Openfire: Opening networks to reduce network attacks on legitimate services

Laura Fink
University of Michigan
7 PUBLICATIONS **142** CITATIONS

OpenFire: Opening Networks to Reduce Network Attacks on Legitimate Services

Kevin Borders, Laura Falk, and Atul Prakash
Department of EECS, University of Michigan, Ann Arbor, MI 48109
{kborders,laura,aprakash}@eecs.umich.edu

Abstract

Remote network attacks are a serious problem facing network administrators in universities, corporations, and governments today. Hackers, worms, and spammers will exploit and control hosts inside of a network to send junk E-mail, create bot nets, launch DDoS attacks, and steal sensitive information. Current solutions exist that attempt to deal with these threats, but they are often unsuccessful. In addition, security researchers have deployed honeypots to lure attackers and learn more about their techniques, but honeypots have had only limited success at gaining information and protecting other machines. In response to need for better security against network attacks, we present OpenFire. OpenFire is protection system that takes the opposite approach of traditional firewalls: instead of blocking unwanted traffic, it instead accepts *all* traffic, forwarding unwanted messages to a cluster of decoy machines. To the outside, all ports and all IP addresses appear open in an OpenFire network. To evaluate OpenFire, we deployed it in a live sub-network here at the University of Michigan. During a three-week evaluation period, we found that OpenFire reduced the number of attacks on legitimate computers by 57%, while honeypots only reduced attacks by 8%. OpenFire decoys also had considerably better exposure than honeypots, and were able to elicit over three times as many attacks during the evaluation period.

1 Introduction

Remote network attacks are a major problem on the internet today. These network attacks come from a variety of adversaries such as script kiddies, worms, and even skilled hackers. Regardless of the perpetrator, remote attacks usually consist of a reconnaissance phase followed by exploitation. Firewalls are usually the first line of defense to mitigate network scans and attacks [1, 9, 10]. Firewalls do a very good job of blocking incoming traffic to services that should not be exposed to the outside network. They also block outgoing replies to remote probes. However, firewalls do not protect services from attack that are exposed to the external network.

In addition to firewalls, many system administrators deploy intrusion detection systems (IDSs) [23, 24] or intrusion prevention systems (IPSs) [2, 8, 11] to help detect and prevent scans and attacks on networked computers. These systems can be very effective at stopping simple scans and known attacks. Unfortunately, IDSs and IPSs have serious shortcomings that prevent them from stopping a skilled hacker. If run slowly, over a long enough period of time, and from many source IP addresses, port scans are virtually undetectable by an IDS or IPS. Furthermore, once an attacker has identified a vulnerable target, most IDSs have a very hard time detecting zero-day and polymorphic attacks, which are becoming increasingly prevalent [14, 27].

Honeypots are another tool that can help system administrators learn when their network is under attack [15, 22, 25, 26]. They also provide the added benefit of exposing the mechanisms used by the adversary to assail the network. Originally, security researchers deployed honeypots on individual computers with a single IP address. This configuration was able to entice a few hackers to attack honeypot machines, but only with limited success. More recently, honeypots have been set up to receive traffic destined to large chunks of empty IP space in order to provide more targets for malicious hackers. Filling a large portion of the IP address space also makes it a little bit more difficult to identify normal targets. As we observed during experiments, however, such a configuration does not significantly reduce attacks on normal machines. Any security benefit seen by the normal machines in this configuration comes from firewall or IDS/IPS reconfiguration following detection of an attack on the honeypot machines.

To help deal with the threat of remote network attacks, we created OpenFire. OpenFire is an alternative to current honeypot solutions that makes network scanning and targeting extremely difficult. OpenFire takes the opposite approach of traditional firewalls; instead of blocking traffic to all unused ports and IP addresses, it allows *all* incoming traffic, but forwards traffic that a firewall would normally block to a cluster of *decoy* machines. Note here that we refer to these machines as decoys instead of honeypots because their primary purpose is to protect other machines, not just to lure attackers.

OpenFire decoys receive traffic to all closed ports and unused IP addresses, even if legitimate services are running on open ports on the same IP address. This means that SMTP (port 25) traffic to a web server (port 80) will not go to the web server, but instead to a decoy SMTP server. In fact, OpenFire may route TCP connections to a single IP address to a variety of decoy machines, all running different services and potentially different operating systems. From the outside, every IP address in an OpenFire network will have every single port open and each open port may lead to one of any number of decoy machines. Our hypothesis is that this will make results from port scans and OS fingerprinting much less useful to an attacker. Without effective reconnaissance, attacks on an OpenFire network are more likely to target decoy machines rather than normal machines. OpenFire greatly increases the exposure of decoy machines over traditional honeypot configurations while at the same time protecting normal hosts from malicious adversaries.

We evaluated OpenFire by testing it against a number of popular scanning tools, deploying it in a live sub-network here at the University, and evaluating its resilience against attacks. We found that tools such as nmap [18] and Nessus [17]were unable to produce very meaningful results when scanning target machines for open services. They returned a long list of open ports for each host, most of which mapped to decoys. Attempts to fingerprint host operating systems also yielded incorrect results for many of the machines in our test network.

