SAND2011-2849C

Large-scale Botnet Analysis on a Budget

Ron Minnich, Andrew Sweeney, Kristopher Watts, Don Rudish
John Floren, David Fritz, Keith Vanderveen, Yung Ryn Choe, Casey Deccio
Sandia National Laboratories

Abstract

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. Having a controlled environment to analyze the spread of infectious malware and behavior of botnets at Internet scale is essential to better understanding botnet behavior, effectively dismantling them, and designing countermeasures. In this paper we present an architecture that we have created to enable these capabilities. Using novel techniques, we are able to boot up to 1,200 Linux virtual machines (VMs), or 200 Microsoft Windows 7 VMs on a single machine, by over-committing the processor and memory resources of the system to an unusual degree. We discuss our application of these techniques to achieve large-scale emulation of malware for botnet analysis and describe our initial findings.

1 Introduction

Botnets are complex, distributed systems consisting of many tens of thousands of individual instances of malware or *bots* which, once connected, are resilient, self-healing, controllable from a central place, and sometimes capable of sophisticated autonomous behavior. The owners and operators of botnets (known as *botherders*) use them to send spam, steal login credentials and other information from victims, and engage in other criminal activities [1]. Botnets are widespread on the Internet and show few signs of going away. It is likely that the reader has hosted a botnet at some point, and almost certain that the reader's organization has.

Recent botnets have shown surprising resiliency to attempts to remove them [12]. Sophisticated botnets such as Storm, Conficker, and Waledac have used peer-to-peer (P2P) networks for command and control [7, 5], resulting in an overlay network independent of the underlying Internet and geography. Such botnets are resilient to outages of either countries or single organizations, and "pulling the plug" is a challenge. Some botnets have even been known to detect probing and autonomously respond with a denial of service attack [13]. An understanding of botnet behavior at large scale in a contained environment will facilitate the defense and dismantling of botnets. We propose such a system suitable for running on a cluster comprised of commodity hardware hosting upwards of hundreds of thousands of virtual machines (VMs).

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 [17] or through insertion of an instrumented false bot controlled by researchers into an extant botnet [11, 8]. As noted by studies such as [4] and [2], however, the aforementioned techniques cannot provide the "big picture" of a botnet's operations, nor do they provide a reliable means to conduct repeatable experiments into possible means of detecting, defending against, or neutralizing botnets.

The solution, as recognized by [4] and [2], 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.

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 list the following as the primary contributions of this paper:

- Application of novel techniques to over-budget resources for scaling VM guest capabilities in a cluster environment.
- Implementation of a prototype testbed for scalable malware analysis in a contained environment.

We have developed a prototype testbed consisting of a 520-node cluster, capable of hosting 62,000 Windows 7 VMs or 600,000 Linux VMs. In this testbed we have run real botnet malware and we make observations about its behavior.

The rest of this paper is organized as follows. In Section 2, we discuss related efforts to study botnets at scale. In Section 3, we discuss challenges and solutions we've contributed towards making efficient use of computing resources to achieve large-scale performance. Section 4 describes the early malware experiments we have run in our environment, including the platform on which it was run and the behavior we observed. In Sections 5 and 6 we describe future work in this area and summarize our conclusions.

2 Related Work

Previous research into understanding the behavior of botnets has followed the approaches of reverse engineering and static analysis of captured bot code [14], breaking into an extant botnet[11], running actual bot code at a small scale[13], or creating a simulation of the botnet's network behavior and observing that [5]. 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.

2.1 Scalable platforms to host bots

Some prior research has sought to achieve scalable tools for better understanding botnets and other malware. The Potemkin project [17] used virtualization, over-subscription 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, but rather to present a large number of apparently 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, over-subscription 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. [2] 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 [19] enabled network testbeds such as DETER [3], 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 [4]. 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 reverse engineering of botnet binaries and observations of botnets in the wild in order to truly understand botnet behavior, and they were able to discover a bottleneck in Waledac's C&C traffic that likely led to the botherder's switch to common session keys and use of unsigned server commands (previously assumed to be design flaws). They state that they would not have made this discovery had they not had the capability to observe the botnet running at the thousand node scale.

Our project has similar goals to the Barford and Calvet emulation testbeds, but we were able to improve on the scale of experiments reported by both groups by more than an order of magnitude owing 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.

