Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm

Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft,
Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage

Collaborative Center for Internet Epidemiology and Defenses
Department of Computer Science and Engineering
University of California, San Diego

ABSTRACT

The rapid evolution of large-scale worms, viruses and botnets have made Internet malware a pressing concern. Such infections are at the root of modern scourges including DDoS extortion, on-line identity theft, SPAM, phishing, and piracy. However, the most widely used tools for gathering intelligence on new malware — network honeypots — have forced investigators to choose between monitoring activity at a large scale or capturing behavior with high fidelity. In this paper, we describe an approach to minimize this tension and improve honeypot scalability by up to six orders of magnitude while still closely emulating the execution behavior of individual Internet hosts. We have built a prototype honeyfarm system, called *Potemkin*, that exploits virtual machines, aggressive memory sharing, and late binding of resources to achieve this goal. While still an immature implementation, Potemkin has emulated over 64,000 Internet honeypots in live test runs, using only a handful of physical servers.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*Invasive software*; C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; D.4.2 [**Operating Systems**]: Storage Management—*Virtual memory*; C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network monitoring*

General Terms

Measurement, Security

Keywords

copy-on-write, honeyfarm, honeypot, malware, virtual machine monitor

SOSP'05, October 23–26, 2005, Brighton, United Kingdom.
Copyright 2005 ACM 1-59593-079-5/05/0010 ...$5.00.

1. INTRODUCTION

The ability to compromise large numbers of Internet hosts has emerged as the backbone of a new criminal economy encompassing bulk-email (SPAM), denial-of-service extortion, phishing, piracy, and identify theft. Using tools such as worms, viruses and scanning botnets, the technical cadre of this community can leverage a handful of software vulnerabilities into a large-scale virtual commodity — hundreds of thousands of remotely controlled "bot" hosts — that are then used, resold or leased for a variety of illegal purposes [13].

While a range of tactical countermeasures can be employed against the *uses* of these hosts (e.g., SPAM filters, DoS defenses), combating the underlying infestation requires first understanding the means and methods used to compromise and subsequently control the bot population. By far, the most important tool for this purpose is the *honeypot.* Put simply, a honeypot is a network-connected system that is carefully monitored (and frequently left unprotected) so that intrusions can be easily detected and precisely analyzed. Such information is then used in turn to create antivirus signatures (to limit further growth), to develop disinfection algorithms (to eradicate existing infections), and to support criminal investigation and prosecution.

In practice, however, deploying a large network of honeypot systems, a *honeyfarm*, exposes a sharp tradeoff between *scalability*, *fidelity*, and *containment.* At one extreme, so-called "low-interaction honeypots" can monitor activity across millions of IP addresses at a time. Such honeypots achieve this scalability by only emulating the network interface exposed by common services and thus maintaining little or no per-honeypot state [27, 40]. However, since these systems do not execute any code from native applications or operating systems they are unable to determine if an attack is effective, why a given exploit works, what the payload does, or how the compromised system will be controlled, updated, or used. Indeed, such systems may be unable to even *elicit* attacks that require multiple phases of communication.

In contrast, "high-interaction honeypots" execute native system and application code and thus can capture malicious code behavior in its full complexity [12, 32]. Unfortunately, the price of this fidelity is invariably quite high. In their simplest form these systems require a single physical host for each monitored IP address, and while some systems use virtual machines to reduce this requirement, it is rarely cost-effective to support more than a few thousand hosts.[1] Finally, all of these systems struggle to balance the need for containment — preventing compromised honeypots from attacking third-party systems — and the desire to allow unfettered network access to enhance system fidelity.

In this paper, we describe a honeyfarm system architecture that can scale to design points previously reserved for stateless monitors (hundreds of thousands of IP addresses), while offering fidelity qualitatively similar to high-interaction honeypots. The heart of our approach is to dynamically bind physical resources to external requests *only* for the short periods of time necessary to emulate the execution behavior of dedicated hosts. By exploiting idleness at the network layer and physical memory coherence between hosts, we argue that the resource requirements of emulating an Internet host can be reduced by up to six orders of magnitude in practice.

To demonstrate our approach, we have implemented a prototype honeyfarm system, called *Potemkin*, based on a specialized network gateway and a virtual machine monitor derived from Xen. At the network layer, individual flows are dispatched to a collection of honeyfarm servers which, in turn, dynamically instantiate new virtual machines to assume the role of each destination IP address. To reduce overhead and increase the number of VMs supported on each honeyfarm server we propose two techniques: *flash cloning* and *delta virtualization*. The former instantiates new VMs quickly by copying and modifying a host reference image, thus avoiding the startup overhead of system or application initialization, while the latter optimizes this copy operation using copy-on-write, thus exploiting the memory coherence between different VMs. Finally, our network gateway supports a wide range of containment policies allowing customizable tradeoffs between potential liability and greater fidelity.

In the remainder of this paper we describe our design in more detail and our initial experiences in its use. Section 2 provides background information on honeypots and places our approach in context with previous related work. We outline our system architecture in Section 3, followed by a more elaborate description of the Potemkin prototype in Section 4, and present the results of our initial experiences in Section 5. Finally, we discuss the limitations and challenges of our approach in Section 6, and summarize our overall findings in Section 7.

2. BACKGROUND AND RELATED WORK

The Honeynet Project gives one of the few precise definitions for the term *honeypot* [12]:

> A honeypot is an information system resource whose value lies in unauthorized or illicit use of that resource.

This basic idea undoubtedly predates computers as it was an established counter-intelligence technique during much of the late 20th century. However, the idea was first popularized in the computing community via Bill Cheswick's paper "An Evening with Berferd" and the contemporaneous novel, *The Cuckoo's Egg* by Cliff Stoll [5, 31]. In these early accounts, attackers were ensnared by presenting the illusion that they had broken into a system of interest. However, these lone honeypots were very low tech – in Cheswick's case, the illusion was manually generated! Dedicated honeypot hosts were soon introduced into practice and by the mid-1990s both the term *honeypot* and the basic approach were well established.

Over the past decade, honeypots have emerged as the principal tool for gathering intelligence on new means and methods used by attackers. The underlying strength of the approach lies in its simplicity: since a honeypot receives few legitimate communications, an attacker's actions are easy to discern. Moreover, since honeypots are carefully provisioned they provide an idealized monitoring capability to the defender. However, this same simplicity is also a weakness. While a defender can try to make a honeypot server externally attractive, it is not possible to force an attacker to take the bait. Thus honeypots are most useful for capturing indiscriminate or large-scale attacks, such as worms, viruses or botnets, rather than very focused intrusions targeting a particular host. We discuss this issue further in Section 6, but in this paper we focus primarily on the former use.

Over the last decade there have been a wide variety of honeypot systems built, both academic and commercial. Roughly speaking, these systems have fallen into two broad categories, *low-interaction* and *high-interaction*, reflecting an inherent tension between fidelity and scalability.

As the name suggests, low-interaction honeypots offer minimal interaction with the attacker and typically only emulate portions of an emulated host's network behavior. Perhaps the most extreme point in this design space is the *network telescope*, which passively monitors inbound packets directed at large ranges of quiescent IP address space [22, 23]. The principal strength of this approach is its scalability — network telescopes have successfully monitored ranges in excess of 16 million IP addresses and have been used to track major worm outbreaks such as CodeRed and Slammer [20, 21]. Unfortunately, since a network telescope is entirely passive it cannot complete the TCP handshake and thus elicit the payload or exploit of most attacks — let alone analyze what they would do.

To address this deficiency, some honeypots employ *active responders* to reply to inbound packets appropriately and elicit a more complex transaction. The most basic of these approaches simply transmits a SYN/ACK sequence in response to TCP SYN packets [1, 30]. More complex systems, such as Yegneswaran *et al.*'s iSink, implement per-protocol responders that can more precisely emulate the network behavior that a real attacker would experience [26, 40]. Both of these approaches can be designed in a purely stateless fashion, approaching the scalability of a passive monitor. Finally, some low-interaction honeypots, such as Provos' widely used *honeyd* system, do maintain per-flow and per-protocol state to allow richer emulation capabilities [27].

However, none of these systems execute the kernel or application code that attackers seek to compromise and therefore they cannot witness an exploit in action, nor observe the attacker's actions after a host is compromised. To address this need, high-interaction honeypots offer an execution environment identical or similar to a real host and thus allow the attacker's behavior to be monitored with high fidelity. The simplest of these approaches uses individual servers for

[1] The largest high-interaction honeypot system we are aware of is Symantec's DeepSight system, which uses 40 servers executing VMware to emulate 2000 IP addresses [24].

each monitored IP address [12]. These are, for all intents and purposes, identical to real systems and therefore represent the gold standard for a honeypot. Unfortunately, this approach is extremely expensive to scale and to manage. For this reason, researchers and practitioners alike have turned to modern virtual machine monitors (VMM) — such as VMware, Xen, Virtual PC, and User Mode Linux — to instantiate (multiple) honeypots on a single server.

The virtual machine environment offers several benefits for implementing a honeypot. First, it is easy to manage, since most VMMs allow individual VMs to be loaded, frozen or stored on demand. Indeed, it is this management advantage that has driven the use of VMs in honeypots in operational deployments (typically under the name *virtual honeynets*) [12]. As well, VMMs also offer an ideal platform for instrumenting and monitoring the activities within a compromised system, including interactive input, memory and disk allocation, patterns of system calls and the content of endpoint network flows [4, 8, 11, 15, 18]. Even stealthy malware, such as kernel rootkits, can be detected since a VMM executes outside the compromised environment. Finally, VMMs allow multiple honeypots to be implemented by a single server — thus reducing deployment costs. For example, Dagon *et al.* propose supporting up to 64 VMs per physical host using VMware's GSX server [8]. In fact, Whitaker *et al.* demonstrate scalability to the *thousands* of simultaneous virtual machines using a customized VMM [38]. However in this case the hosted "guest" systems were small customized libraries rather than a transparent commodity operating system and application environment required by a honeypot. In practice, we are unaware of any VM-based honeypot deployment that uses more than 8 VMs per physical host or scales beyond 2000 IP addresses in total.

