster management software, and use of larger clusters commonplace in the high performance computing community.

III. Emulation Environment

1) Overview: The emulation environment aims to achieve two goals, scale and fidelity. Scale is essential for understanding characteristics of botnets that only emerge at realistic sizes [3]. Host level fidelity, on the other hand, is necessary in order

to achieve credible simulation results, and simplifies the task of getting malware to run in the emulation environment with minimal modifications.

Oversubscribing resources on a massive scale create various problems. First, we are required to use network hardware not designed for such over-subscription. Limitations on cache sizes may result in a breakdown in normal network operation. On an Ethernet switch, too many MAC addresses will result in a CAM table overflow. Second, we are constrained on providing a high fidelity emulation of the target environment. As such, our system virtualizes hardware but uses the genuine software stack from the operating system, such as Linux, Windows, or Cisco IOS, as well as the applications running on it. We employ different methods for putting together the different instances in the emulation environment depending on desired results. For example, we employ Linux as the unaltered host OS on each of the physical nodes to maintain a genuine protocol stack, but then we may execute windows malware inside of WINE for the compatibility layer. The MegaWin effort (discussed below) runs VMs containing a true Windows 7 operating system to achieve higher host-level fidelity, but at some cost in scale. The emulation environment also aims at compensating for properties that are intrinsic to a virtualized environment. An example of a compensating factor would be the insertion of artificial latency through traffic controls.

2) vmatic: We created the vmatic software package to address the problem of bootstrapping an Internet-like emulation. It consists of a set of modifications to the Sandia oneSIS software[1] and some new tools for monitoring and process startup.

We designed vmatic to run on diskless nodes, for several reasons, not all of them technical:

- The high end systems – Blue Gene and Cray XT – we are targeting for the largest:wq runs consist completely of diskless nodes. It is essential that we design our software in a way that will run well on these systems as they allow us to create environments ranging to the multi-million node scale.
- It has not proven practical to share a single disk between thousands of VMs per node;
- if we ever connect the emulated envirnment to the real internet, we want to be able to erase any downloaded data with a simple power cycle.

The standard oneSIS diskless mode requires an NFS root file system. This does not scale very far; in fact oneSIS requires a hierarchy of NFS servers to support clusters with more than 256 nodes, and even in that case many write-required files and directories, such as `/tmp`, need to be mounted on a local ram disk, not on the NFS server.

We extended the oneSIS diskless mode to support a pure local RAM root file system. On clusters, the kernel and initial RAM disk (initrd) are downloaded via PXE. The initial RAM disk includes a kernel and initrd for the guest VMs. In each case, the initrd includes enough programs to boot the node. Any other programs that need to be run are pushed to the node via gproc (described below). The result is a pure memory-based node that has no dependencies on external file system mounts. This design has been tested on a broad scale of systems, from booting 100 VMs on our laptops to booting millions of VMs on the Cray XT system at Oak Ridge (Jaguar).

vmatic is responsible for the configuration of network and startup services, with the end goal of creating a national-scale, unified standalone virtual Internet.

A. Computational Configuration

A common design for cluster management systems is to use a configuration file to define address to IP mappings and other host parameters for each host. Many of these systems have an excruciatingly detailed per-host configuration, with in some cases hundreds of bytes of XML *per host*. On some cluster systems, a million hosts would require a configuration file on the order of 500 Megabytes. This configuration approach is clearly impractical as we move to larger scale systems.

Put another way, cluster systems follow a "configure, then boot" model. For the millions-of-nodes scale we are targeting, we have adopted a "boot, then configure" model. As the nodes come up, they examine their state, and the host-local configuration files are written dynamically. In some cases, such as for gproc, there are no configuration files at all; the software determines its place in the hierarchy by reading hardware state (such as the Torus coordinates on the Cray) and then computing parameters.

We call this technique computational configuration: the key control parameters are computed, rather than written in a file, and are recomputed each time the system is booted. The computation itself is embarrassingly parallel. Many of the parameters are deterministic, i.e. a given node will have the same configuration from boot to boot. Repeatable configuration is important on higher end systems to enable optimized use of the network. On Blue Gene, for example, the 3D coordinate of a node in the Torus network is directly linked to that node's position in the Collective Routing Network, and hence software trees should replicate the hardware structure as much as possible.

The use of the computation tactic for deployment as opposed to a centralized approach like DHCP has allowed our HPC platforms to emulate much larger systems containing networks and routers ranging in the thousands and other fully routable nodes ranging in the millions. The total time to instantiate such a network is a matter of minutes. The vmatic deployment on 7,816 Jaguar Cray Compute Nodes achieved its boot of 4.5 million fully routable virtual nodes in 18 minutes.