Both [4] and [2] demonstrated testbeds capable of holding a few hundred to a few thousand nodes of a botnet, and both correctly point out the need to study botnets at scale to truly understand them. However, we believe that even greater scale is needed, because actual botnets can consist of hundreds of thousands to millions of nodes [18]. 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 on the real Internet.

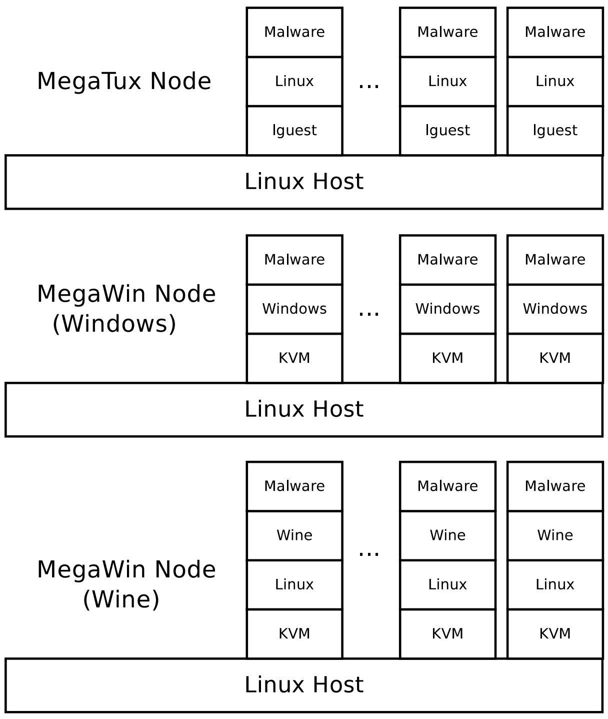

Figure 1: Architectural diagrams of a MegaTux node and MegaWin nodes based on Windows and Wine.

3 Scalable Computing Resources

Creating an environment of Internet proportions with limited physical machines necessitates the use of virtual machines (VMs). To maximize the number of virtual machines that can be deployed on each physical machine, resources such as processor (CPU) and memory on host systems must be used efficiently and even overcommitted. We achieve this by lightening the effective footprint of the host system, the virtual machine monitor (VMM) and the guest operating systems (OS). However, any optimizations must not impede the end goal of being able to run botnets at scale, monitor their behavior, and make meaningful observations.

In this section we discuss the methodology we have employed to enable deployment of large numbers of VMs on commodity hardware in a cluster environment. Our techniques aim to run both Linux- and Microsoft Windows-based malware, and we call these environments *MegaTux* and *MegaWin*, respectively, referring to the scale we aim to achieve. Architectural diagrams of our models are illustrated in Figure 1.

3.1 Host Environment

The host OS on each physical machine runs in a diskless environment. There are several reasons for this:

- although our contributions focus on the ability to scale simulation on even commodity hardware, the ability to port our system to high-end systems (e.g., Blue Gene and Cray XT) for the largest experiments is extremely useful, and these systems consist completely of diskless nodes;
- it has not proven practical to share a single disk between thousands of VMs on a physical machine; and
- were our environment ever connected to the real Internet, we would want to be able to erase any downloaded data with a simple power cycle.

Our approach is based on the oneSIS[1] platform, using diskless mode. However, rather than using a network filesystem (NFS) mounted root filesystem, we extend the oneSIS model to support a root filesystem based entirely in memory—a RAM (random access memory) disk. On clusters, the kernel and initial RAM disk (initrd) are downloaded via a preboot execution environment (PXE).

The initrd includes a kernel and initrd for the guest VMs. Both the host and guest initrd include enough files to boot the host OS. Any additional programs that need to be executed are later pushed to the node using software built in-house for efficient transfer to large numbers of cluster nodes. The result is a pure memory-based node that has no dependencies on external filesystem mounts.

3.2 MegaTux

The MegaTux project is targeted at environments of 500,000 to tens of millions of Linux kernels. It consists of a number of components. A full description of MegaTux is outside the scope of this paper, but we provide a brief overview.

3.3 VMatic

VMatic is a tool for rapidly provisioning virtual machine environments. It is an extension of Sandia's oneSIS cluster management system, modified so that nodes can be operated with a RAMdisk root, instead of a local disk or NFS root file system. We have tested scaling of VMatic to over 4096 physical nodes, and over 4.5 million nodes, as of this writing.