3. ARCHITECTURE

In its purest incarnation, a honeyfarm is simply a collection of monitored Internet hosts running common operating system and application software. It is exactly this high-fidelity abstraction that we strive to preserve. However, our other goals, of scalability and containment, demand a significantly modified realization. In this section, we discuss how these requirements shape our architecture and describe how our network gateway and virtual machine monitor components achieve these.

3.1 Scalability

Addressing scalability first, our key insight is that the work required to emulate a host is ultimately driven by external perception. To paraphrase Bishop Berkeley: If a host exposes a vulnerability, but no one exploits it, was it really vulnerable? We argue that since dedicated honeypots have no independent computational purpose, only tasks driven by external input have any value.

By this metric, a conventional network of honeypot servers is horrendously inefficient. First, most of a honeypot's processor cycles are wasted idling (since any given IP address is rarely accessed). Second, even when serving a request, most of a honeypot's memory is idle as well (since few unsolicited requests demand significant memory resources from a host). Finally, different honeypot servers in a honeyfarm replicate the same environment and thus duplicate the effort in maintaining common state and executing common code paths. In fact, a conventional honeypot network will use far fewer than one percent of processor or memory resources for their intended purpose.

To avoid these inefficiencies, our architecture stresses the late binding of resources to requests. As network packets arrive, a specialized gateway router dynamically binds IP addresses to physical honeyfarm servers. For each active IP address, a physical server creates a lightweight virtual machine from a reference image (*flash cloning*), but only allocates new physical memory for these VMs as they diverge from the reference (*delta virtualization*). In its full implementation, we believe that this design can minimize processor and memory *idleness* and allow a single physical server to support the illusion of hundreds of network-attached hosts.

3.2 Containment

In steady-state, virtual machines are created and then quickly retired since most probes do not instantiate successful attacks. However, a new problem emerges when a virtual host *is* successfully compromised: it may attempt to attack or infect a third party. Indeed, a large-scale network honeyfarm could easily become a malware incubator or even an *accelerator* for a network worm. While we are unaware of a precise governing legal precedent, our purposeful negligence, foreknowledge of the risk, and inaction after detecting an intrusion creates the potential for significant third-party liability. Thus, making a large-scale honeyfarm practicable requires a defensible procedure and containment policy for controlling the actions of a compromised host.

However, the instantiation of this policy exposes an inherent tradeoff between a honeypot's fidelity and containment rigor. For example, the most extreme form of containment would disallow all outbound packets. Unfortunately, this policy would also prevent such simple interactions as the TCP handshake and thus blind a honeyfarm to the vast majority of attack vectors. A slightly less restrictive policy, used by the Honeywall system, might only forward outbound packets sent in response to inbound packets [12]. This policy too presents challenges; it does not allow benign third-party requests, such as for Domain Name System (DNS) translations. Moreover, modern worms, viruses and particularly botnets incorporate the ability to "phone home" to receive updates and commands. Without allowing these transactions it is impossible to understand the native behavior of a given piece of malware. Supporting these situations potentially requires a dynamic policy that can act on traffic characteristics exhibited by a particular honeypot.

Our architecture places the burden of implementing the containment policy on the gateway router. Thus, the gateway must track the communication patterns between external Internet addresses and the addresses being emulated by its honeyfarm servers. Moreover, it must be able to proxy or *scrub* standard outbound service requests such as DNS [19]. While these requirements add overhead, centralizing this complexity in one place dramatically simplifies management and policy specialization. For example, this design makes *internal reflection* simple to implement. When the containment policy determines that an outbound packet cannot be safely forwarded to the Internet, the gateway can *reflect* it back into the honeyfarm which will then adopt the identity of the destination IP address — effectively virtualizing the entire Internet. While this reflection must be carefully managed to avoid resource starvation, it can offer *signifi-*

cant insight into the spreading dynamics of new worms and supports high-quality detection algorithms based on causal linkage [9, 39].

However, reflection introduces an additional challenge as well. Two distinct compromised VMs may each send a packet to the same external IP address A — each packet itself subject to reflection. Thus, the gateway must decide if the honeyfarm creates a single VM to represent the address A or if these requests create two distinct VMs each assuming the same IP address A. The first approach allows cross-contamination between distinct contagions, and in turn this creates an opportunity for the honeyfarm to infect an outside host. For example, suppose external Internet hosts X and Y each infect the honeyfarm via addresses A_X and A_Y with unique worms, W_X and W_Y. In turn, the VM representing A_X may then infect A_Y with W_X and the VM representing A_Y may infect A_X with its infection in turn. Thus, each VM is infected with both worms. Since packets from A_X to Internet host X are forwarded without reflection, it is now possible for A_X to infect this host with worm W_Y, which again creates a new potential liability. Just because a host is infected by one virus does not give us the right to infect it with an unrelated virus. At the same time if the gateway creates unique aliases for each infection (such that no two worms may intermingle) this eliminates the possibility to capture symbiotic behavior between different pieces of malware. For example, the Nimda worm exploited a backdoor previously left by the CodeRed II worm. These problems have no ideal solution and thus it is important to be able to control how and when address aliasing is allowed to occur.

To support this level of control, we have introduced an additional address aliasing mechanism that captures the causal relationships of communication within the honeyfarm. Conceptually each packet is extended with a *universe identifier* that identifies a unique virtual IP address space. Universe identifiers are created when a new transaction is initiated from the external Internet. For example, when a new packet P arrives from the Internet destined for address X, it may create a new universe id U_{PX}. In turn, packets may only be reflected to hosts within the *same* universe from whence they came. Thus, a given infection can spread within the honeyfarm while being completely isolated from the influence of other infections. Capturing symbiotic, inter-worm behavior can be achieved by designating "mix-in" universes that allow instances of specific contagions to intermingle. Space limits prevent a full exploration of the richness and challenges presented by this abstraction, but we believe it encapsulates much of complexity introduced by reflection.

Our overall architecture, including an example of reflection, is roughly illustrated in Figure 1. In the remainder of this section we describe the gateway and virtual machine monitor functionality in more depth.

3.3 Gateway Router

The gateway is effectively the "brains" of the honeyfarm and is the only component that implements policy or maintains long-term state. More precisely, it supports four distinct functions: it must direct inbound traffic to individual honeyfarm servers, manage the containment of outbound traffic, implement long-term resource management across honeyfarm servers, and interface with detection, analysis and user-interface components. We consider each of these functions in turn.

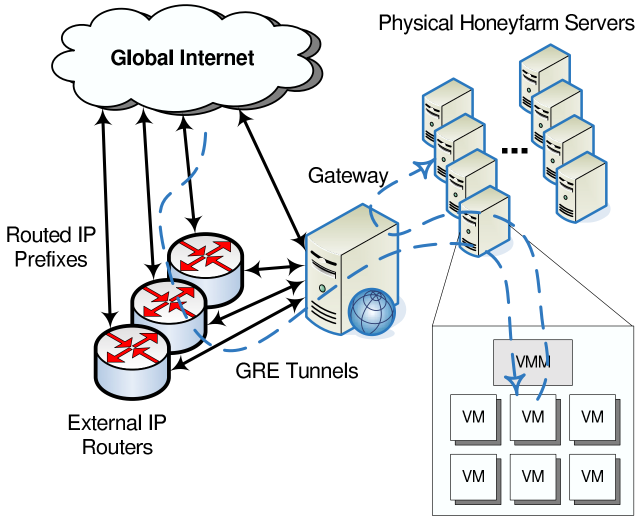


Figure 1: Honeyfarm architecture. Packets are relayed to the gateway from the global Internet via direct routes or tunnel encapsulation. The gateway dispatches packets to honeyfarm servers that, in turn, spawn a new virtual machine on demand to represent each active destination IP address. Subsequent outbound responses are subject to the gateway's containment policy, but even those packets restricted from the public Internet can be reflected back into the honeyfarm to elicit further behavior.

3.3.1 *Inbound Traffic*

The gateway attracts inbound traffic through two mechanisms: routing and tunneling. Individual IP prefixes may be globally advertised via the Border Gateway Protocol (BGP) and the gateway will act as the "last hop" router for packets destined to those addresses. This is the simplest way of focusing traffic into the honeyfarm, but has a number of practical drawbacks, since it requires the ability to make globally-visible BGP advertisements and also renders the location of the honeyfarm visible to anyone using tools such as `traceroute`. An alternative mechanism for attracting traffic is to configure external Internet routers to tunnel packets destined for a particular address range back to the gateway. This approach adds latency (and the added possibility of packet loss) but is invisible to traceroute and can be operationally and politically simpler for many organizations. These techniques have been used by several other honeyfarm designs, most recently in the Collapsar system [15].

Through this combination of mechanisms, a variety of packets destined for a variety of IP address prefixes arrive at the gateway. These must then be assigned to an appropriate backend honeyfarm server. Packets destined for inactive IP addresses — for which there is no active VM — are sent to a non-overloaded honeyfarm server. This assignment can be made randomly, can try to maximize the probability of infection (e.g., a packet destined for a NetBIOS service is unlikely to infect a VM hosting Linux since that service is not natively offered) or can be biased via a *type map* to preserve the illusion that a given IP address hosts a particular software configuration. Such mappings can be important if attackers "pre-scan" the honeyfarm for vulnerabilities and then exploit them at a significantly later date. Packets destined for active IP addresses are sent to the physical server implementing the associated VM. Thus, all of the packets in an extended flow — the set of contemporaneous transactions between a source and destination — will be directed to the same VM on the same physical honeyfarm server. To support this assignment the gateway must maintain state for each live VM in the system and must track the load and liveness of individual servers.