Deploying OpenFire in a live network along with traditional honeypot configurations yielded a number of interesting results. We found that attacks on normal computers, as measured by the Snort intrusion detection system [24], decreased in volume by 57%. OpenFire decoys were also three times more effective at eliciting attacks than honeypots occupying the same IP space. During the course of the evaluation, OpenFire decoys saw an average of 89 attacks per day, while the honeypots only saw 27 attacks per day. This suggests that OpenFire decoys have much better exposure than traditional honeypots.

For the final part of our evaluation, we carefully examined potential shortcomings and attacks against OpenFire as well as their countermeasures. One limitation of OpenFire is that it is designed to prevent transport-layer reconnaissance. It is not designed to not protect against application-layer and out-of-band information gathering. If an HTML link exists to a company's www server, for example, then OpenFire cannot protect a hacker from fingerprinting and attacking the server. We also considered the possibility of a denial-of-service attack distributed evenly across the entire OpenFire network's address space aimed at bringing down decoy machines, as well as methods of mitigating this attack. Next, we looked at probe latency characteristics to make sure that timing attacks could not be used to differentiate between decoy and normal machines. Finally, we discuss the use of passive monitoring to identify real services and propose a method for obfuscating such attacks.

The remainder of the paper is laid out as follows: Section 2 discusses related work. Section 3 presents the details of OpenFire's design. Section 4 describes OpenFire's implementation. Section 5 gives the results of our evaluation. Section 6 provides an in-depth analysis of attacks on OpenFire. Section 7 gives an overview of possible future work, and section 8 concludes.

2 Related Work

Much work has been done in the field of intrusion detection and prevention with the goal of stopping remote network attacks. In general, IDS's can be divided into two categories: host intrusion detection systems (HIDS) and network intrusion detection systems (NIDS). Most HIDS focus on monitoring local activity such as file system access [13] or system call patterns [5] rather than remote scans. For this reason, the security benefits of HIDS are largely independent of those gained by using OpenFire. Deploying these two systems in conjunction with one another will provide a greater level of security.

In contrast to HIDS, NIDS attempt to counteract remote scans and attacks by examining packets on the network. First, some NIDS can modify or drop packets that have non-standard TCP or IP flags set in order to prevent OS fingerprinting. This is commonly referred to as protocol scrubbing [29]. This method, however, does not prevent an attacker from being able to see which ports are open. NIDS also utilize a number of signatures to detect incoming traffic containing known attack payloads and port scans [19, 23, 24]. As mentioned earlier, known attack signatures are not good for identifying zero-day or polymorphic attacks, and scan signatures are easy to evade by doing a full TCP connect scan over a long period of time while changing your source IP address. Due to these shortcomings, deploying OpenFire alongside traditional NIDS provides a much higher level of security. OpenFire does a better job of countering port scans because it makes the results almost completely useless. Furthermore, OpenFire still protects machines against zero-day and polymorphic attacks by exposing decoys on a large portion of the address space.

In addition to blocking traffic based on scan and attack signatures, some intrusion prevention systems (IPSs) [2, 8, 11] make an effort to obfuscate the results of "stealth" SYN scans. An attacker can send out TCP SYN packets to various destinations, waiting for a SYN/ACK packet or a reset packet to indicate whether or not the port is open. In response, some IPS's automatically respond to all SYN packets with a SYN/ACK, regardless of their destination, and do not complete the actual connection on the internal network until they receive an ACK packet from the remote host. This way, a SYN scan would return that all destination ports are open. (Incidentally, this mechanism also helps prevent SYN flood attacks.) Performing a full TCP connect scan, however, is easy to do and will completely bypass this filter.

Another popular method of securing services from remote attack is port knocking [16]. Port knocking works by modifying the TCP stack on a server. From the outside, all ports will appear to be closed on the server. But, if packets are sent in order to a predetermined sequence of destination ports, a port will open up on the server for the client who "knocked" to connect. Port knocking does prevent remote transport-layer scans because all services appear closed to the outside. It also prevents remote exploitation, unless the attacker is able to discover the knocking sequence through an out-of-band channel. However, in order for a client to connect to a network using port knocking, the client needs to install special software and obtain the correct knocking sequence. Teaching potential clients to use port knocking can be costly and inconvenient. In contrast, OpenFire is a

Service	Host A	Host B	Host C	Honeypot A	Honeypot B	Honeypot C	Total (Honeypot)
FTP (21)	Open	Open	X	Open	Open	X	4 (2)
SSH (22)	Open	X	Open	Open	X	Open	4 (2)
SMTP (25)	X	X	Open	X	X	Open	2 (1)
HTTP (80)	X	Open	X	X	Open	X	2 (1)
SSL (443)	X	Open	X	X	Open	X	2 (1)

(a)

Service	Host A	Host B	Host C	Decoy A	Decoy B	Decoy C	Total(Decoy)
FTP (21)	Open	Open	Decoy A	Open	Open	Decoy C	6 (4)
SSH (22)	Open	Decoy C	Open	Open	Decoy A	Open	6 (4)
SMTP (25)	Decoy C	Decoy C	Open	Decoy C	Decoy C	Open	6 (5)
HTTP (80)	Decoy B	Open	Decoy B	Decoy B	Open	Decoy B	6 (5)
SSL (443)	Decoy B	Open	Decoy B	Decoy B	Open	Decoy B	6 (5)

(b)

Table 1: (a) List of some open ports for a network with three different machine types and honeypots on half of the address space. (b) List of some open ports for same network with OpenFire running. Here decoys cover approximately three times as much address space for the ports listed above.

complete server-side solution that does not require clients to run special software or know how it works. Furthermore, by accepting dead-end traffic OpenFire can leverage information about the methods and capabilities of potential attackers.