VMatic can operate with any virtual machine technology, although we currently only use either lguest or KVM. We initially targeted only Linux guests, but VMatic has been able to support Windows guests with no changes. VMatic can also, if needed, support mixed Windows/Linux configurations. We see no limits to the

[1]http://onesis.org/

guest OSes we deploy on VMatic; if they will work under a Linux virtual machine, they will work on VMatic.

Using a simple configuration file and command set, VMatic produces system images that can be uploaded to compute nodes via network boot. The user is able to specify the configuration of both the host and guest images. Through the use of multiple VMatic configuration files, a user can maintain separate build configurations for a variety of experiments on multiple clusters. We can do test runs on our laptops, real runs on clusters, and "hero" runs on the large Cray XT systems at Oak Ridge National Labs.

Once the physical host starts booting, it has all the information it needs to configure its own services and configure its local virtual machines. The MAC addresses, virtual Ethernet devices, and routes to other networks are all computed as a node starts up; there is no central store of MAC addresses for all the VMs. A single Linux virtual machine takes about one second to fully boot.

In addition to provisioning HPC systems, VMatic can also be used to provision lightweight VM's on a local machine. This has proven to be invaluable for our development work as it gives each team member the ability to boot their own virtual cluster on their laptop. On our laptops, we can reconfigure and start 100 new VMs in less than 30 seconds. This speed makes testing easy. Developers now have an immediate, convenient, and reliable way for performing automated testing on their code sets without consuming precious time on limited HPC resources. This mechanism was used in the creation of our first botnet prototype with successful results when we started scaling out.

3.4 gproc

Gproc is the latest in a line of process startup systems we have developed, starting with BProc[6] and continuing with XCPU[10]. Gproc uses a tree spawn mechanism similar to that of XCPU, and also moves all the libraries a given command needs to run, as does XCPU. Instead of the ad-hoc command tree spawn technique that BProc uses, Gproc sets up a persistent tree of servers that reduces the tree spawn overhead. Gproc uses intermediate nodes in the tree to aggregate I/O from remote processes.

The BProc daemon structure has two different programs: the master on a control node and slave daemons on each compute node. This model scales well to several thousand nodes but no further. Gproc daemons adopt the XCPU model: the daemons are all the same code, and can adopt the role of server or client. A process tree hence consists of a root server, intermediate daemons playing both roles, and a set of daemons at the leaves. The ultimate clients are at the leaves of the tree, i.e. individual processes. The ultimate server is at the root of the tree, i.e. the program that initiates the million or more commands. Daemons in internal nodes of the tree control processes below them, and relay data up and down the tree.

Each slave is a master of all the VM guests on its node. Hence, the process repeats for the VMs on each physical node.

```
(("MARK: 1266084142.710273")0x4336 "o0x4336 s1 #0")
(("MARK: 1266084143.272552")0x924e "o0x924e s1 #0")
(("MARK: 1266084145.387336")0x9879 "o0x9879 s1 #0")
```

Figure 2: Sample of Pushmon output. MARK: denotes the Unix epoch time from the root node and is embedded with the original messages using S-expressions.

3.5 pushmon

Pushmon is a hierarchical monitoring system built from Supermon[16]. 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. Pushmon is also self-configuring, with the nodes using a low-cost computation to determine where their parent in the tree is, up to the root. Finally, Pushmon is designed not to just group S-expressions together, as Supermon does, 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, up to and including symbolic computing. See Figure 2.

Data load on the network is also reduced when the VM's 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 their child VMs before pushing to the root minimizing load on the physical network.

We are working to build an efficient virtio[15] transport for guest to host Pushmon communications. In spite of the plethora of virtio software that has been written, there is nothing that resembles an efficient pipe. We plan to remedy this problem.

3.6 MegaWin

Unlike the case for Linux-based OSes, we do not have access to the source code for Microsoft Windows, which presents unique challenges to minimizing the resources it requires to run as a VM. First, due to its closed nature, we have no way to modify Windows to run as *paravirtual* guest—that is, one that is aware that it is running as a VM and optimizes accordingly. Therefore, in order to run Windows as a guest OS, we must use full virtualization, which imposes costs that affect both memory usage and performance. We detail our experiences running Windows under full virtualization in this section, as well as an alternative solution for running Windows-based malware using Wine [2].