Moreover, the gateway must manage these assignments intelligently in response to changes in inbound traffic. For example, a common behavior that must be accommodated is network port scanning — where a single source contacts a large number of destinations in the same address range simply to see if the machine is live or if a given service is implemented. Instantiating a VM for each scan packet can inflate the demands on the honeyfarm unnecessarily. Instead, the gateway can filter packets from the same source to multiple destinations or arrange for scans to be proxied (either by the gateway itself or a shared VM) and then migrate to a dedicated VM if a more substantive transaction occurs. Similarly, during a worm outbreak a large number of effectively identical infections will cross the gateway. Since a new identical infection is unlikely to impart additional knowledge, the gateway may use pattern-matching algorithms to filter out such "known" attacks [16, 18, 29]. Bailey *et al.* describe a similar data reduction technique using low-fidelity responders to help prune traffic destined for high-fidelity virtual machines [2].

Finally, to simplify the operation of the gateway, the honeyfarm servers do not have innate IP addresses themselves. Instead, each server is addressed entirely using datalink-layer (i.e., Ethernet) addresses. This approach eliminates the need for the gateway to implement Network Address Translation (NAT) in the common case and allows the gateway to forward packets to individual servers unchanged.

3.3.2 Outbound Traffic

The gateway implements the only physical connection between the honeyfarm servers and the outside Internet so it can ensure that all traffic is subject to the same containment policy. To implement a basic response-only policy requires the gateway to track which source addresses have recently contacted each destination address. By contrast, a "fast-spread" containment policy might buffer outbound packets on a per-VM basis and block or reflect these packets if more than a small number of unique destination addresses are found. An additional practical complication is presented by DNS requests since a "normal" host is typically configured with the IP address of its local DNS server to which it directs name translation requests. Thus the gateway must either implement a DNS server itself (and forge the source address to appear to belong to the appropriate network) or allow outbound DNS requests to be proxied to a dedicated DNS server.

Finally, traffic that does not pass the containment filter may be reflected into the honeyfarm as described earlier. The gateway must moderate this reflection to prevent a worm epidemic from consuming all honeyfarm resources. Among the policies for managing honeyfarm resources are: partitioning the honeyfarm so only a subset of servers are dedicated to internal reflection; limiting the number of "hops" that a given communication may traverse; and, limiting the number of reflections with an identical or similar payload.

3.3.3 Resource Allocation and Detection

While it is clear when to create a new virtual machine, when to reclaim one is a more complex decision. Ideally, a VM would only persist long enough to determine that a particular attack was unsuccessful. Thus, optimizing resource usage requires the gateway to interpret input from network-based [33, 37] or host-based detectors [7, 25, 28]. The gateway can allow VMs that are known to be compromised to persist for further analysis, logging, or manipulation. VMs that remain in an uncompromised state can be reclaimed if they are no longer receiving inbound traffic. As well, when resources are low — due to network epidemics or explicit denial-of-service attacks — the gateway must prioritize which VMs should be reclaimed, which should be frozen to secondary storage, and which can continue to execute. However, balancing this need with the requirement for load balancing may be challenging. Servers running system images that have a disproportionate number of targeted vulnerabilities may, in turn, become disproportionately loaded (assuming compromised VMs are reclaimed with lower priority).

3.4 Virtual Machine Monitor

A principal role of the gateway is to remove idleness in the IP address space utilization. While the number of simultaneously active IP addresses is typically orders of magnitude smaller than the enclosing address range, it is still expensive to provision that many physical honeyfarm servers. However, since handling a typical request only requires a small subset of the server's hardware resources, a single server can potentially be multiplexed across a larger number of distinct IP addresses.[2] However, one cannot safely share a single operating system and application environment among requests to distinct IP addresses. Without isolation, one successful attack could pollute the behavior of all subsequent interactions. This is particularly problematic since modern botnets routinely patch security vulnerabilities to prevent competition from other attackers.

To provide isolation, our architecture uses a specialized virtual machine monitor to create a new virtual machine for *each* distinct IP address served. When a packet arrives for a new IP address, the VMM spawns a new VM. Once this new VM is ready, it adopts the packet's destination address and handles the request as though it were the intended recipient. Subsequent packets to the same IP address can then be delivered directly to that VM. Since each IP address is served by a distinct VM, any side effects from an attack will be isolated from other VMs. Finally, when instructed by the gateway the VMM can reclaim the resources of VMs that are no longer being monitored.

While this strategy provides isolation, it can also be quite expensive if implemented naively. In particular, a new VM can incur significant overhead initializing, booting an operating system, and loading application software. Not only does this overhead subtract from useful work, but if initialization takes too long any inbound connection request may time out. More critically, each VM may consume tens or

[2] In our measurements, handling a TCP session and serving a simple Web page consumes significantly less than 1ms of CPU time on a modern Pentium-based server system.

hundreds of megabytes of memory to represent its machine state.

For general computing purposes these costs are mostly unavoidable. However, given the restricted context of a network honeyfarm there are tremendous optimization opportunities. In particular, since the gateway manages address assignment, it can preserve the illusion of heterogeneity while physical honeyfarm servers only support homogeneous virtual machine images. Thus, by restricting a server to execute a *single* combination of operating system and application software, a new virtual machine may be created entirely via copying — a process we call *flash cloning*. To support this technique, each server maintains a reference image that provides a memory snapshot of a pre-initialized operating system and application environment. When a new VM needs to be created, this reference image is simply copied, its identity changed to reflect the appropriate network state (IP address, default gateway, DNS server, etc.), and then control is transferred directly. Using this approach, the overhead of creating a new VM can be reduced by well over an order of magnitude.

Exploiting the same homogeneity, the per-VM state requirement can also be significantly reduced. Since each VM is known to be a near-perfect clone of the same reference image, most of the state (code and data) in each VM is identical. Thus, instead of physically copying the reference image, a new VM can be equivalently instantiated via a copy-on-write optimization, which we term *delta virtualization*. Consequently, all memory pages which are unique to a given VM will be represented with their own dedicated storage, but most pages that are identical between VMs will be physically shared.[3] Similarly, when a VM is reclaimed this simply requires flushing the modified mappings and returning newly allocated pages to the free pool. Depending on the memory behavior of each system/application environment and the lifetime of each VM, this optimization can reduce per-VM state requirements by factor of 100. As well, it can reduce the VM startup overhead even further in exchange for slower execution as modified pages are lazily copied.

4. IMPLEMENTATION

We have implemented a limited version of this architecture in a prototype system we call *Potemkin*.[4] In the remainder of this section we discuss the implementation of the individual network gateway and VMM components that make up the system.

4.1 Gateway

The Potemkin gateway is built on top of the Click modular software router framework [17]. A router implemented in Click consists of a set of packet processing modules called *elements*, implemented as C++ classes. Using a special description language (a Click "configuration"), elements can be connected together to form a directed graph that represents how packets flow through the processing modules. Upon the base Click installation, our implementation adds roughly 5800 lines of custom element code, 450 lines for configuration and 1000 lines for an administrative interface. We briefly describe the operation, overheads and tradeoffs in our implementation below.

Each incoming packet is first stripped of any Generic Routing Encapsulation (GRE) headers [10] and then validated as being a credible honeyfarm address. Packets are subsequently matched against a series of programmable filters, which can be specified dynamically using a tcpdump-like syntax. The most important of these is an optional "scan filter" that limits the number of inbound packets from a given external source IP address using the same destination port and transport protocol. In our implementation, only one such packet may be delivered in any sixty-second window while the rest are dropped. This filter is designed to eliminate the overhead of creating thousands of short-lived VMs (i.e., one packet lifetime) when a single Internet host probes the same service across a range of monitored IP addresses (known commonly as a "horizontal port scan"). In future implementations, we plan to replace this filter with a responder proxy on each VMM that can defer VM creation until a complete session is established.

Unfiltered packets are matched against a flow cache to see if a VM has already been allocated for it. If so, the cache entry includes packet rewriting rules that install the destination MAC address of the physical server hosting the VM. In this case total forwarding overhead is approximately 8 μs. Failing a match in the flow cache, packets are checked against a history table that maintains long-term state about source/destination communication as well as access timestamps used to drive state reclamation decisions under load. Initial packets to a given IP address are matched against a host *type map* to find a physical machine compatible with the emulated type for that IP address. Between machines with compatible type, the gateway picks the least lightly loaded (based on regular reports from the VMM on each server). The resulting flow information, server address and timestamps are used to create new history and flow cache entries and the packet is dispatched to a physical server to create a new VM. In the worst case, with entirely random traffic, forwarding overhead is $\approx$40 μs.

On the outbound path, the operations are quite similar and differ primarily in their support for containment. The current implementation provides several containment policies that can be enabled independently: "history", which only forwards packets to external Internet addresses that have recent entries in the history table; "internal reflect", which reflects outbound packets that have been filtered back into the honeyfarm (triggering the creation of a new VM); and "protocol proxy", which allows individual protocol and service combinations to be forwarded to a proxy (in the current gateway this policy is used exclusively to allow a local DNS server to service DNS requests). Central to the implementation of these policies is the multi-universe concept described in Section 3.2. Thus, each history table entry tracks the universe identifier for a given packet. Packets from the Internet are assigned to "Universe-0" and new universe ids are created in the honeyfarm for each (src, dest, src port) combination. Internal reflections track the universe id via the history table so that an outbound packet is subject to the containment policy of the initial targeted IP address. Since reflection allows the possibility of externally visible

[3] This approach is similar in principle to the *content-based page sharing* in VMware's ESX Server, but our environment is far simpler since it is known that a new VM will exhibit perfect sharing relative to the reference image [34].