In the realm of honeypot research, there have been a number of projects of increasing scale and sophistication. The original Honeynet project consisted of a few computers in a production network that were set up to lure attackers and gain information about their techniques [25]. Each physical computer mapped to a single IP address, just as the other machines in the network. Although this setup can provide an early warning that an attack is underway, the honeypots typically occupied only a small portion of the IP space, and thus would only detect wide-scale attacks. Due to their sparse deployment, these honeypots did not significantly hinder scanning operations on the network as a whole.

More recently, low-interaction honeypots have been set up to cover much larger address blocks [21]. Low-interaction means that the honeypots will accept incoming packets or connections, but are not running real services that can interact with an attacker. Such honeypots cannot elicit attacks that involve more than one message and are easy to identify, but require far less computing resources than high-interaction honeypots (honeypots that run real services and operating systems). A new system, the Potemkin Virtual Honeyfarm [28], gets around many of the performance limitations of high-interaction honeypots so that they can cover large address blocks just like low-interaction honeypots.

State-of-the-art honeypot networks are much better at detecting focused attacks that only target a few machines because they cover a much larger portion of the address space. Furthermore, they can make scanning difficult because every IP address within a subnet can appear to be live, while many of them map to honeypot machines. Even when configured to take up the entire empty address space, however, these honeypots still do not significantly reduce the number of attacks on other hosts, as we will see later in the evaluation section. OpenFire, on the other hand, does reduce attacks on normal machines by 57%. It also increases malicious traffic to decoys by over three times, providing a better early warning against potential attacks on a network.

3 Design

In this section, we present the detailed design of the OpenFire system. First, we analyze address space coverage and attack probability. Next, we take a look at individual decoy configuration possibilities and how they affect OpenFire. Finally, we address failure and fault tolerance issues, highlighting design decisions made to ensure usability and eliminate vulnerabilities in OpenFire.

3.1 OpenFire Address Space Coverage

The main design principle of OpenFire is that to the outside network, all ports and IP addresses should appear to be running legitimate services. This way, it is extremely difficult for attackers to identify actual services using remote scans. Probabilistically, OpenFire aims to minimize the chance of a random scan hitting an actual service. As a corollary, OpenFire greatly increases the likelihood of random attacks hitting decoy services.

An example of OpenFire's exposure versus that a network with conventional honeypots on unused IP addresses can be seen in Table 1. On this network, there are three servers labeled Host A, B, and C. Each of these hosts is running a different set of services. For each host, there is a corresponding decoy with the same configuration. In Table 1a half of the addresses on the network map to honeypot machines. Because each machine only runs a few services, much of the address space (16 of 30 destinations) is still closed. Also notice that only 50% of the destinations map to honeypots.

In the OpenFire configuration seen in Table 1b, all 30 of the {IP, port} destinations are open. Instead of covering 50% of the open address space with 7 targets, decoys now cover 77% with 23 targets. This number is a lower bound and does not include other ports, which are open but map to a dummy service (described in more detail in section 3.2). If an attacker attacks a randomly selected network service 3 times, there will be a 45% chance of only hitting decoys and a 1.27% chance of avoiding decoys entirely, compared to a 12.5% of only hitting honeypots and a 12.5% chance of missing them entirely. Keep in mind that these numbers are hypothetical, but give a good idea of the extra protection provided by OpenFire. On a real network, there are likely to be more addresses (a /24 subnet contains 256 addresses), only a few servers of each type, and a greater diversity of services. In this example, 23% of the address space mapped to real services. In a production network, this number may be 10% or less, making the benefits of OpenFire over honeypots even more pronounced.

3.2 OpenFire Decoy Configuration

In order for the OpenFire decoys to be effective, it is essential that they run the same services as other hosts on the network. If an attacker discovers a service that is running on the normal machines but not on the decoys, then it would be possible for him or her to map out the legitimate servers and avoid the decoys. Also, if one wants OpenFire to provide the same level of protection for all services on a network, then it is important to configure decoys in the same proportion as the legitimate servers. For example, if half of the web servers on the network are running Apache and half are running Microsoft IIS, then having the same ratio of decoys will ensure equal protection for Apache and IIS servers. If none of the decoys had Apache installed, then an attacker who targets Apache will hit a real server every time. However, an unbalanced configuration may be beneficial if one server is believed to be more secure than the other and in less need of protection.