3.6.1 Full virtualization

We currently use the Kernel Based Virtual Machine (KVM) software to support fully virtualized Windows guests [9]. Using this platform, we combine two strategies to maximize the efficiency of running a Windows VM: minimizing the size of the Windows OS image as much as possible; and using Linux and KVM capabilities to the maximum extent possible.

The memory-only operation of our host machines 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 memory. Windows can be considered to have two classes of memory footprints: *static* and *dynamic*. The static image is the disk image which Windows boots and which is stored as a file in the root filesystem of the host. This bootable image can be reduced to 1GB, but further reduction is very difficult, because we have no way to modify the Windows build to eliminate components not essential for our experimentation, such as the graphical user interface (GUI). The dynamic image is the memory Windows grows to occupy as it runs.

In standard usage a bootable Windows image file will only work for a single VM instance because Windows images contain a lot of per-machine information. Because booting large numbers of Windows VM guests is not feasible or efficient if each has its own 1GB image, we use the following methodology to enable many guests to share a common Windows image on a single host. Rather than using a Windows image in the "powered off" state to boot each VM, we use the "snapshot" of an almost-booted Windows VM, taken previously, as a starting point in the boot process for each VM on a host. Each VM is appropriately configured once the snapshot is resumed, so each guest gets its own personality, including network configuration. Also, this speeds up the boot process, reducing per-VM startup time from approximately two minutes to only a few seconds since much of the booting was done prior to taking the VM snapshot.

Another Windows component affecting the efficient use of computing resources is the desktop environment. Since we cannot eliminate the GUI completely, we mitigate the issue by using the Windows Embedded version and replacing the standard Windows desktop environment with the bblean desktop[3]. These changes decrease both the static and dynamic memory footprint, and the bblean desktop decreases CPU utilization because it excludes features such as the anti-aliasing of fonts.

The result of our optimizations is a Windows image that consumes 512MB for the dynamic footprint per VM and shares a 1GB static footprint across other VMs on the same host.

3.6.2 Memory deduplication

Even with the improvements to static and dynamic memory footprint, host resources limit booting anything more than a nominal number of Windows VMs on a given host, e.g., around five on a host system with 12GB of memory. To further optimize the resources of the host machine, we exploit new capabilities of Linux virtualization, in particular a new software system called KSM [20]. KSM, as the authors describe it, "is code running in the Linux kernel scanning the memory of all the VMs running on a single host, looking for duplication and consolidating" [20]. 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 mergeable data across hundreds of identical VMs, with the exception of some runtime activity.

Using KSM to increase the number of Windows VMs running on a host an order of magnitude beyond its natural limits introduces some key problems. KSM cannot scan and merge pages faster than we can allocate them through launching new VMs. To facilitate this, we modified the KSM interface to force the KSM thread to only scan memory belonging to processes that we designate. This allows us to focus KSM on newly created VMs during launch, or on key VMs that we know are better matches for deduplication 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 3 illustrates launching many Windows 7 VMs on a host with KSM. The number of distinct VMs to focus on is varied from 0 (default KSM behavior) to 5. In the most naive case, we launch VMs 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 VMs, which saves time and maintains enough free memory to avoid out-of-memory events when running VMs become

[2]http://www.winehq.org/

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

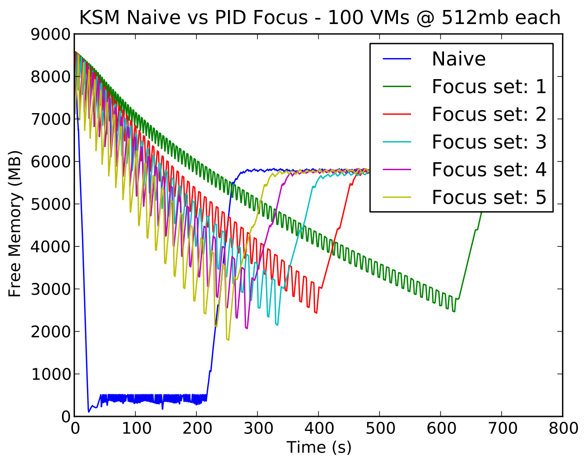

Figure 3: KSM: Free memory vs. Time

more active. In practice, we are able to launch ten VMs at a time, focusing KSM on all ten. When KSM completes a full scan, we launch another ten VMs. Using this process, we were able to boot 200 Windows 7 Embedded VM instances on a machine with only 12GB of memory.