[4] The name originates from the legendary Potemkin Villages — elaborate facades allegedly created by Russian Field Marshal Grigor Potemkin to fool Empress Catherine II during her tour of conquered lands in the late 18th century.

address aliasing (e.g., two hosts in different universes, each purporting to be IP address A, sending packets to external IP address X) the flow cache rewriter rules support address translation for such connections initiated to the external Internet. In general, a reverse lookup on the flow cache is used for both forwarding outbound packets to the Internet and internal reflection. Best and worst case forwarding is similar to the inbound path, with the proviso that additional user-specified filters can add more overhead.

Finally, the gateway also provides a minimal administration interface used to register and unregister servers, monitor load, and communicate with intrusion detection components. In the current implementation this detection is provided solely via a reimplementation of Singh *et al.*'s *content sifting* worm signature inference algorithm that processes packets mirrored by the gateway [29].

4.2 Virtual Machine Monitor

The Potemkin VMM is based on the "xen-unstable" development branch, soon to be released as Xen 3.0, which we modified to support our virtual honeyfarm architecture. In its current form, it only supports *paravirtualized* hosts — requiring source modifications to the host OS — and our experience is solely with Linux.[5] We have added or modified roughly 2,000 lines of code within Xen and an additional 2,000 lines of code in control software, running within Domain-0, Xen's management VM.

In the following, we describe how the Potemkin VMM instantiates reference images and implements flash cloning and delta virtualization.

4.2.1 Reference Image Instantiation

To create the reference VM memory image in preparation for flash cloning, the VMM initializes a new VM, boots the guest operating system, and finally starts and warms the designated applications. A snapshot of this environment is then used to derive subsequent VM images. Note, our current implementation does not support disk devices reliably and thus we use uses memory-based filesystems exclusively. We expect to fix this restriction shortly and integrate with the Parallax storage subsystem that offers low-overhead disk snapshots [36].

4.2.2 Flash Cloning

Figure 2 illustrates the steps for cloning a VM. The principal entity involved in this process is the *clone manager*. When a honeyfarm server receives a data packet destined for a previously unseen address A, the VMM passes the packet to the clone manager, which in turn starts to initialize a new VM. During this time, it queues subsequent packets arriving for address A until the cloning process completes. The clone manager then instructs Xen to create a new VM as a copy of the reference image. Finally, the manager sends a special management packet to the newly instantiated VM to notify it of the new IP address A. When the cloned VM is resumed it receives this packet and instructs the guest operating system to set its IP address appropriately.

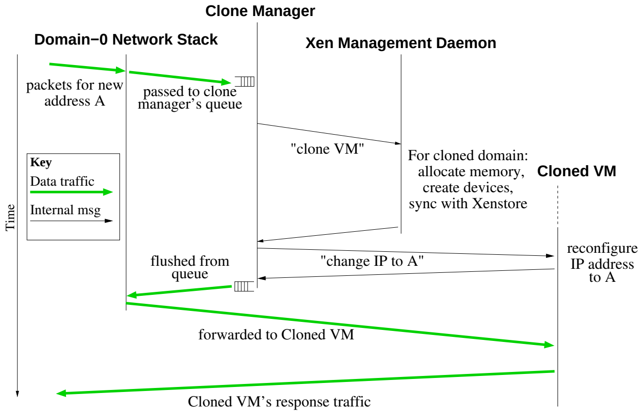

Clone Manager
Domain−0 Network Stack
Xen Management Daemon
packets for new address A
passed to clone manager's queue
"clone VM"
For cloned domain: allocate memory, create devices, sync with Xenstore
Cloned VM
Key
Data traffic
Internal msg
Time
flushed from queue
"change IP to A"
reconfigure IP address to A
forwarded to Cloned VM
Cloned VM's response traffic

Figure 2: Steps to clone a Linux VM.

After the cloning process completes, the clone manager flushes the packets destined for address A to the cloned VM and installs a packet filter entry that forwards subsequent packets destined for A directly to the VM for the remainder of the VM's lifetime.

Since the reference VM never responds to packets itself, the memory allocated to it is in a sense overhead; we get no useful computation from devoting this memory to the reference VM. However, the presence of the reference VM does permit a reduction in the memory requirements of cloned VMs, as described in the next section.

4.2.3 Delta Virtualization

To reduce state overhead, the Potemkin VMM implements a copy-on-write optimization, which we call *delta virtualization*, for flash cloned domains. Thus, the cloning operation simply maps all code and data pages from the reference image into the new domain. However, the clone's mappings are write-protected so subsequent modifications can then create private copies. Although particularly useful for optimizing flash cloning, our copy-on-write support is a general mechanism integrated with Xen. In the rest of this section, we describe the Xen translated shadow memory model, our extensions to the model to support copy-on-write functionality, and end with an example illustrating copy-on-write sharing.

Translated shadow memory. In the standard paravirtualized Xen memory model, Xen exposes the underlying machine addresses of each data page to the VM. Each VM directly maintains the set of page tables used by the processor, subject to safety checks imposed by the Xen VMM [3].

Xen additionally supports a shadow page table mode, in which the page tables maintained by each VM are not used directly by the processor. Instead, Xen creates shadow page tables on demand based upon the contents of the VM's page tables, and only the shadow page tables are used by hardware. This interposition gives Xen significant flexibility, since the shadow page tables need not be identical to the VM's page tables (though it does come at a performance cost). For example, shadow page tables are used to assist in tracking writes to memory pages during live VM migration [6].

Building on shadow page table support, Xen has a translated page table mode in which the page frame numbers stored in guest OS page tables are translated before they are written to the shadow page tables. Xen can then provide the illusion of a contiguous range of physical pages to the VM

[5] We expect to support Windows hosts in the near future using Xen's full virtualization support, in development and slated for inclusion in Xen 3.0. For full virtualization, Xen relies upon processors supporting Intel's Vanderpool Technology (VT) hardware feature set [14].

as a "guest physical address space", translating addresses to the scattered set of machine memory pages actually allocated when filling in the shadow page tables. Shadow page tables can be viewed as a cache of the final computed mapping between virtual addresses and machine page frames; like a cache, shadow page tables can be discarded at any time and recreated later.

Copy-on-write data structures. We implement delta virtualization in Potemkin as an extension to Xen's translated shadow mode. The indirection provided by translated shadow mode conveniently lends itself to the need to change the underlying machine address of a page after a copy-on-write fault.

For each VM, Xen stores the mapping between the VM's notion of physical page frame numbers to the actual machine page frames in the *physical-to-machine mapping table*.[6] Xen stores this data structure, one per VM, in the format of a hardware page table; this format permits the physical-to-machine mapping table to be directly used as a page table when fully virtualizing a VM that believes paging is disabled. On the x86 architecture, the page table format includes several bits in each page table entry (PTE) that are reserved for operating system use; in our delta virtualization implementation we claim these bits for our own use (although the use of these bits is never visible to the underlying TLB hardware nor the guest OS).

For each machine page in the system, Xen tracks several pieces of information in a data structure called the *frame table*: which VM owns the page (if any), a count of references to the page, a type (for enforcing safety properties), and a count of references pinning the page to its current type. Since at most one VM can own each page, pages shared copy-on-write require special treatment. We create a single "copy-on-write" VM in the system to act as a container for all shared pages; this VM never executes any code. Determining whether a particular page is shared between VMs is straightforward: check whether the owner is the copy-on-write VM. We do not track each individual domain that references a shared page since our implementation does not require this information. This "copy-on-write" domain represents an additional memory overhead relative to standard Xen; however it is a constant overhead irrespective of the number of VMs and pages, on the order of no more than tens of kilobytes.

The reference count on each page is used as before, totalling the number of references to this page, including PTEs in the shadow page tables. The type of a shared page is set to "writable", which is a default type for pages that do not have any other special type. We redefine the type count as the *sharing count*, or the number of VMs that reference the page. Even though the page type is marked as "writable", when constructing shadow page tables Xen will make all mappings to a shared page read-only.

Copy-on-write operations. The Potemkin copy-on-write implementation modifies three main operations: cloning VM memories, handling copy-on-write faults, and dealing with shared pages during VM termination.

To create a copy-on-write VM during flash cloning, Potemkin performs the following steps:

1. Discard the shadow page tables associated with the reference VM as a global invalidation step. Since shared pages require shadow PTEs to be read-only, the existing shadow PTEs will likely be invalid after flash cloning (recall that the shadow page tables are soft state that can be recreated on demand).

2. Iterate through the physical-to-machine table of the reference VM. For any page that is already shared, copy the address of the machine memory page to the physical-to-machine table of the clone. Increment the sharing count for the machine page in the frame table by one.

3. For any page that is not currently shared, but can be, change the owner of the page to *Shared* in the frame table, insert a reference to the page in the clone's physical-to-machine table, and set the sharing count to two in the frame table.

4. For any pages that cannot be shared, allocate a new page for the clone, copy the page contents, update the frame table for the new page, and leave the pages unshared.

To handle a copy-on-write fault, Potemkin performs the following steps:

1. Check whether the sharing count is currently one. If so, the page is not actually shared with another VM — simply transfer ownership to the faulting VM.

2. Otherwise, allocate a new page and copy the contents from the shared page. Update the physical-to-machine mapping table of the faulting VM to point to the newly created page.