For destination ports that are not associated with any known service, OpenFire routes traffic to a "dummy" service. The purpose of the dummy service is to accept connections and data sent to it. This way, all ports will appear open on a transport-layer scan. The dummy service could be a simple script that receives data indefinitely without replying or it could be a real server that normally runs at a different port. Bear in mind that an attacker can still use an application-layer probe to tell what type of service is running on a particular port and potentially identify dummy services. OpenFire cannot prevent this type of fingerprinting.

When setting up the decoys, one tradeoff to consider is whether or not to install services that are not currently running on the network. For example, if you only have Apache servers on your network, would it be a good idea to install IIS servers on some decoys? On one hand, installing IIS on some decoys will decrease the level of protection for the Apache servers. It may also cause a false alarm if an attacker exploits an IIS decoy. Time and effort must be spent diagnosing the break-in even if normal hosts on the network are not vulnerable. Running additional decoy services, however, does have benefits. Hackers may select a particular service on a machine, try to exploit it, and then give up if they fail. Also, even though a dummy service is running that accepts connections on all ports, many attackers will check the target application version before launching an exploit. If they see a dummy service, they may move on to another target. Having a greater variety of real services running means more targets for attackers, and less attacks on real machines. For our implementation, we chose not to have services running on decoys that were not running on the other machines to avoid false positives, but it may be advantageous in some situations to run extra decoy services.

3.3 Fault Tolerance

Another critical aspect of OpenFire's design is resilience against attacks and system failures. If OpenFire were to use a static mapping of destination {IP, port} combinations and one of the normal hosts went down, then an attacker would be able to map its open services by probing for closed ports. Then, if the machine came back up, the information could be used to attack it without targeting any decoys. If a decoy fails, an attacker could also use this information to see which ports mapped to that decoy and avoid them in the future. (One can differentiate decoys from other machines in most configurations because one decoy usually accepts traffic to many IP addresses, while other machines only accept traffic to one address.)

To address the issue of closed ports resulting from system crashes and reboots, OpenFire needs a strategy for dynamically rerouting traffic when a destination becomes unavailable. One option would be to redirect each failed connection to a randomly chosen decoy. However, if subsequent probes elicit different behavior very often, then an attacker can assume that they are going to different machines and the destination must be a decoy.

Instead of using random redirection on a per-connection basis, OpenFire maintains a static mapping from external {IP, port} destinations to actual {IP, port} destinations that it updates dynamically when connections fail or hosts (including decoys) stop responding. OpenFire reconfigures the mapping and redirects connections on-the-fly when they fail. When a host stops responding entirely, OpenFire reassigns all of its destinations. Aside from timing issues, which we will discuss later in the evaluation section, this redirection is completely transparent to the external network.

Although it works well in most situations, problems arise with dynamic assignment when a major public server goes down. If an organization's main www server or main SMTP crashes, for example, then it may be undesirable for OpenFire to reroute connections to a decoy machine. First of all, sometimes the benefits of knowing when a service has gone down outweigh the desire to not disclose whether it is a decoy to the outside. For a large public server, attackers are likely to know that it is real through out-of-band channels such as website links. Also, users may become confused or flustered and overwhelm administrators with support calls. Finally, redirecting such a large amount of traffic to one decoy may overload it, potentially causing it to crash or start running very slowly and causing even more problems. To address this issue, OpenFire supports a white list of destinations that it will not redirect to decoys, regardless of whether the real services fail. An alternative method for dealing with failed services, instead of diverting traffic to decoys, is to have the decoy services fail randomly as well. This way, when a particular service stops responding, an attacker will not be able to tell whether or not it is a decoy. Also, when a host stops responding entirely, OpenFire should stop forwarding all traffic to its IP address. This ensures that an attacker cannot map the open ports on a machine when it crashes or reboots. Ideally, decoy services should fail with the same frequency as real services. If they do not fail with the same frequency, then an attacker could measure failure frequency to determine if a service is a decoy. Doing this, however, would be difficult because it would require the attacker to know the failure profile for real services.

4 Implementation

When implementing the OpenFire system, we considered a few possible configurations. OpenFire needs to be able to redirect packets on the fly from closed ports on the legitimate machines to services running on the decoys. This can be done either at the network gateway, or the end host. If OpenFire operates on the end host, it has the added benefit of providing protection against attackers *inside* the network. Unfortunately, this configuration is more prone to error and requires extra effort from all hosts in the network. If any of the computers have configuration errors or do not have OpenFire installed, then they will not be protected. This is especially problematic with mobile hosts coming and going from the network, because they are often beyond the domain of administrative control. For these reasons, and because we are concerned about external attacks, we chose to implement OpenFire at the network gateway.

OpenFire runs at the edge of a network, with an interface to the internal and external networks. It redirects connections according to a mapping from external {IP, port} pairs to internal {IP, port} destinations, much like a network address translation (NAT) device. Whenever OpenFire sees a TCP reset packet indicating a closed port it will temporarily change the destination to an appropriate decoy service. If a host stops responding entirely, OpenFire will change the destination for all of the entries with that host's IP to a decoy service. In order to gracefully handle system reboots, OpenFire will probe computers that stop responding every five minutes, and reset the address mapping when they become available again. If a computer becomes permanently unavailable, then it is the system administrator's responsibility to manually update OpenFire's configuration.