In order to scale efficiently and ease the deployment on various HPC platforms, vmatic splits the build process into a static and a dynamic segment. The static segment is primarily responsible for defining the physical resources and fixed addresses of the HPC platform which will serve as a basis for constructing the backbone of the virtual Internet. The dynamic segment uses the backbone definition, a unique key

[1]http://onesis.org

identifier and a consistent hash shared amongst all nodes to compute non-conflicting network information. In this scheme, a physical node encompasses all the necessary information for assembling a piece of the virtual Internet independently and in a non-conflicting manner.

Advanced features in the Linux kernel were applied to achieve the excessive oversubscription of physical resources. The first was the tickless kernel feature applied to both the host and guest operating systems. This was done to avert interrupt timers that would periodically query the system for outstanding tasks to processes. With over one thousand virtual machines on a single physical host, each VM would be constantly active and would be in a constant state of contention for the physical CPU. With tickless kernel enabled in combination with a convention to limit unnecessary applications that aggressively contend for CPU cycles, a physical node can deploy over a thousand virtual machines but still achieve close to 100% idle time. This is possible since the VM process can remain in a sleep state longer when idle, and will only wake up when a task is scheduled for execution. This tactic is applicable to various malware applications that do not demand a constant state of processing.

Conservation of memory resources is critical, making tactics such as copy on write and other techniques such as page-level merging schemes essential. The KSM page level de-duplication module is discussed elsewhere in this paper. Other strategies, such as a shared read-only block device that is shared amongst VMs on the same physical machine, help present VMs with a vast amount of storage without traversing the network stack as would be required for an NFS equivalent. The shared block device lives on RAM on the physical machine and VMs mount this device read-only. If the virtualization technology can take advantage of Execute-In-Place (XIP), programs can be executed directly without the need of a memory copy. Regardless, a shared RAM based storage system gives over a thousand virtual machines a fast filesystem with which to access data.

Accommodating the diversity and changes in virtualization technology, vmatic is designed to be impartial to any particular hypervisor implementation. Execution arguments to a virtual machine contain little system configuration information and is primarily limited to the kernel location and the memory allocation. Configuration information is passed directly into Linux through Kernel command line arguments. Exploiting this type of communication allows vmatic to keep the configuration framework virtual machine independent.

1) DNS: In the default DNS configuration, vmatic establishes a zone of authority for each physical host on the cluster, making that node the master authoritative server for the virtual machines within it. The DNS configuration files are built as part of the boot process. Slave DNS servers also exist for this zone as a mechanism to emulate DNS redundancy throughout the emulated environment. In the real world, DNS backup servers should use separate power and network resources; in an HPC system this is impractical, there being one power feed per machine room, so vmatic uses the next node in the system modulo the number of nodes as its zone slave. For example, if an HPC system consists of physical nodes 1 through 100, node1 will be the Master DNS server for the .node1 zone and node2 will be the Slave for the .node1 zone; node100 will be Master for the .node100 zone and node1 will be slave for the .node100 zone. Changes to entries on the Master zone automatically notify their slaves of the change and update them accordingly. Physical hosts act as a caching name server to reduce the amount of network traffic on the system as well as the load on the authoritative DNS server. The cache has an arbitrary time-to-live parameter set to keep the system functioning in a dynamic environment.

The DNS servers used are BIND9 based and have support for Dynamic DNS enabled, allowing authorized users to update their DNS entries at will, which in turn allows a propagation of names throughout the emulated Internet environment. Users or automated programs also have the ability to register sub-domains to be registered through delegated subzones to let users manage a part of a particular zone. The dynamic nature of vmatic allows for different scenarios or experiments to be simulated once bootstrap of the emulated Internet is complete.

B. Monitoring

Monitoring on the emulated environment continues to be an area of development. As we are running on HPC platforms, we also inherit HPC-related problems associated with storage and analysis of execution results. Particularly, analysis of real time data remains challenging since it is important to avoid creating artifacts that would influence the actual results of the experiment. On a multi-million node environment, data reduction techniques must be exercised so as to reduce network congestion. Even modest amount of data generated from each node can result in a self denial-of-service.