3. Decrement the sharing count of the shared page in the frame table.

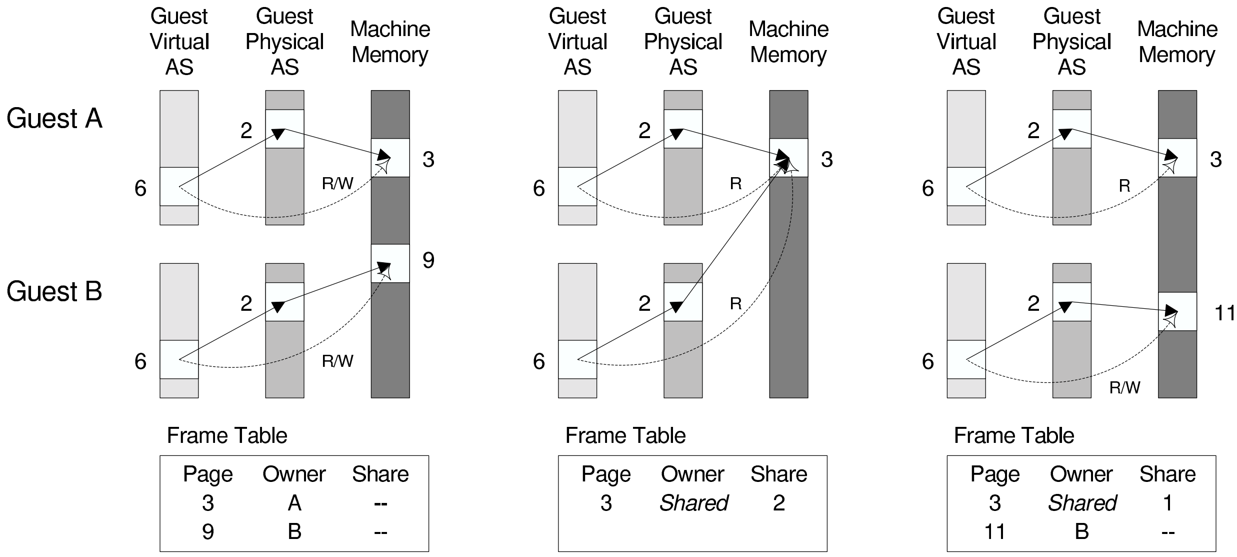
4. Invalidate all shadow PTEs in the faulting VM that point at the shared page since the machine address of that page has just changed.

We may break the sharing of pages in this manner at other times. When the type of a page must be changed, such as when a page is used as part of the x86's Global Descriptor Table, we break page sharing. We also do not currently permit a page to be shared when it is used as part of a page table and has associated shadow page table state. Finally, pages used for I/O may not be shared.

The last operation in a copy-on-write fault, invalidating shadow PTEs that point at the old page, is currently unoptimized because we do not track the location of all the PTEs pointing at a given memory page. Presently, we scan all shadow page table pages within the VM to perform this operation. In the future, we plan to optimize this step using another data structure. At a memory cost of approximately four bytes per shadow page table entry, we can track the page table entries to be invalidated and remove the need to scan all shadow page table entries.

In the current implementation of Potemkin, normal page faults are handled by Xen in 0.5–10 μs, depending upon the type of page fault and the work that must be done. Copy-on-write faults that only have to mark a page as unshared take an average of 10 μs, and faults that must make a full page copy an average of 60 μs.

[6] Xen uses the term "physical" to refer to addresses of pages within the illusory contiguous memory space presented to each VM. The term "machine" refers to the hardware address of pages after translation.



Page	Owner	Share
3	A	--
9	B	--

(a) No sharing

Page	Owner	Share
3	*Shared*	2

(b) Copy-on-write sharing

Page	Owner	Share
3	*Shared*	1
11	B	--

(c) After copy-on-write fault

Figure 3: Illustration of translated shadow memory management in Xen for two guest domains in three scenarios: (a) without sharing, (b) with copy-on-write sharing, and (c) after a copy-on-write fault. Boxes represent various memory abstractions managed by Xen, squares represent pages, and arrows represent mappings of pages from one memory domain to another as maintained by the Xen mapping data structures: guest OS page tables (guest VAS to PAS), physical-to-machine mapping tables (guest PAS to MM), and shadow page tables (guest VAS to MM). The frame table tracks the ownership and sharing status of the pages in machine memory.

Finally, at VM destruction, Xen ordinarily frees all the pages owned by the VM. In addition, the Potemkin VMM:

1. Walks the physical-to-machine mapping table, decrementing the sharing count in the frame table of all shared pages.
2. Frees any machine memory pages whose sharing count drops to zero.

Delta virtualization example. Figure 3 illustrates Potemkin's extension of Xen's translated shadow mode with copy-on-write sharing. It shows how Xen maps a virtual page in each of two guest domains, A and B, into machine memory for three scenarios. The first scenario depicts the use of translated shadow mode without copy-on-write. Xen maps virtual pages in a virtual address space (VAS) into a linear guest physical address space (PAS); for guest A, Figure 3(a) shows this mapping as a solid arrow from page 6 in the guest VAS to page 2 in the guest PAS. The page tables in the guest operating system maintain this mapping under the illusion that the guest OS is managing main memory directly. Xen then maps pages in the guest physical address space into machine memory (the arrow from page 2 in the PAS to page 3 in MM). The physical-to-machine table for each domain maintains this mapping. Recall that the shadow page tables embody the full mapping from a guest virtual page to a machine memory page so that the hardware can perform virtual address translation. The figure represents this full mapping as a dashed arrow from page 6 in the guest VAS to page 3 in MM. Finally, the frame table tracks the ownership of each machine page. In this case, guest A owns machine page 3 and guest B owns page 9. Since neither of the machine pages is shared, the protection bits in the PTEs from the shadow page tables are set "read/write".

Figure 3(b) shows copy-on-write sharing as if Potemkin flash-cloned guest B from A. In this example, Potemkin maps physical page 2 in the physical address spaces of both guests to page 3 of machine memory. Potemkin sets the protection bits in the PTEs in the shadow page tables for each guest domain to "read-only" to protect the shared page. In the frame table, Potemkin marks page 3 as *shared* with a sharing count of two.

Finally, Figure 3(c) shows the situation after a copy-on-write fault when guest B writes to its virtual page 6. In handling the fault, Potemkin allocates machine page 11, copies the contents of machine page 3 to 11, and updates the physical-to-machine mapping table for guest B so that B's physical page 2 now maps to machine page 11. Potemkin also updates B's shadow page table to reflect the new mapping, and sets the protection for the shadow PTE to "read/write" now that B has a private copy of the page. Potemkin updates the frame table as well, recording that B owns machine page 11. Note that machine page 3 remains shared, although Potemkin decrements its reference count from two to one.

5. EVALUATION

In this section we evaluate our architecture and the Potemkin prototype honeyfarm implementation. We focus on answering the following three key questions addressing the scalability of our approach:

- How many high-fidelity honeypot virtual machines are necessary for a honeyfarm to multiplex a given IP address space?

- How many physical resources are necessary for a honeyfarm to multiplex those honeypot virtual machines?

- How quickly does a centralized gateway become a bottleneck for honeyfarm scalability?

In summary, our results demonstrate that dynamic IP address multiplexing and scan filtering can reduce the number of honeypot servers required by three orders of magnitude. Further, we show that flash cloning and delta virtualization enable our design to support potentially hundreds of live VMs on each of these servers. Finally, we show that there is only moderate gateway overhead required to support fine-grained containment policies. Taken together, we argue that it is practical to deploy a honeyfarm that offers high-fidelity host emulation for hundreds of thousands of IP addresses while using only tens of physical servers.

5.1 Potemkin Testbed

Our experimental testbed consists of Dell 1750 and SC1425 servers, configured with 2.8GHz Xeon processors, 2GB of physical memory, and 1Gbps Ethernet NICs. The honeyfarm servers run the Potemkin VMM based on a pre-release version of Xen 3.0 as described in Section 4. We use Debian GNU/Linux 3.1 as the Xen guest operating system for constructing the honeypots. The gateway server is based on the Click 1.4.1 distribution running in kernel mode.

The honeyfarm servers and gateway host are directly connected through a shared Ethernet switch and the gateway is also the termination point of a GRE-tunneled /16 network prefix (64k IP addresses) nestled within an active, operational network. We use live traffic to this network as input to the honeyfarm.

5.2 Multiplexing Address Space

A key factor that determines the scalability of a honeyfarm is the number of honeypots required to handle the traffic from a particular IP address range. For example, a Potemkin honeyfarm serving all of a /8 network would naively require over 16 million honeypot VMs to cover the entire address range. However, in practice a honeyfarm can multiplex honeypot VMs across a given address space over time since not every IP address receives traffic simultaneously. As a result, a honeyfarm only needs sufficient VM resources to serve the peak number of *active* IP addresses at any particular point in time.

The number of honeypot VMs required depends on two properties: the address space locality of traffic within a limited time window and the lifetime of VMs handling traffic. If the traffic for a monitored address space has good locality, i.e., traffic is destined to only a small fraction of the address space in a limited time interval, then a honeyfarm requires fewer VMs to serve the traffic. However, if the traffic has poor spatial locality, then more VMs will be required. Finally, VM lifetime is also important since the longer a VM persists on average, the more additional VMs are required to service a given traffic load.

To evaluate address space multiplexing, we simulate a honeyfarm handling measured traffic to a /16 network. For this traffic workload, we estimate the number of active honeypot VMs required to process the traffic workload. We create an active VM when the honeyfarm receives a packet to a new destination IP address. This VM stays active so long as it receives traffic and terminates only if no new packets arrive for N seconds. Thus, over time, the number of active VMs in the honeyfarm reflects the IP address space utilization and determines the extent to which the honeyfarm can multiplex VM resources across the address space.