In order to increase the effectiveness of OpenFire, it is helpful to deploy it in as sparse of an address space as possible. As discussed in section 3.1, the more unused addresses that are available to OpenFire the more protection it can provide. Here, sparseness is not just measured by the number of unused IP addresses, but the total size of the unused {IP, port} space for each service. For example, consider a /24 subnet (256 addresses) with every IP address taken by a legitimate machine, but with only 5 web servers. This address space is still sparse because OpenFire is able to fill over 98% of it with decoys.

4.1 Decoy Configuration

Decoy configuration is a very important part of OpenFire's implementation. We installed the decoys on virtual machines for easier management and to minimize resource utilization. We set up the decoy virtual machines to mimic actual machines on the network, one decoy per machine type. In general, it is best to configure most decoys to be very similar to legitimate machines so that it is very difficult an attacker to tell them apart. As mentioned earlier, we did not configure decoys for services that were not originally running on the network.

After setting up the decoy machines, the next step in deploying OpenFire is to create the port forwarding configuration. The forwarding configuration specifies the mapping of external {IP, port} addresses to internal {IP, port} addresses. For each service running in the OpenFire network, we checked every IP address to see if a server was running. If not, then we created a forwarding entry to redirect all traffic sent to that address to a particular decoy. We selected the type of decoy in proportion to the number of similar legitimate machines on the network (i.e. if 20% of legitimate machines that are running an SSH server have version 1 and 80% had version 2, then SSH traffic to 20% of unused IPs would go to an SSH 1 decoy and 80% to an SSH 2 decoy).

For ports that do not have a service commonly associated with them, or ports whose typical service was not running on the network, we configured OpenFire to send all of the traffic to dummy service running on one of the decoys. Instead of choosing a popular service such as HTTP or SSH for the dummy server, we used a simple server that accepted all connections, received data indefinitely, but never sent anything back. The reason we chose a simple dummy server instead of a more sophisticated one (which may lure more attacks) is to make denial-of-service attacks on OpenFire more difficult. An attack against the dummy service that is evenly distributed across all addresses and ports may be able to bring down a complex server that uses a lot of resources to handle each client. A simple server, however, is harder to attack because it does no processing on incoming data whatsoever and keeps no per-connection state except for the TCP stack. If more computing resources are available for the decoys, or denial-of-service attacks are not an issue, then it may be more beneficial to use HTTP, SSH, and other common servers for the dummy service.

For our implementation of OpenFire, we largely disregarded UDP traffic and services. None of the machines in our test network ran services that respond to unsolicited UDP packets. The OpenFire gateway was set to permit all incoming and outgoing UDP packets in order to allow voice-over-IP and other real-time media applications, but block ICMP destination unreachable messages to prevent transport-layer UDP port scans. For networks that do run UDP servers, such as DNS, NTP, Microsoft SQL monitor, OpenFire can be configured to forward UDP traffic to decoys and dummy servers the exact same way that it forwards TCP traffic. Because UDP is a connectionless protocol, however, OpenFire would need to do special handling for multiple packets from the same source IP address sent to different destinations that get mapped to the same decoy. OpenFire must assign the correct source IP address to the decoy's UDP replies.

4.2 Management and Scalability

When deploying OpenFire in a medium to large-sized network, management and system performance start to become major concerns. In order to work properly, OpenFire need an up-to-date list of all the allocated IP addresses and open ports on the network. If it does not have this data, then it may route legitimate traffic to decoys or allow probes to closed ports. Maintaining a configuration file with all this information would be very time-consuming if done by hand. However, OpenFire could use a system similar to Dark Orcale [3] that monitors routing tables, ARP entries, and other data to keep an up-to-date list of unused IPs. In addition, OpenFire could run port scans on the occupied address space to build its decoy routing configuration. We plan to investigate integration of OpenFire with a system like Dark Oracle in the future to facilitate automated management for large address spaces.

Another issue that may arise when setting up OpenFire in a large network is performance. Because our current design operates at the network gateway, it must be able to handle all of the traffic coming in and out of the network. For a medium or large network, this could be a more traffic than a typical software firewall can manage. Fortunately, OpenFire can run independently on smaller sub-networks instead of at the edge of the entire network. This is the best way of deploying OpenFire in larger networks. Each sub-network will have its own OpenFire gateway and decoy machines that can handle all of the sub-network's traffic. This configuration solves the performance scalability problem while at the same time providing extra security for traffic flowing between sub-networks in an enterprise environment.

5 Evaluation

5.1 Experimental Setup

To evaluate OpenFire, we deployed it here at the University. We used it to protect three available workstations. These workstations included one computer running Windows XP Service Pack 2 and two Linux computers running Fedora Core 3. All three machines were secured with host-based firewalls and virus protection. The workstations had the following ports open:

- Windows XP SP 2: 3389
- Fedora Core 3 Machine 1: 21, 22
- Fedora Core 3 Machine 2: 21, 22, 111, 139, 445,1024.