1) The Pushmon Monitoring Tool: Currently, our main monitoring and data collection mechanism has been a tool we developed called Pushmon. Pushmon is a hierarchical monitoring program built from Supermon, a cluster monitoring system developed at Las Alamos [17]. Like Supermon, Pushmon uses S-expressions to describe the data, and is designed for hierarchy with Pushmon nodes functioning as both clients and servers. Unlike Supermon, Pushmon relies on a push model, with data being periodically pushed from the leaves to the root. Like Supermon, Pushmon uses S-expressions to describe the data, and is designed for hierarchy with Pushmon nodes functioning as both clients and servers. However unlike Supermon, Pushmon relies on a push model, with data being periodically pushed from the leaves to the root. Supermon used a ”pull” model, in which data collection was initiated by a request from a central collector. On multi-thousand node systems this model had scaled extremely well, far better than existing ”push”-based models such as Ganglia: Supermon can sample thousands of nodes at several hz., whereas Ganglia, on the same scale is limited to 1/600 hz. Nevertheless, the pull model did not seem a good fit to one million nodes.

Pushmon is also self-configuring, with the nodes using a low-cost computation to determine where their parent in the

tree resides, up to the root. Finally, Pushmon is designed not just to group S-expressions together, but also to perform computations on the S-expressions so as to reduce the data load on the network. The computations to be performed can themselves be defined by S-expressions and interpreted, allowing a great deal of flexibility. Data load on the network is also reduced when the VMs' relationship to their host OS is taken into account. When considering the fast communication path between a VM and its host OS, Pushmon can be used as an effective aggregator to collect messages from the child VMs before forwarding the messages up the tree.

C. Other Approaches to Data Collection and Visualization

Despite the care with which we designed Pushmon to be efficient, any cpu cycles or memory devoted to monitoring and data collection comes at the expense of the emulation itself, and this remains a big concern given the extent to which we are overcommitting resources in our system. Hence, a major focus of our work going forward will be to develop even more scalable techniques for monitoring and analysis which require as few resources as possible. Directions we are exploring include conducting aggressive in situ analysis and reduction of data at each physical node, and "entropy aware" data collection methods that only collect data when it departs from what is expected.

A completely different mechanism for collecting data would be to pause the entire emulation, take snapshots of particular VMs, and then resume the emulation. By taking regular snapshots, we can gain an understanding of what happens to VMs as the experiment progresses.

Scalable visualization remains a long term goal of our research. We are seeking ways to visually represent both the functioning of the emulation (how many VMs are up, how much memory, cpu, and network bandwidth is in use at the physical nodes) and the progress of the experiment itself (e.g., nodes joining and leaving the botnet, traffic originating from the botnet compared to other traffic, dissemination of commands through the botnet).

D. Management of Large Numbers

Past projects, bproc, XCPU.

1) gproc: Gproc is a reimplementation and combination of the best parts of the LANL version of bproc[7] and xcpu[11]. Like bproc, gproc uses a push model, in which the user specifies a program to be run, and the program is pushed to a server daemon on a node and set up and started by that server daemon. In contrast, ssh uses a pull model, in which a command string is sent to each node and the node is responsible for pulling the program and all its libraries to itself, usually via NFS. Push allows more coordinated control of data movement, whereas pull, even in systems as small as a few hundred nodes, can lead to contention for the file system, as hundreds of nodes all vie to scan the same directories and read the same data blocks. Network utilization for pull models is also typically much lower, which increases the time to start a program and move its data files. Finally, for our 1 gigabyte virtual machine image files, pull data movement requires millions of RPCs, each of which involves a high latency request/response cycle. Push on the other hand provides a more ordered set of network operations, lower file server load, and higher network utilization, as the request/response RPC transactions of the pull model are replaced by TCP streams. We have done extensive measurement of push vs. pull models over the last decade, most recently on our KANE cluster and on Blue Gene/P, and for our uses the push model is superior.

The push model can also be cleanly extended to support hierarchy, since it uses streams, not Remote Procedure Calls. While the construction of hierarchical file systems is still a research project, hierarchical push model systems like LANL's bproc implementation have been in use for 10 years.

Like the LANL bproc software, gproc has support for hierarchy. The server daemons can be arranged as a tree, so that in a system of **n** nodes, no server daemon is ever pushing a program to more than, eg, $\sqrt{n}$ nodes. Unlike bproc, there is no kernel component to gproc, which makes it more portable– in fact, it is written entirely in Google's Go programming language, and compiles to a single statically-linked binary.

Unlike bproc, the hierarchy in gproc can be as deep as needed; in a system of 32,000 nodes gproc can be arranged with a 3-level tree, such that no daemon is ever talking to more than 32 daemons. Here, the tree spawn mechanism used in gproc is similar to that of xcpu. [11] However, instead of the ad-hoc command tree spawn technique that xcpu uses, Gproc sets up a persistent tree of servers that reduces the tree spawn overhead. Finally, Gproc uses intermediate nodes in the tree to aggregate I/O from remote processes, instead of counting on the top-level command to aggregate I/O as in BProc. Figure 1 shows an example of such a tree, as used in the KANE cluster, in which commands and files are passed from the master node to every twentieth node. These level 1 slaves then pass the commands and files on to the 19 nodes under each of them and relay the output from their subnodes back to the master.