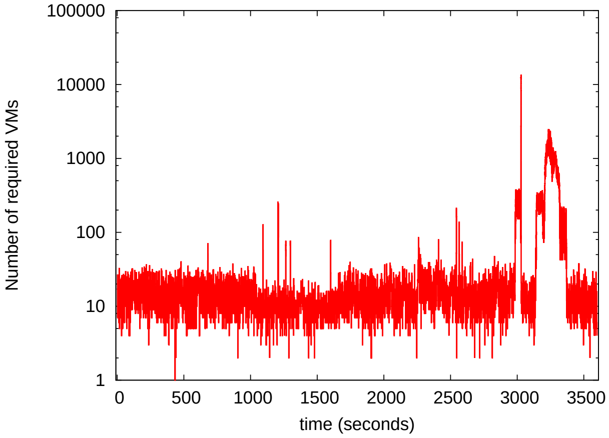


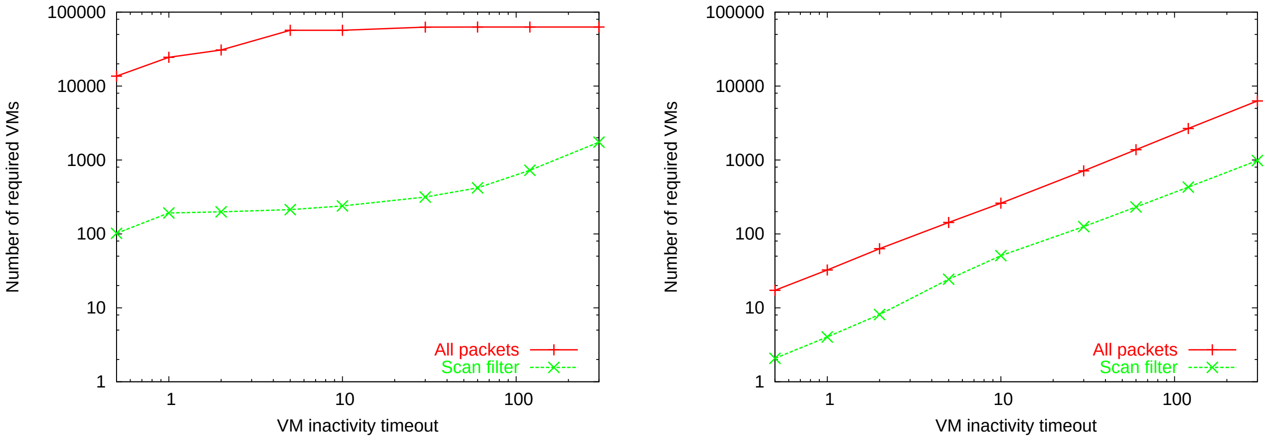
Figure 4: Required number of VMs active in response to all measured traffic from a /16 network when VMs are aggressively recycled after 500 milliseconds of inactivity. Traffic is from the one hour period starting Monday, March 21, 2005 04:05 GMT.

During a measured hour period, Figure 4 shows the number of active VMs required as a function of time when using an aggressive 500 millisecond inactivity timeout for VMs. While the average number of active VMs is 58, peak activity requires 13,614 VMs. Trace examination showed that the peaks were the result of wide-scale port scanning by a small number of source IP addresses — motivating the need for the "scan filtering" described previously (see Section 4.1).

Figure 5 shows the maximum and average number of simultaneously active VMs as a function of the VM inactivity timeout with and without scan filtering in place. The scan filter reduces the number of simultaneous VMs by over two orders of magnitude for VM timeouts up through 60 seconds. Even with five-minute timeouts, a honeyfarm would only require 1,745 VMs with the scan filter, whereas 62,960 (out of 65,536) would be required without filtering. Using a 60-second timeout, a honeyfarm can multiplex 156 destination addresses per active VM instance even during the worst-case period.

5.3 Multiplexing Honeyfarm Servers

The second key factor that determines the scalability of Potemkin is the number of VMs that each physical honeyfarm server can simultaneously multiplex. Three overheads determine the scalability of a honeyfarm server: the memory overhead required by honeypot VMs, the CPU overhead required to create honeypot VMs, and the CPU utilization of honeypot VMs responding to traffic. In the following series of experiments, we evaluate the effectiveness of delta virtualization and flash cloning in reducing the memory and time



(a) Maximum number of simultaneous VMs

(b) Average number of simultaneous VMs

Figure 5: Required number of active VMs as a function of the VM inactivity timeout, i.e., the amount of time a VM waits for additional network traffic since the last seen packet. Traffic is from a /16 network during the 48 hour period starting Sunday, March 21, 2005 at 08:05 GMT.

overheads, and quantify CPU utilization for a simple honeypot VM responder. Together, our results suggest that a honeyfarm server can multiplex hundreds of honeypot VMs on a single physical machine.

5.3.1 Delta Virtualization

Delta virtualization has the potential to significantly reduce the memory overhead of a cloned VM to just the set of pages the VM modifies after cloning. The number of dirty pages depends on which service executes in response to the first, or *trigger*, packet and how long the VM executes the service before terminating. This number limits overall scalability since a given honeyfarm server is only effective when it is able to maintain the working sets of cloned VMs entirely in physical memory.

To approach an upper bound on the number of VMs we can instantiate on a physical machine, we created as many clone VMs as possible within Xen resource constraints. The reference VM was a 128 MB Linux image, and each clone was a copy-on-write fork of the reference. Clones ran a process spinning in an infinite loop. We found that we could clone 116 VMs before exhausting the Xen heap, which is used to store important VM metadata but is presently limited to about 10 MB in size. In this experiment, Domain-0 consumed 512 MB of memory, the reference image consumed 128 MB, and the 116 clones together consumed 98 MB. On our 2 GB platform, 1310 MB of memory remained unallocated. Extrapolating past the current Xen heap limitation, such a machine could support approximately 1500 VMs before exhausting memory resources.

To estimate the memory scalability enabled by delta virtualization under application scenarios, we measured the memory overhead of three simple services running in a cloned VM. First, we configured Potemkin to clone VMs with the Apache Web server running, then requested a static web page from the VM. In separate experiments, we also connected via telnet into the machine to perform some simple file operations, and pinged the machine for approximately one minute. In each case, we tracked the number of pages not shared with the reference VM over time.

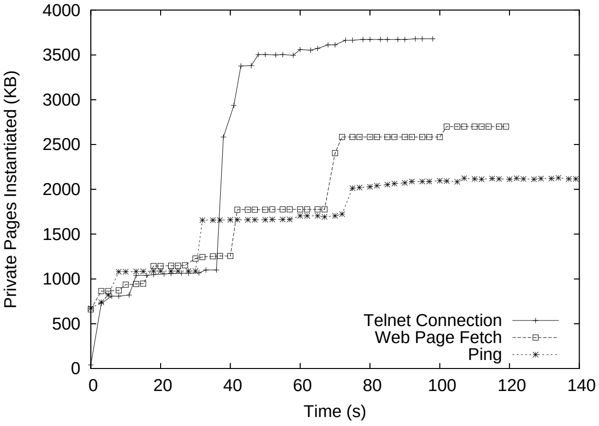


Figure 6: Memory modified by a cloned VM opening a telnet session, serving a Web page, and responding to pings. Activity for telnet begins at 40 sec, and activity for Web page fetch and ping begins at 70 sec. Until then, the VMs gradually modify pages while idling.

Figure 6 shows the memory modified by the cloned VM as a function of time. In the Web page experiment, the clone VM modifies 2.7 MB of memory when it finishes returning the requested page. Similarly, the telnet experiment modifies 3.7 MB and the ping experiment modifies 2.1 MB. Note that in the Web and ping experiments, the application activity occurs only after 70 seconds, while in in the telnet experiment, activity occurs roughly 40 seconds into the measurement. Until then, the VMs are idle and thus page modification during this period (1.1–1.8 MB) primarily reflects the activity of other OS processes and daemons, and not the application itself.

Flash Cloning		
Operation	Fork	Prealloc
Messaging/Python	124.5	124.7
Prepare/Copy memory pages	11.1	—
Sync with Xenstore DB	66.3	—
Create devices	149.2	—
Other management structures	11.4	—
Unpause	0.9	0.9
Raise interface	15.0	53.0
Configure IP	142.8	147.3
Total	521.2	325.9

VM Teardown		
Operation	Fork	Prealloc
Messaging/Python	116.0	116.6
Release devices	181.0	169.9
Release memory pages	7.0	7.9
Lower interface	10.6	10.6
Sync with Xenstore DB	0.9	0.7
Total	315.5	305.7

Table 1: Time breakdown for flash cloning with delta virtualization (Fork), and cloning preallocated domains (Prealloc). All times are in milliseconds.

Together, these results indicate that delta virtualization has the potential to provide excellent memory scalability. On a machine with 2 GB of physical memory, for example, a honeyfarm server can clearly accommodate the memory demands of hundreds of distinct VMs, even when executing simple services.

5.3.2 Flash Cloning

Next we evaluate the execution overhead to perform flash cloning and discuss our efforts to further improve its performance. Two driving factors motivate this concern. First, the time to flash clone a VM fundamentally determines the response time of the cloned VM. Until the cloned VM starts executing, it cannot respond to the trigger packet. As a result, if the time to clone a VM is very long (e.g., many seconds), the honeyfarm may miss activity of interest; a malicious host probing the honeyfarm, for example, may timeout and decide that no host exists at that destination address. Second, the time overhead of flash cloning impacts the number of VMs a physical machine can multiplex without degrading the performance of those VMs: the time to clone a VM is CPU overhead that could otherwise be used by already executing VMs. Since a Potemkin server will be cloning VMs continuously in steady state, CPU overhead required to clone VMs directly reduces the number of VMs that can efficiently execute on the machine. Reducing the time to clone a VM will therefore improve utilization of Potemkin servers and the number of VMs they can multiplex.