The OpenFire gateway was set up in between these three hosts and the internet. A diagram of our test network layout can be seen in Figure 1. The gateway ran on dual core x86 processor with 4 gigabytes of RAM and two 100 megabit Ethernet cards. On the same computer as the OpenFire gateway, we ran three decoy virtual machines, each with 1 GB of RAM, to mimic the three workstations. Each decoy was configured with the same services as its corresponding workstation. In addition to the three original IP addresses of the workstations, we gave OpenFire 36 unused IP addresses to map to the decoys. Using the method outlined in section 4.1, we created a configuration file to redirect all dead-end traffic for the 39 IPs to the decoy machines.

In the remainder of this section, we describe the results of our evaluation in two phases. During the first phase of the experiment, we ran popular scanning and fingerprinting tools on the OpenFire network and analyzed the results. In the second phase, we let OpenFire run for three weeks along with an intrusion detection system to test its effectiveness in a live environment. We also compared these results with the original configuration and a standard honeypot configuration.

5.2 Fingerprinting Programs

For the first phase of our evaluation, we ran the popular scanning tools nmap [18] and Nessus [17] on machines inside of the OpenFire network. Using a full TCP connection scan, nmap showed that all ports were open on all destination IP addresses inside of the network. It was unable to differentiate between the dummy services and the real services. Next, we ran Nessus, which produced a similar result. Nessus, however, also performed some application-level probes for particular services and reported that many of the ports "might be protected by some TCP wrapper" because the dummy service did not send back any data. This result suggests that it may be a good idea to run a large number of decoy services rather than dummy services. The decoy services will interact with clients at the application level, while the dummy services will not send back any data and can be identified as bogus by application-layer scanning tools like Nessus. Transport layer scanning tools like nmap, however, are completely ineffective against OpenFire.

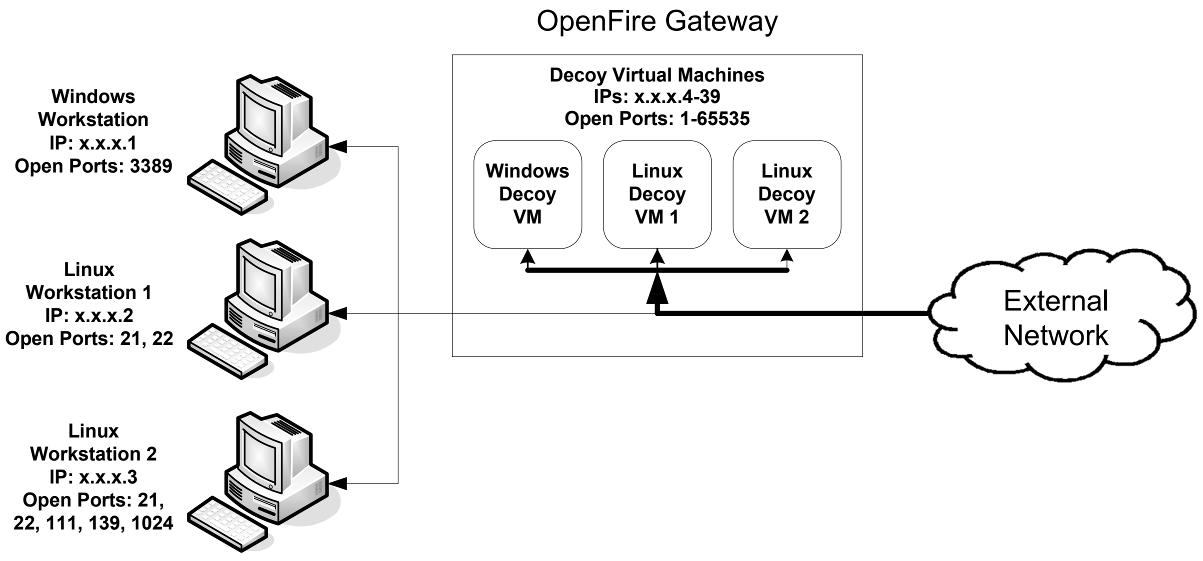

Figure 1: Openfire's Network Layout

After running port scans, we attempted to fingerprint the Operating Systems of the three workstations, as well as some decoys. For each of the three workstations, nmap said that they were running the "Linux 2.4.X" operating system. This was correct for two of the workstations, but the third was actually running Windows XP SP 2. When we tried to fingerprint the decoys, nmap returned either "Linux 2.4.X" or "No exact OS matches for host." The reason for the inconsistency is that the same ports on different decoys mapped to machines with different operating systems. When nmap does OS fingerprinting, it sends probes to different ports to identify unique characteristics of the target host's TCP implementation [6]. If all the probes end up going to a Linux 2.4.X host, then nmap can identify the OS correctly. If some of the probes go to a Windows machine, then nmap is unable to determine the OS.