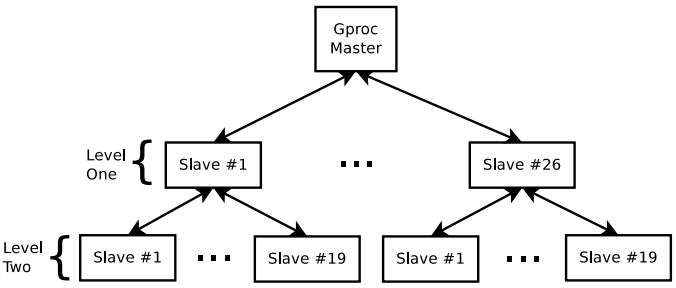

Fig. 1. The KANE cluster's default gproc hierarchy

Users frequently need to push input files along with the program to a node. Gproc is like xcpu in that it has support for pushing these additional files, via a simple command-line switch. Even directories can be pushed, by specifying their path in the command line.

Gproc preserves the file system hierarchy of the files and directories it transfers. On the remote node, the file system tree is reconstructed to avoid clashes between file names. Consider

a user running in some part of their home directory, e.g. `src/megatux`. If the user runs `/bin/date`, the remote gproc hierarchy will consist of a directory tree including `/home/$user/src/megatux`, as well as the requisite `/lib`, `/usr/lib` and `/bin` directories.

A common problem in cluster management systems is getting rid of the files created by users. The problem is even more serious when working with malware; it is important to clean the node up after a program is done. The simplest solution is to reboot the node each process is run, but that is not always practical or even desirable.

Gproc uses a Linux mount type called *process private mounts*. Process private mounts, as the name implies, are not visible outside the process that performs them – they are only visible to the process and its children. Transferred files are placed in process-private hierarchy mounted on /tmp/xproc. When the process and its children exit, the mount is garbage collected by the kernel; in other words, the unmount comes for free. Since the process-private mount is on a ram disk file system, once these processes exit, all their files disappear. Process-private mounts are a very powerful way to ensure that the files used by a process are gone when the process is gone; the clumsy and failure-prone file cleanup of, e.g., PBS, is not needed. Note that if file persistence is needed, a program can copy a file to, e.g., /tmp, for other subsequent processes to see.

Configuration is a complex problem. The LANL bproc configuration file would be impractical for systems consisting of millions of nodes, for example. Gproc uses a form of computational configuration, and allows several different configurations to be contained in one gproc binary. We call these configurations **locales**. Locales are in fact sets of simple Go functions (in Go terms, an interface, specified as one package per locale). These interfaces return information about a configuration. The **kane** locale, for instance, defines a tree hierarchy, with 1 root node, 26 nodes on the next level, and 20 nodes for each of the 26 nodes on the level below that.

For most locales, a node's identity and all other information can be computed from its IP address or some other unique property of the node. There is no configuration file to read. This technique is particularly useful on systems such as Blue Gene or the Cray XT series, where a node's location in the 3D torus is easily determined, and from there all other information can be computed.

A node can compute which level of the tree it must inhabit and which node is directly above it in the tree. The use of programmatic configurations allows interesting flexibility; rather than trying to design a configuration file format to encompass all possible configurations, configuration information is instead expressed through functions.

Gproc has proven to be fast and efficient. For example, on a 520-node cluster with only Gigabit Ethernet, we are able to move a 1 Gigabyte DVD to all 520 nodes at an effective bandwidth of 16 Gbits/second, thanks to the gproc's tree structure. We are thus able to get 16 Gigabit ethernets for the price of one.