Table 1 shows the time to clone a 128 MB Linux VM using flash cloning with delta virtualization ("Fork"), and cloning preallocated domains ("Prealloc"). The total time measures the time from sending a ping packet to a Potemkin server until the time of receiving a response packet from a cloned VM on that server. For the degree of functionality instantiated, our current Potemkin implementation is able to create VMs relatively quickly: it flash clones a new VM, runs it, and generates a response packet in 521 ms. Subsequent packets to the same VM incur no such delay. Tearing down VMs is relatively costly at 315 ms, but we have yet to start optimizing this operation.

In steady state, a Potemkin server will clone and retire VMs continuously. To avoid repeating the same work, an obvious optimization is to recycle VM data structures rather than tear them down and reallocate them. While we do not yet perform this optimization, an upper bound on its benefits is represented by the "Prealloc" column in Table 1. In this scenario, domain data structures are preallocated and preinitialized, thus reflecting the potential benefits of data structure reuse. While this optimization reduces cloning overhead to 326ms, a substantial improvement, significant unnecessary overheads still remain.

Examining flash cloning overhead in more detail, we break down the flash cloning times into various subcomponents according to major steps shown in Figure 2. The total time in Table 1 corresponds to the time to perform the entire sequence in the figure. The time breakdowns of "Prepare/Copy memory pages" through "Unpause" in the table comprise the "clone VM" step in the figure. "Raise interface" and "Configure IP" comprise the "change IP to A" step. "Messaging/Python" comprises the remaining steps in the timeline, covering communication overhead among different entities on the server. From these breakdowns, we observe that the time overhead is distributed among a number of different operations. We are addressing all of them in further optimization, starting with recycling since it eliminates needless work. Device creation during flash cloning and release during teardown are obvious candidates for recycling, as are other management and memory data structures as well as configurations communicated through the Xenstore DB. We are reducing messaging and control overheads through general improvements to the Xen configuration tools (particularly eliminating the use of Python) as well as optimizations specifically in support of flash cloning. Finally, we are investigating techniques to better prime reference images for configuring their network identity when cloned. We believe that true flash cloning overheads should ultimately be well under 30ms.

5.3.3 CPU Utilization

Delta virtualization reduces the memory overhead of VM clones by orders of magnitude, and flash cloning correspondingly reduces the time overhead of creating honeypot VMs. The last factor that determines the number of VMs a honeyfarm server can successfully multiplex is the CPU utilization of responding honeypot VMs. Putting aside the overhead to create new VMs, if the combined utilization of *existing* VMs exceeds the honeyfarm server CPU capacity, then fidelity will suffer as all honeypots grind to a crawl.

Fortunately, due to the nature of typical honeypot traffic (scans, probes, etc.), most honeypot VMs consume minimal overhead. As a baseline, we measured the CPU utilization of a honeypot VM responding to an HTTP request for the top-level page of a default Apache server, in addition to idle CPU usage of the VM during periods of Web server I/O. The utilization was below 0.01%, indicating that memory resources and cloning overhead will remain the primary resource bottleneck for multiplexing honeypot VMs on honeyfarm servers. We note that these results represent a common case when incoming packets perform harmless operations and the server can retire the VM quickly. If a VM becomes infected, however, it will consume additional resources.

5.4 Gateway

The gateway is an important component of our honeyfarm architecture, since it globally manages honeyfarm resources in response to both external inbound traffic and internal traffic. As a result, the scalability of the honeyfarm ultimately depends on the gateway's behavior. In turn, the scalability of the gateway depends upon the rate at which it can forward traffic and the amount of state required to implement its forwarding and containment policies.

To evaluate these limits, we transmitted packets to the gateway at a given rate and measured the rate at which the gateway was able to deliver the packets as output. In our current implementation, the delivered forwarding rate depends heavily on the locality of the packet stream. For packets hitting in the flow cache, the gateway is able to deliver over 160,000 packets per second. However, random traffic can reduce this rate by a factor of six (28,000 packets per second). While this is still easily sufficient to support the peak delivery rates from Section 5.2, additional optimization may be required to support still-larger deployments.

The gateway also maintains tables to track active mappings of flows to honeypot VMs, and the history of recent sources to destinations communications for implementing containment policies. Entries in these tables require roughly 256 bytes per flow or 256 MB to support a million simultaneous flows. As a result, the gateway will easily scale to very large address ranges as a single central node.

5.5 Live Deployment

We have very preliminary experience combining all of the components together into a live honeyfarm deployment. Using a cluster of ten servers, we configured one as the Potemkin gateway and the remaining nine as Potemkin VMM servers. We directed the traffic from our tunneled /16 network to the gateway, which then dispersed the traffic among the servers. For a representative 10-minute period, over 2,100 VMs were dynamically created in response to external packets and responded in kind. Anecdotally, looking at the packets exchanged in more detail, we observed the expected common behavior. TCP SYN packets to live services established TCP connections, while those addressed to closed ports generated RSTs in response. We received background Slammer infection attempts (engendering an ICMP port unreachable response from our VMs since they are not configured to provide service on UDP port 1434). In fairness, we are far from production use in our current deployment and we continue to debug various crashes, overloads, and hangs. However, our initial experience is promising and reinforces our position that it is possible to create honeyfarms with both scale and fidelity.

6. LIMITATIONS AND CHALLENGES

Computer security is a field fraught with "almosts" and "gotchas". Few technologies can address all threats completely or precisely. Indeed, honeypots are no exception to this rule, nor does our approach avoid introducing its own unique challenges. In this section we briefly discuss some limitations of our work and how they might be addressed in future work by our group or others.

6.1 Attracting Attacks

Chief among these limitations is the assumption that a honeypot will attract all traffic of interest. A typical honeypot's only salient connection to the rest of the Internet is its IP address and, thus, it will tend to only receive traffic from randomly targeted attacks. In fact, many non-random attack vectors, such as peer-to-peer, instant messenger, and e-mail, spread along application-specific topologies. To *attract* attacks on these systems requires additional mechanisms to connect the honeypot into these application-specific networks. In some cases, such as for peer-to-peer applications, the application itself provides a connection; many common applications, however, do not (e.g., e-mail). Indeed, to capture e-mail viruses, a honeypot must possess an e-mail address, must be scripted to read mail (potentially executing attachments like a naive user) and, most critically, *real e-mail users* must be influenced to add the honeypot to their address books. Passive malware (such as many spyware applications) may require a honeypot to generate explicit requests and narrowly focused malware (e.g., targeting only financial institutions) may carefully section their victims and never touch a large-scale honeyfarm. In each of these cases there are at least partial solutions (e.g., Wang *et al.*'s HoneyMonkey system incorporates a Web crawler to gather candidate spyware applications [35]) but they require careful engineering to truly mimic the vulnerable characteristics of the target environment.

6.2 Honeypot Detection

A related problem is that attackers may attempt to explicitly detect that they are running in a honeypot environment and modify their behavior to evade detection or analysis. Indeed, this is not a hypothetical threat: several modern bots (e.g., the Agobot strains) actively detect and react to VMware-based execution environments. Camouflaging a honeypot against these threats presents several challenges. First, the local execution environment must be completely virtualized to prevent detection. While this is impossible today in the Intel x86 architecture, the next generation of Intel and AMD processors offer extensions that make complete functional virtualization possible. However, even with this processor-level support, effectively camouflaging the overall platform characteristics (e.g., presenting a believable and consistent description of local devices) can be quite difficult. Moreover, even if a honeypot's execution environment is perfectly virtualized, an attacker may still be able to infer its presence via external side-effects. For example, virtualization typically incurs overhead and, while local time can be dilated to compensate, external time will still advance on schedule. Thus an attacker could detect a virtualized honeypot by executing a long series of instructions with high virtualization overhead and comparing the elapsed time to some external Internet reference (e.g., via the network time protocol). As well, an attacker may exploit any communication barriers imposed by the containment process to infer the presence of a honeypot. For instance, if containment prevents outbound infection attempts to a third-party host, an attacker may issue such an attempt to detect the presence of a likely honeypot. Finally, if a honeypot uses a static range of IP addresses, then long-term testing — of the variety described here — could be used to populate a blacklist avoided by future malware.

Again, each of these challenges has corresponding engineering approaches. For example, honeyfarm blacklists can be thwarted by using dynamic address assignment (e.g., temporarily redirecting the unused addresses from a DHCP server) rather than building honeyfarms from static swaths of contiguous prefixes. Similarly, many network-based "fingerprints" can be hidden by emulating ambiguities imposed by common network elements, such as firewalls, network address translators, shared Wi-Fi, variable delay cable modems, etc. Each of these situations, however, requires a unique and ad hoc solution. Ultimately, it remains unclear if it is possible to perfectly hide a honeyfarm from a skilled and determined attacker.