These results seem to indicate that if a large portion of machines are running a common operating system, then nmap is likely to guess that OS when fingerprinting the OpenFire network. This means that nmap is unable to reliably determine the operating system of any particular host. If modified to only probe one port at a time, however, it would be possible for nmap to fingerprint the target operating system on a per-port basis. If one wants to completely eliminate the possibility of OS fingerprinting, then OpenFire can be deployed in conjunction with a protocol scrubber [29], which was mentioned earlier.

5.3 Deployment in a Live Network

In the next step of our evaluation, we deployed OpenFire in a live network and collected attack statistics using the Snort intrusion detection system [24]. We also gathered attack data from a normal configuration and from a honeypot configuration for the purposes of comparison. Figure 2 shows the normal and honeypot network layouts. In the normal setup, three workstations ran Snort to record the number of attacks, and the 36 other IP addresses allocated for our experiment went unused. The three workstations went unchanged for the honeypot configuration, but we assigned the 36 extra IP addresses to three honeypot machines.

We ran each of the three configurations, normal, honeypot, and OpenFire, for a period of three

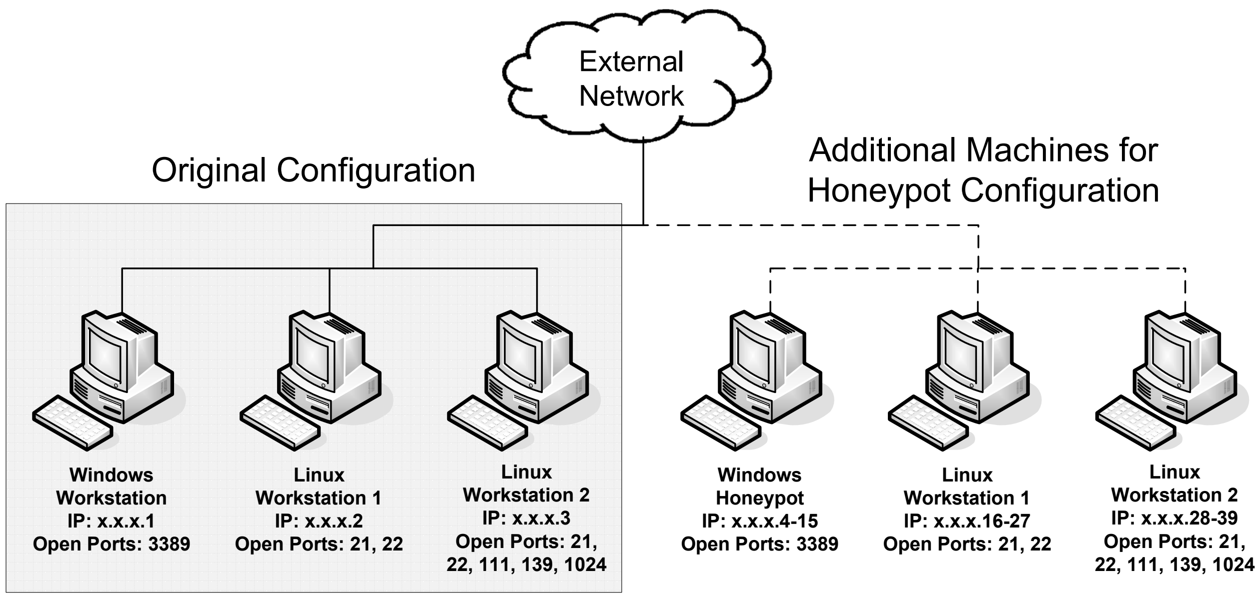
External Network
Original Configuration
Additional Machines for Honeypot Configuration
Windows Workstation IP: x.x.x.1 Open Ports: 3389
Linux Workstation 1 IP: x.x.x.2 Open Ports: 21, 22
Linux Workstation 2 IP: x.x.x.3 Open Ports: 21, 22, 111, 139, 1024
Windows Honeypot IP: x.x.x.4-15 Open Ports: 3389
Linux Workstation 1 IP: x.x.x.16-27 Open Ports: 21, 22
Linux Workstation 2 IP: x.x.x.28-39 Open Ports: 21, 22, 111, 139, 1024

Figure 2: Normal and Honeypot Network Layouts

Type of Attack	**Percentage of Total Attacks**
FTP Bounce Attack	31.7%
Web XML RPC Exploit	26.1%
SSH Brute Force Login Attempt	12.3%
x86 Shellcode Payload	7.2%

Table 2: The top four alerts generated by attacks on the OpenFire network during the three week training period.

weeks and observed the number of attacks on the network. After removing false positives and alerts from scans, we were left with remote exploits and brute force login attempts. The top four alerts from the OpenFire network are shown in Table 2.

Figure 3 shows the results from our evaluation of the normal, honeypot, and OpenFire configurations. The average number of alerts per day for the normal configuration was 9.7 compared to 8.9 for the honeypot setup and 4.2 for OpenFire. These results seem to indicate that if attackers have fewer targets, then they will attack those targets more frequently. Also, having honeypots in a sub-network on all of the empty IP addresses may slightly reduce the number of attacks on the real machines. Here, honeypots decreased attacks on other machines by 8%. This difference is small, however, and could have been caused partially by random variation. The protection provided by having honeypot machines on empty IP addresses is minimal.