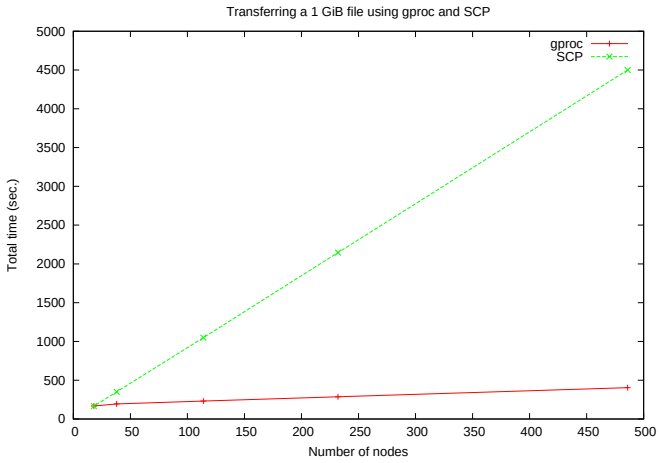

Fig. 2. Transfer times for gproc and SCP

Gproc was much more effective than any alternatives. A 1 GiB file was transferred to different numbers of nodes using gproc and SCP to compare the efficiency of the two methods. As Figure 2 shows, gproc scales considerably better than SCP for large numbers of nodes, completing approximately 10 times faster. We did experiment with hand-building a tree-structured copy with ssh/scp, but it was hard to make it reliable and it was still not as fast as gproc. We also experimented with using NFS (for timing purposes) but it never ran to completion; NFS does not handle this amount of activity very well on Linux.

IV. KANE Testbed

In Sandia California's Network Research Laboratory, we have created a cluster known as KANE which stands for the Knowledge Acquisition Network Emulation system. The system is unique from other HPC platforms in that it was purchased using true off the shelf commodity PC's for less than $500,000 (including auxiliary hardware) and contains no exceptional message passing interconnect. The nodes are comparable to home desktop PC's that are connected to the Internet using a single Ethernet connection. Using the software we have developed, each KANE node is capable of booting 1024 virtual machines providing us with a low cost cluster of over half a million nodes. KANE serves as our dedicated testbed environment which allows us to prototype experiments prior to running on bigger systems like Oak Ridge National Laboratory's Jaguar Super Computer, where our time on the system is more scarce. The KANE network also contains a heterogeneous environment comprised of Linux, Windows and 900+ ARM Cortex-A9 devices running Linux/Android operating systems to represent the increased role of mobile devices on the Internet.

KANE differs from other network testbeds in that it is primarily focused on scaling, leveraging virtualization technology. Unlike the DETER testbed, which is distributed across a geographic region, the KANE testbed is isolated from the

outside Internet and is entirely contained within the Network Research Laboratory in California.

A. Hardware

The KANE cluster is composed of 13 shelving units, which we call racks, of 5 shelves each. Each rack contains 40 compute nodes, a gigabit Ethernet switch, and a PDU. These racks act as the basic unit of the KANE cluster and are interconnected using a central Enterasys switch. They are managed using a front-end node called "cesspool." Figure 3 illustrates this layout.

Cesspool provides a number of services for the cluster. First and foremost, it acts as a barrier between the potentially harmful programs running on KANE and the outside network. Nodes are powered on and off using "powerman" from LLNL. The nodes boot using DHCP and tftp to load a minimal memory-resident Linux system. Cesspool also serves as the "master" node when running gproc.

KANE's interconnect is simple gigabit Ethernet. Each rack contains a 48-port switch, which is connected to the 40 nodes, the PDU's Ethernet interface, and the central Enterasys switch. While individual compute nodes only have a single gigabit interface, the head node (cesspool) is equipped with 4 channel-bonded interfaces to provide greater throughput.

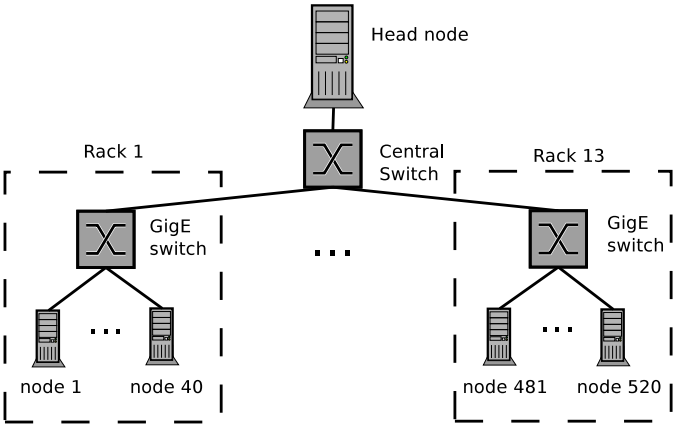

Fig. 3. This figure illustrates the basic layout of the KANE cluster.

V. MegaWin

Megawin is a platform for running large numbers of Windows images on Megatux. Windows does not permit the high degree of controllability that comes with an open source operating system such as Linux, and we have found that a ten Mbyte OS image is simply not possible. Again, due to its closed nature, it is not possible to modify Windows to run as a paravirtual guest: we must use full virtualization, which imposes costs in both memory usage and performance. We currently use KVM[10] to support Windows guests.

Hence, in Megawin, we combine two strategies: cutting down the size of the Windows image, as much as possible; and using Linux and KVM capabilities to the maximum extent possible.