3.6.3 Wine

In addition to our efforts to minimize the memory footprint of a full Microsoft Windows image, we have explored other alternatives to running Windows-based malware in our environment. Wine implements the Windows application programming interface (API), allowing Windows software to run in other environments, such as Linux.

Building on the MegaTux model, we run Wine on the Linux-based VMs. Standard Wine includes provisions for special features, such as graphics and sound. However, since our focus is on running Windows-based malware, we can minimize the footprint associated with Wine by eliminating libraries not essential to its execution when building the Wine binary. We reduced the size of the Wine binary about 5%, from 195MB to 186MB, using this process.

While the Wine software is not equivalent to a Windows installation, in many cases it will suit our purposes and provide a flexibility that we might not otherwise have with a Windows image. One benefit mentioned previously is that the binary itself can be reduced because its source is available. Also, Wine running in a VM environment inherits the virtualization benefits of the underlying guest OS, including reducing the memory footprint incurred by the guest OS, as well as paravirtualization capabilities. Finally, in contrast to licensing for large numbers of Windows installations, which might be costly and infeasible, Wine is freely available under an open source license.

Because Wine is an implementation of the Windows API, it might not be suitable for some scenarios, such as the spread of malware through the exploit of a vulnerability in the Windows implementation. However, for the purposes of this paper, we focus on running botnets comprised of hosts already infected with malware.

3.7 Compatibility tradeoffs

The process of minimizing the resource footprint used by VM images allows scalability through efficient use of resources. However, it also presents potential challenges with regard to execution compatibility in the slimmer environments. If programs (in this case botnet malware) executed in such environments fail to detect certain functionality, then they might: refuse to run, knowing they are subjects of analysis; behave differently from how they would in an unmodified environment; or fail to run, because of dependencies on missing components. In any of these cases, the observed behavior from macro-analysis might be misleading or counter-productive.

In the future we intend to do implement several provisions to mitigate these circumstances, making use of our in-house isolated malware testing and reverse engineering environment. First, we plan to a full-system comparison in terms of file and registry existence in a full environment compared to the virtual environment. Second, we intend to compare micro behaviors of individual malware specimens with their collaborative behavior in the full virtualization environment.

4 Experimentation

In this section we describe the prototype testbed built on commodity hardware on which we've applied the techniques discussed earlier in the paper to boot large numbers of VMs for botnet analysis. We also describe some of the behaviors we've observed in initial experimentation.

4.1 Testbed

We have created a cluster as a prototype testbed for large-scale botnet analysis This system, dubbed the Knowledge Acquisition Network Emulation system (KANE), is unique from other clusters in that it is comprised of true off-the-shelf commodity personal computers (PCs), purchased for less than $500,000 (including auxiliary hardware). The nodes are comparable to home desktop PCs connected to the Internet using a single Ethernet connection.

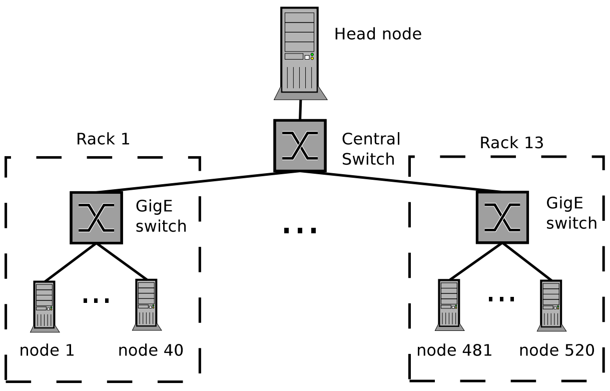

Figure 4: This figure illustrates the basic layout of the KANE cluster.

4.1.1 Hardware configuration

The KANE cluster is composed of 13 racks having five shelves each. Each rack contains 40 compute nodes and a gigabit Ethernet (GbE) switch which connects them all. All rack switches are interconnected at a central switch, and nodes are managed using a single front-end *head* node connected to the central switch. While individual compute nodes only have a single GbE interface, the head node is equipped with four channel-bonded interfaces to provide greater throughput. The head node also provides services, such as the Dynamic Host Configuration Protocol (DHCP), Domain Name System (DNS), and Trivial File Transfer Protocol (TFTP) for PXE boot. Figure 4 illustrates this layout.