6.3 Denial-of-Service

Finally, the scalability of Potemkin is predicated on the assumption that the address space is sparsely utilized over short time scales and that VMs themselves need only live for short periods of time. However, an attacker with some knowledge of the honeyfarm's location could violate these assumptions and overwhelm the system. For example, by blanketing a honeyfarm's address space with seemingly distinct attacks, an attacker may exceed the capacity of the physical infrastructure. Worse, successful attacks could then attempt to dirty large numbers of pages to minimize the value of copy-on-write sharing. Finally, if attackers are is privy to the detection algorithms and policy used to decide how long to maintain a given VM, they can artificially extend the VM lifetime and therefore exhaust shared processor and memory resources. As with all denial-of-service vulnerabilities, these require careful resource management to mitigate. However, resource management requires the ability to "name" different users — in this case different attacks. Thus, important capabilities are identifying identical or isomorphic attacks and limiting their resource consumption.

As with all security technologies, we expect that the importance of these additional threats will depend largely on the value of honeypots themselves. If honeypots are a significant deterrent, then attackers will be motivated to evade or overwhelm them — once again mirroring the classic arms race between espionage and counter-espionage.

7. CONCLUSION

Traditionally, honeypots have been used to detect and observe the behavior of worms, viruses, and botnets. Current deployments have either been limited to running high-fidelity honeypots over a small number of IP addresses, or low-fidelity honeypots over a large number of IP addresses. In this paper, we propose a honeyfarm architecture that achieves both goals: it supports high-fidelity honeypots to capture attacker behavior at scales that are orders of magnitude greater than previous high-fidelity honeypot approaches. We have built a prototype honeyfarm system, called *Potemkin*, that exploits virtual machines, late binding of resources, and aggressive memory sharing to achieve this goal. Late binding of virtual machines to IP addresses enables Potemkin to monitor the steady-state attack traffic of an IP address space several orders of magnitude larger than the available number of physical honeyfarm servers. Aggressive memory sharing and efficient VM creation enable our design to support potentially hundreds of live VMs on each of these servers. We evaluate our architecture and the Potemkin prototype honeyfarm implementation, and argue that it is practical to deploy a honeyfarm that offers high-fidelity host emulation for hundreds of thousands of IP addresses while using only tens of physical servers.

8. ACKNOWLEDGMENTS

This work and this paper would not have been possible without significant efforts of others. First we would like to thank the Xen team in its entirety for the Xen platform itself — without which we would never have considered this line of research. We are particularly grateful to Michael Fetterman for his insights concerning the integration of copy-on-write into shadow-mode page tables, Andrew Warfield for his skunkworks efforts to help us integrate Parallax, and Ian Pratt for his overall support and guidance. Steve Hand was our shepherd and was extremely patient with our "last-minute" research efforts. George Dunlap and Dominic Lucchetti of the University of Michigan helped us to get Xen running in translated shadow mode. Back at UCSD Marvin McNett kept the honeyfarm running in spite of our move across campus, and Colleen Shannon kept the packets flowing. Michelle Panik kept us organized and grammatical. Our ICSI counterparts in CCIED, particularly Vern Paxson, Nick Weaver and Weidong Cui, provided frequent useful feedback on our work. We thank the anonymous reviewers for their comments, and Bill Cheswick for his unique contribution. Finally, we would like to specially acknowledge our co-author Alex Snoeren who, in the week before his wedding, proved to our students that faculty members can still hack.

Support for this work was provided by the National Science Foundation under CyberTrust Grant No. CNS-0433668 and Trusted Computing Grant No. CCR-0311690, a gift from Microsoft Research, and the UCSD Center for Networked Systems.

9. REFERENCES

[1] M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson. The Internet Motion Sensor: A Distributed Blackhole Monitoring System. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, San Diego, CA, Feb. 2005.

[2] M. Bailey, E. Cooke, F. Jahanian, N. Provos, K. Rosaen, and D. Watson. Data Reduction for the Scalable Automated Analysis of Distributed Darknet Traffic. In *Proceedings of the USENIX/ACM Internet Measurement Conference*, New Orleans, LA, Oct. 2005.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the 19th ACM Symposium on Operating System Principles (SOSP)*, Bolton Landing, NY, Oct. 2003.

[4] P. Chen, A. Joshi, S. King, and G. Dunlap. Detecting Past and Present Intrusions through Vulnerability-specific Predicates. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP)*, Brighton, UK, Oct. 2005.

[5] B. Cheswick. An Evening with Berferd In Which a Cracker is Lured, Endured, and Studied. In *Proceedings of the Winter Usenix Conference*, San Francisco, CA, 1992.

[6] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[7] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating System Principles (SOSP)*, Brighton, UK, Oct. 2005.

[8] D. Dagon, X. Qin, G. Gu, W. Lee, J. Grizzard, J. Levine, and H. Owen. HoneyStat: Local Worm Detection Using

Honeypots. In *In Recent Advances In Intrusion Detection (RAID) 2004*, Sept. 2004.

[9] D. Ellis, J. Aiken, K. Attwood, and S. Tenaglia. A Behavioral Approach to Worm Detection. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, Fairfax, VA, Oct. 2004.

[10] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina. RFC 2784 - Generic Routing Encapsulation (GRE). RFC 2784, Mar. 2000.

[11] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of the 10th Annual Network and Distributed System Security Symposium (NDSS '03)*, San Diego, CA, Feb. 2003.

[12] Honeynet Project. *Know Your Enemy: Learning about Security Threats*. Pearson Education, Inc., Boston, MA, second edition, 2004.

[13] Honeynet Project. Know Your Enemy: Tracking Botnets. http://www.honeynet.org/papers/bots/, Mar. 2005.

[14] Intel. Virtualization Technology. http://www.intel.com/technology/computing/vptech/.

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

[16] H.-A. Kim and B. Karp. Autograph: Toward Automated, Distributed Worm Signature Detection. In *Proceedings of the USENIX Security Symposium*, San Diego, CA, Aug. 2004.

[17] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM Transactions on Computer Systems*, 18(3):263–297, Aug. 2000.

[18] C. Kreibich and J. Crowcroft. Honeycomb — Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the 2nd ACM Workshop on Hot Topics in Networks (HotNets-II)*, Cambridge, MA, Nov. 2003.

[19] G. R. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and Application Protocol Scrubbing. In *Proceedings of IEEE Infocom Conference*, pages 1381–1390, Tel-Aviv, Isreal, Mar. 2000.

[20] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the Slammer Worm. *IEEE Security and Privacy*, 1(4):33–39, July 2003.

[21] D. Moore, C. Shannon, and J. Brown. Code-Red: a case study on the spread and victims of an Internet worm. In *Proceedings of the ACM/USENIX Internet Measurement Workshop (IMW)*, Marseille, France, Nov. 2002.

[22] D. Moore, C. Shannon, G. Voelker, and S. Savage. Network telescopes: Technical report. Technical Report CS2004-0795, UCSD, July 2004.

[23] D. Moore, G. M. Voelker, and S. Savage. Inferring Internet Denial of Service Activity. In *Proceedings of the USENIX Security Symposium*, Washington, D.C., Aug. 2001.

[24] C. Nachenberg. From AntiVirus to AntiWorm: A New Strategy for A New Threat Landscape. Invited talk at 2004 ACM Worm, http://www.icir.org/vern/worm04/carey.ppt.

[25] J. Newsome and D. Song. Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, San Diego, CA, Feb. 2005.

[26] R. Pang, V. Yegneswaran, P. Barford, V. Paxson, and L. Peterson. Characteristics of Internet Background Radiation. In *Proceedings of the USENIX/ACM Internet Measurement Conference*, Taormina, Sicily, Italy, Oct. 2004.

[27] N. Provos. A Virtual Honeypot Framework. In *Proceedings of the USENIX Security Symposium*, San Diego, CA, Aug. 2004.

[28] J. Rabek, R. Khazan, S. Lewandowski, and R. Cunningham. Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, Washington, D.C., Oct. 2003.

[29] S. Singh, C. Estan, G. Varghese, and S. Savage. Automated Worm Fingerprinting. In *Proceedings of the 6th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[30] D. Song, R. Malan, and R. Stone. A Snapshot of Global Internet Worm Activity. Technical report, Arbor Networks Technical Report, Nov. 2001.

[31] C. Stoll. *The Cuckoo's Egg*. Pocket Books, New York, NY, 1990.

[32] Symantec. Decoy Server Product Sheet. http://www.symantec.com/.

[33] S. Venkataraman, D. Song, P. Gibbons, and A. Blum. New Streaming Algorithms for Superspreader Detection. In *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS '05)*, San Diego, CA, Feb. 2005.

[34] C. A. Waldspurger. Memory Resource Management in VMware ESX Server. In *Proceedings of the 5th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[35] Y.-M. Wang, D. Beck, X. Jiang, and R. Roussev. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites that Exploit Browser Vulnerabilities. Technical Report MSR-TR-2005-72, Microsoft Research, Aug. 2005.

[36] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing Storage for a Million Machines. In *Proceedings of the 10th USENIX Workshop on Hot Topics in Operating Systems (HotOS-X)*, Santa Fe, NM, June 2005.

[37] N. Weaver, S. Staniford, and V. Paxson. Very Fast Containment of Scanning Worms. In *Proceedings of the USENIX Security Symposium*, San Diego, CA, Aug. 2004.

[38] A. Whitaker, M. Shaw, and S. D. Gribble. Scale and Performance in the Denali Isolation Kernel. In *Proceedings of the 5th ACM/USENIX Symposium on Operating System Design and Implementation (OSDI)*, Boston, MA, Dec. 2002.

[39] J. Xiong. ACT: Attachment Chain Tracing Scheme for Email Virus Detection and Control. In *Proceedings of the ACM Workshop on Rapid Malcode (WORM)*, Fairfax, VA, Oct. 2004.

[40] V. Yegneswaran, P. Barford, and D. Plonka. On the Design and Use of Internet Sinks for Network Abuse Monitoring. In *Proceedings of Symposium on Recent Advances in Intrusion Detection (RAID)*, Sept. 2004.