A. Cutting down Windows

As mentioned above, Megatux is targeted to cluster nodes that are memory-only. Memory-only operation improves image file access and greatly simplifies the problem of wiping the machine, but complicates the problem of managing Windows images, because everything is in RAM. Windows can be considered to have two footprints: static and dynamic. The static image is the disk image which Windows boots and which is held in the root file system. The dynamic image is the memory Windows grows to occupy as it runs.

A bootable Windows image is for a virtual machine is contained in a file and in standard usage configured for only one machine: Windows image files contain a lot of per-machine information. Image files can be reduced to 1 Gbyte, but further reduction is very difficult.

It is not possible to boot large numbers of Windows VM guests if they each require a 1 Gbyte image file. MegaWin allows many guests to share a common Windows image. Further, for supporting quick boot, MegaWin can boot from a "frozen" image. To create a frozen image, MegaWin support software takes a snapshot of an almost-booted Windows and stores that snapshot. Per-instance information is generated once the snapshot is unfrozen, so that each guest gets its own personality, including network configuration. MegaWin can hence boot up to 200 Windows instances on a machine with only 12 Gybtes of memory.

Some issues affect both the static and dynamic footprint. To minimize these factors, we use the Windows Embedded version; further, we replaced the standard huge desktop with the bblean desktop[2]. The result is a Windows image that consumes only one Gbyte of disk for the static footprint, and 512 Mbytes for the dynamic footprint. Thanks to the use of freezing an image, we can boot each image in a few seconds.

Clearly, one can not boot 100 512Mbyte images on a system with only 12 Gbytes of memory. To get further VMs booted we exploit new capabilites of Linux virtualization, in particular a new software system called KSM[21]. KSM, as the authors describe it, "is code running in the Linux kernel scanning the memory of all the virtual machines running on a single host, looking for duplication and consolidating"[21]. KSM accomplishes this by periodically scanning all pages that are eligible for deduplication and merging identical ones into copy-on-write pages. The use of KSM is especially effective with our workload, as there exists a large amount of mergable data across hundreds of identical, with the exception of some runtime activity, virtual machines.

The use of KSM with a dataset approaching 100GB introduces some key problems. KSM cannot scan and merge pages faster than we can allocate them through launching new virtual machines. To facilitate this, we modified the KSM interface to force the KSM thread to only scan memory belonging to processes that we indicate. This allows us to focus KSM on newly create virtual machines during launch, or on key virtual machines that we know are better matches for deduplication

[2]http://bb4win.sourceforge.net/bblean/

during runtime. KSM can operate on any number of processes that we inform it to at any point in time. The result is the ability to more intelligently manage significantly over-budgeted memory. Figure ¡X¿ illustrates launching many Windows 7 virtual machines on a host with KSM. In the default KSM usage, we launch virtual machines until we run out of physical memory, and block until KSM merges enough memory to continue launching. With our modification, we can force KSM to focus on newly launched virtual machines, which saves time and maintains enough free memory to avoid out of memory events when running virtual machines become more active. This result is more pronounced as the amount of volatile (rapidly changing) pages increases. In practice, we are able to launch ten virtual machines at a time, focusing KSM on all ten. When KSM completes a full scan, we launch another ten virtual machines.

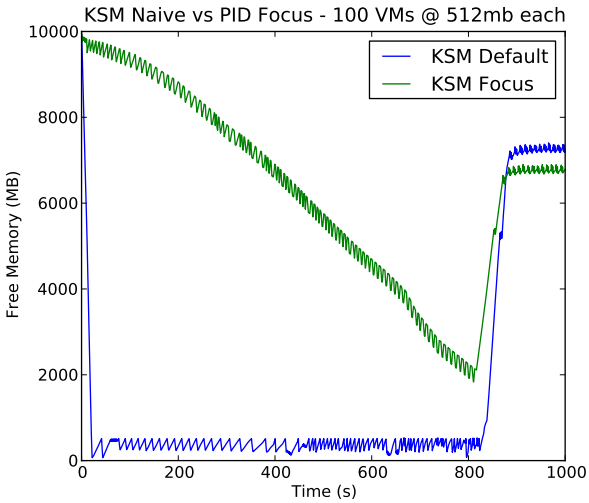

Fig. 4. KSM: Free memory vs. Time

B. Reverse Engineering and Preparation of Malware Samples

Traditionally, studying malware has required significant forensics and reverse engineering analysis, dissecting and probing the sample from multiple avenues [13]. These techniques are simply not feasible when looking at botnets and malware at scale in a traditional HPC environment. There are numerous reverse engineering challenges that must be overcome before deploying a live malware sample in an HPC setting. Current malware uses numerous custom protection mechanisms that are often tied to a specific operating system, making it difficult to provide an overarching automated reversing solution. Malware takes this a step further by breaking specifications, exploiting implementation errors in loaders, and using various anti-tamper techniques. Since malware is so closely coupled to the system for which it was designed, taking a random malware sample and attempting to run it in an HPC environment may fail. When attempting to execute malware on a platform for which it was not specifically designed, analysts must often remove the protective features of an executable that prevent it from running or expose the platform's abnormalities to the sample. It is common for malware to detect that it is executing in a virtual machine and take responsive action by either altering its behavior or refusing to run at all [4].