KANE differs from other network testbeds in that it is primarily focused on scaling, leveraging virtualization technology. For example, KANE nodes use the Intel Core i7 processor, which makes virtualization more transparent to guest OSes. 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 our research laboratory. Network properties, such as latency, are artificially emulated or introduced using software.

KANE serves as our dedicated testbed environment. Using our setup, we can prototype experiments prior to running on bigger systems, where our time on the system is more scarce.

4.1.2 Networking

Networking for the VMs is an essential element of our testbed. Ideally our environment would support arbitrary network topologies, including routing protocols. For the present, we have focused on host scalability in a two-tier hierarchical architecture. Each host OS running on a physical node acts as a router to the VMs it hosts, and the guest VMs are networked on the same broadcast domain using a software switch run by the host OS. All host OSes maintain static routes to all other host OSes, enabling global routing in the testbed.

4.2 Experiments

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.

5 Future Work

Significant challenges remain to be addressed to make emulation of botnets with millions of nodes a viable adjunct to other research methods in studying botnets. First among these are developing scalable methods for visualization and analysis of data. While we have developed prototype tools for monitoring, the challenge is in effectively collecting and aggregating useful information from each VM without overwhelming network or computing resources, and without affecting experimentation. We intend to extend our prototype tools to incorporate these desirable characteristics.

Another capability alluded to earlier in this paper is the ability to deploy arbitrary network topologies, resembling either known networks or large networks with Internet-like characteristics, complete with dynamic routing. We intend to implement this in a future version of our toolset for managing the setup and configuration of our VMs.

6 Conclusions

In this paper we have presented an approach to achieving realistic scale in emulation of botnets in a laboratory setting. Our approach builds on lightweight virtualization technology, leveraging novel techniques for efficient use of computing resources by both Linux- and Windows-based VMs.

We have performed initial experiments with actual malware in our testbed environment built from commodity hardware. Using this relatively small cluster of 520 nodes, we successfully ran an instance of the virut botnet with 62,000 members.

While the tools and techniques reported in this paper have been developed for and prototyped on our commodity testbed cluster, they have been designed with the vision to run them on the largest supercomputers available. Preliminary experiments 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 of botnets with millions of nodes a viable adjunct to other research methods in studying botnets. First among these are developing scalable methods for visualization and analysis of data.

We hope to use our emulation testbed, KANE, to determine the optimal strategies to deal with botnets. Our plan is to infect a virtual Internet with a botnet and have it operate. We can then test counter measures, refine them, and repeat the experiment if necessary. This type of experiment has the benefit in that it allows us to conduct malware analysis in a contained, representative, and cleanable environment.

To conclude, emulation enables a highly repeatable, flexible test laboratory for conducting experiments on malware that cannot be conducted in any other way. Emulation of a whole botnet at its natural scale will allow researchers to see the big picture of how a botnet operates in a way that they cannot see either from smaller scale experiments with bot code, from simulation, or from observations of botnets on the Internet. The other approaches remain valuable and necessary, but we believe being able to run an actual sized botnet in a controlled environment will allow for a new type of experimentation that will be a much needed additional tool for researchers.

Acknowledgments

Thanks to IBM and Rusty Russel for Lguest. There have been many important contributors to the KVM software on Linux, and it is not possible to thank them all, but thanks to the authors of KVM, KSM, and QEMU.

Thanks also to Kevin Pedretti of Sandia for his valuable assistance with experiments in large-scale virtualization on the Jaguar supercomputer at Oak Ridge National Laboratory. We also wish to thank the folks at Oak Ridge who keep Jaguar running for letting us run on it, and thanks also to the Department of Energy's INSIGHT award of time on Jaguar to our group for making our experiments possible.

We also wish to thank Matt Leiniger and the Hyperion team at Lawrence Livermore National Laboratory for letting us run experiments on the Hyperion cluster there early on in our project.

This work was funded by the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories. Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the United States Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] ABU RAJAB, M., ZARFOSS, J., MONROSE, F., AND TERZIS, A. A multifaceted approach to understanding the botnet phenomenon. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement* (New York, NY, USA, 2006), IMC '06, ACM, pp. 41–52.