The Storm worm is an example of a piece of malware that attempts to identify when it is executing on a platform typically used for analysis such as a virtual machine [4]. While our specific platform uses LGuest and KVM to host the guests, malware could just as easily detect these platforms and alter their behavior. The Spybot malware contains a corrupted PE header as a form of protection [14]. The PE header is slightly modified in a way that technically violates the PE32 specification, however the Windows XP loader is lenient enough that it will properly load and execute the malware. However, WINE and other tools that adhere to the PE32 specification will reject the sample. In this case, removing the protection mechanisms from the malware allowed us to execute the sample in our testbed environment. While time consuming and difficult, the effort allowed us to properly execute the Spybot sample and subsequently allowed for study at scale in our HPC environment.

Reverse engineering and manually modifying each piece of malware prior to study is not a scalable solution. Our ultimate goal is to create a platform that is robust enough that malware will run without modification on our HPC platform without the need to individually modify each sample. This will allow us to focus on on studying botnets at scale, and processing larger quantities of samples that are captured from the wild.

C. Performance

Need numbers from Andrew.

D. High Fidelity

VI. Conclusions and Future Work

In this paper we presented an approach to achieving realistic scale in emulation of botnets in a laboratory setting. Our approach builds on lightweight virtualization technology and scalable cluster management tools. While our cluster management tools owe their heritage to tools familiar in the HPC world, we have had to make significant modifications to boot, configure, and manage the very large numbers of VMs with which we are working.

Our research is directed at understanding Internet-scale phenomena. We are starting with botnets because botnets implement protocols and capabilities we are interested in understanding. We have begun to perform experiments with real botnets, and in our first such experiment we ran an instance of the Virut botnet with 62,000 members. The individual bots ran on a Windows 7 image which we were able to make significantly smaller than is typical. We determined that the bots, once booted, registered with the IRC command and control channel, and we could issue commands to the bots.

The tools and techniques which we reported here were developed on a specially built cluster in our laboratory. However, we have designed these tools and techniques with the vision to run them on the largest supercomputers available,

and preliminary experiments we have conducted on the Jaguar supercomputer at Oak Ridge National Laboratory indicate that our approaches will work on such platforms. Therefore, we see no reason why emulations of botnets with millions of nodes should not be possible using our approach.

Significant challenges remain to be addressed to make emulation with millions of nodes a viable adjunct to other research methods in studying Internet-scale phenomena. First among these are developing scalable methods for visualization and analysis of data. While we have a tool, Pushmon (described above), for monitoring, we are still limited in the amount of information we can collect from each VM without contending for resources with the emulation experiment.

Simulation and Emulation of the Internet is a valuable tool for gaining insight into its functionality and the impact of proposed changes. The benefit of performing emulations as opposed to simulations is the level of fidelity an emulated environment can provide. Real bug for bug compatible OS instances are used. Components may be real or virtualized but either method should provide the same functional capabilities. The virtual Cisco routers run an actual Cisco IOS software image which routes real network traffic across virtual machines or real hardware.

Performing measurement or experimentation directly on Internet is an indispensable tool for understanding botnets and other malware, but it does have drawbacks. We can only take measurements on the present Internet with its existing protocols and architectures, not possible alternatives. Experimentation is also valuable, but the nature of experiments we can conduct on the existing Internet is constrained by the necessity not to interfere with the Internet's function or to cause harm to other Internet users. Experimentation on smaller physical networks intended to replicate Internet functionality has been a valuable tool, but the scale of the largest experimental networks is at least five orders of magnitude smaller than that of the Internet, and many phenomena of interest in the real Internet do not occur at the scale of the much smaller experimental networks. Furthermore, experimental networks using real networking equipment and real hosts are quite costly to build compared with simulation.

Emulation enables a highly repeatable, flexible test laboratory for conducting experiments. When an experiment is executed, it could also be checkpointed during critical events allowing staff to analyze the global state of the network. Events such as the 2007 botnet attack which took down many important network services in Estonia would be interesting to re-enact on the internet emulator. During the 2 week attack, companies resorted to blocking all international traffic in order to keep their servers from crashing, essentially cutting themselves off from the rest of the world.

REFERENCES

[1] Paul Barford and Mike Blodgett. Toward botnet mesocosms. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets*, pages 6–6, Berkeley, CA, USA, 2007. USENIX Association.

[2] T. Benzel, R. Braden, D. Kim, C. Neuman, A. Joseph, K. Sklower, R. Ostrenga, and S. Schwab. Experience with deter: a testbed for security research. In *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2006. TRIDENTCOM 2006. 2nd International Conference on*, pages 10 pp. –388, 2006.

[3] Joan Calvet, Carlton R. Davis, José M. Fernandez, Jean-Yves Marion, Pier-Luc St-Onge, Wadie Guizani, Pierre-Marc Bureau, and Anil Somayaji. The case for in-the-lab botnet experimentation: creating and taking down a 3000-node botnet. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 141–150, New York, NY, USA, 2010. ACM.

[4] Xu Chen, J. Andersen, Z.M. Mao, M. Bailey, and J. Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177 –186, june 2008.

[5] David Dagon, Guofei Gu, Christopher P. Lee, and Wenke Lee. A taxonomy of botnet structures. *Computer Security Applications Conference, Annual*, 0:325–339, 2007.

[6] Carlton Davis, Stephen Neville, Jos Fernandez, Jean-Marc Robert, and John McHugh. Structured peer-to-peer overlay networks: Ideal botnets command and control infrastructures? In Sushil Jajodia and Javier Lopez, editors, *Computer Security - ESORICS 2008*, volume 5283 of *Lecture Notes in Computer Science*, pages 461–480. Springer Berlin / Heidelberg, 2008. 10.1007/978-3-540-88313-5_30.

[7] Erik A. Hendriks and Ronald Minnich. How to build a fast and reliable 1024 node cluster with only one disk. *The Journal of Supercomputing*, 36(2):171–181, 2006.

[8] Brent ByungHoon Kang, Eric Chan-Tin, Christopher P. Lee, James Tyra, Hun Jeong Kang, Chris Nunnery, Zachariah Wadler, Greg Sinclair, Nicholas Hopper, David Dagon, and Yongdae Kim. Towards complete node enumeration in a peer-to-peer botnet. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*, ASIACCS '09, pages 23–34, New York, NY, USA, 2009. ACM.

[9] Chris Kanich, Christian Kreibich, Kirill Levchenko, Brandon Enright, Geoffrey M. Voelker, Vern Paxson, and Stefan Savage. Spamalytics: an empirical analysis of spam marketing conversion. In *Proceedings of the 15th ACM conference on Computer and communications security*, CCS '08, pages 3–14, New York, NY, USA, 2008. ACM.

[10] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium*, volume 1, pages 225–230, 2007.

[11] Ronald Minnich and Andrey Mirtchovski. Xcpu: a new, 9p-based, process management system for clusters and grids. In *CLUSTER*. IEEE, 2006.

[12] Chris Nunnery, Greg Sinclair, and Brent ByungHoon Kang Kang. Tumbling down the rabbit hole: Exploring the idiosyncrasies of botmaster systems in a multi-tier botnet infrastructure. In *Proceedings of the 4th Usenix Workshop on Large-Scale Exploits and Emergent Threats*, Berkeley, CA, USA, 2011. USENIX Association.

[13] PLACEHOLDER, editor. *PLACEHOLDER*, PLACEHOLDER.

[14] PLACEHOLDER, editor. *PLACEHOLDER*, PLACEHOLDER.

[15] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. A multi-perspective analysis of the storm (peacomm) worm. Technical report, SRI International, October 2007.

[16] Phillip Porras, Hassen Sadi, and Vinod Yegneswaran. A foray into conficker's logic and rendezvous points. In *In USENIX Workshop on Large-Scale Exploits and Emergent Threats*, 2009.

[17] M.J. Sottile and R.G. Minnich. Supermon: a high-speed cluster monitoring system. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 39 – 46, 2002.

[18] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, pages 148–162, New York, NY, USA, 2005. ACM.

[19] Rhiannon Weaver. A probabilistic population study of the conficker-c botnet. In Arvind Krishnamurthy and Bernhard Plattner, editors, *Passive and Active Measurement*, volume 6032 of *Lecture Notes in Computer Science*, pages 181–190. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-12334-4_19.

[20] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar.

An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev.*, 36:255–270, December 2002.

[21] Chris Wright. Ksm: A mechanism for improving virtualization density with kvm. In *linuxcon2009*, 2009.