[2] BARFORD, P., AND BLODGETT, M. Toward botnet mesocosms. In *Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets* (Berkeley, CA, USA, 2007), USENIX Association, pp. 6–6.

[3] BENZEL, T., BRADEN, R., KIM, D., NEUMAN, C., JOSEPH, A., SKLOWER, K., OSTRENGA, R., AND SCHWAB, S. 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* (2006), pp. 10 pp. –388.

[4] CALVET, J., DAVIS, C. R., FERNANDEZ, J. M., MARION, J.-Y., ST-ONGE, P.-L., GUIZANI, W., BUREAU, P.-M., AND SOMAYAJI, A. 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* (New York, NY, USA, 2010), ACSAC '10, ACM, pp. 141–150.

[5] DAVIS, C., NEVILLE, S., FERNANDEZ, J., ROBERT, J.-M., AND MCHUGH, J. Structured peer-to-peer overlay networks: Ideal botnets command and control infrastructures? In *Computer Security - ESORICS 2008*, S. Jajodia and J. Lopez, Eds., vol. 5283 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2008, pp. 461–480. 10.1007/978-3-540-88313-5_30.

[6] HENDRIKS, E. A., AND MINNICH, R. How to build a fast and reliable 1024 node cluster with only one disk. *The Journal of Supercomputing 36*, 2 (2006), 171–181.

[7] KANG, B. B., CHAN-TIN, E., LEE, C. P., TYRA, J., KANG, H. J., NUNNERY, C., WADLER, Z., SINCLAIR, G., HOPPER, N., DAGON, D., AND KIM, Y. Towards complete node enumeration in a peer-to-peer botnet. In *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security* (New York, NY, USA, 2009), ASIACCS '09, ACM, pp. 23–34.

[8] KANICH, C., KREIBICH, C., LEVCHENKO, K., ENRIGHT, B., VOELKER, G. M., PAXSON, V., AND SAVAGE, S. Spamalytics: an empirical analysis of spam marketing conversion. In *Proceedings of the 15th ACM conference on Computer and communications security* (New York, NY, USA, 2008), CCS '08, ACM, pp. 3–14.

[9] KIVITY, A., KAMAY, Y., LAOR, D., LUBLIN, U., AND LIGUORI, A. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux Symposium* (2007), vol. 1, pp. 225–230.

[10] MINNICH, R., AND MIRTCHOVSKI, A. Xcpu: a new, 9p-based, process management system for clusters and grids. In *CLUSTER* (2006), IEEE.

[11] NUNNERY, C., SINCLAIR, G., AND KANG, B. B. K. 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.

[12] PORRAS, P. Inside risks: Reflections on conficker. *Commun. ACM 52* (October 2009), 23–24.

[13] PORRAS, P., SAIDI, H., AND YEGNESWARAN, V. A multi-perspective analysis of the storm (peacomm) worm. Tech. rep., SRI International, October 2007.

[14] PORRAS, P., SADI, H., AND YEGNESWARAN, V. A foray into confickers logic and rendezvous points. In *In USENIX Workshop on Large-Scale Exploits and Emergent Threats* (2009).

[15] RUSSELL, R. virtio: towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev. 42*, 5 (2008), 95–103.

[16] SOTTILE, M., AND MINNICH, R. Supermon: a high-speed cluster monitoring system. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on* (2002), pp. 39 – 46.

[17] VRABLE, M., MA, J., CHEN, J., MOORE, D., VANDEKIEFT, E., SNOEREN, A. C., VOELKER, G. M., AND SAVAGE, S. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), SOSP '05, ACM, pp. 148–162.

[18] WEAVER, R. A probabilistic population study of the conficker-c botnet. In *Passive and Active Measurement*, A. Krishnamurthy and B. Plattner, Eds., vol. 6032 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2010, pp. 181–190. 10.1007/978-3-642-12334-4_19.

[19] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., AND JOGLEKAR, A. An integrated experimental environment for distributed systems and networks. *SIGOPS Oper. Syst. Rev. 36* (December 2002), 255–270.

[20] WRIGHT, C. Ksm: A mechanism for improving virtualization density with kvm. In *linuxcon2009* (2009).