In the OpenFire configuration, the number of attacks per day on real machines went down 57% from the original setup. This shows that OpenFire helps protect real computers by giving adversaries a large number of bogus targets. The most likely cause for this trend is that an attacker will choose a particular service for an IP address. If the attacker focuses on exploiting the chosen service and moves on to the next IP if it is not vulnerable, then OpenFire will do a good job of protecting the real machines. On the other hand, if an attacker chooses an IP address and

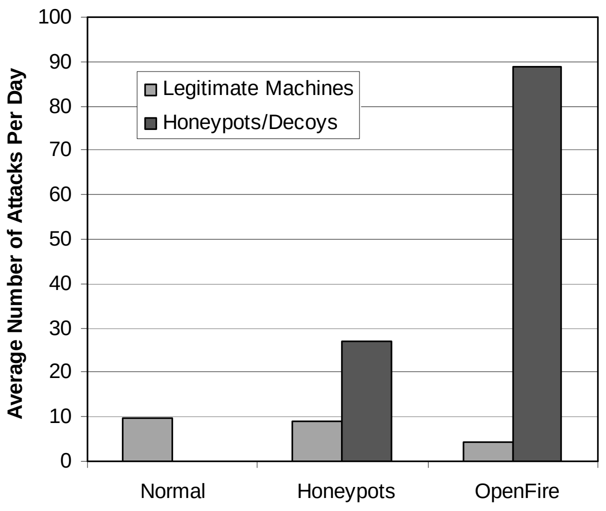

Figure 3: Average number of attacks per day over a three-week evaluation period for the normal, honeypot, and OpenFire configurations. The attacks are separated into attacks on legitimate machines and attack on honeyport or decoy machines

proceeds to attack all available services on until it finds one that is vulnerable, then OpenFire will not be able to directly protect the computers as well. In the second case, however, an administrator is likely to see a very large volume of the attacks going to the decoy machines, and will have an early warning that an attack is underway.

In comparison to honeypots occupying the same IP space, OpenFire was very successful at drawing attacks. On average, honeypots occupying 36 IP addresses saw 27 attacks per day, while the OpenFire decoys saw 89. This result is not surprising, especially considering that many of the attacks probably came from script kiddies, worms, or other automated programs. A knowledgeable human who manually analyzes the results of a port scan and sees that every single port is open may catch on and decide not to attack the system. A worm or auto-rooter, however, will have a field day and just start attacking whatever it can. With normal honeypots, having closed ports for certain services means that each honeypot is seeing fewer attacks. OpenFire takes full advantage of its address space to lure as many attacks as possible

One potential issue that may arise from having such a large number of attacks on a network is bandwidth usage. During the evaluation period, we also logged all of the packets going in and out of the decoy network to measure their bandwidth usage. The results from this measurement can be seen in Figure 4. This figure shows the bandwidth usage for the decoys averaged over 5-minute intervals. For most the majority of the three-week period, the bandwidth usage stayed below 1 kilobyte/second. On November 15th, there was a spike in traffic where the peak bandwidth reached 63 KB/s, or 29 KB/s average over a 5-minute period. The aggregate bandwidth used during the entire three weeks summed up to 293 MB. Even though OpenFire lured many more attacks than honeypot systems, the amount of extra bandwidth from attack traffic is still very reasonable for a network with 39 IP addresses.

6 Attacks on OpenFire

In this section we take a look at potential attacks on OpenFire and their countermeasures. The first threat we will consider is a denial-of-service attack on the decoy machines. Next, we evaluate

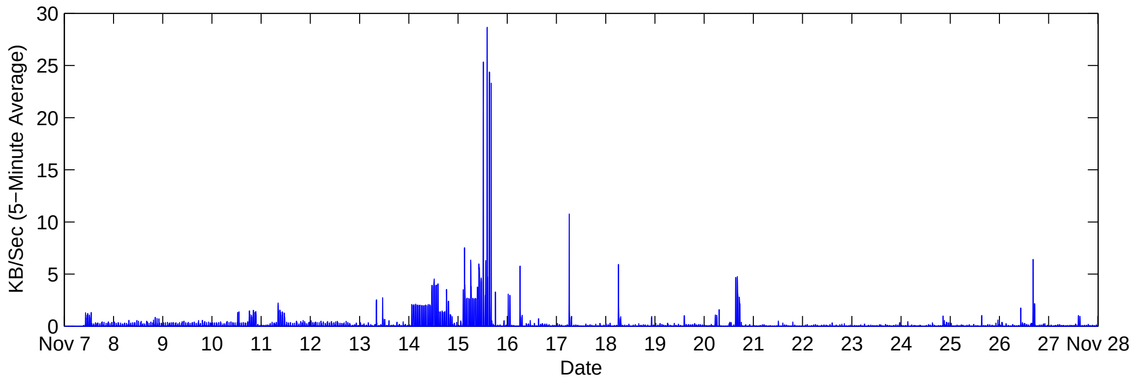

Figure 4: Total bandwidth from probes and attacks on decoy machines during a three-week period. Traffic spikes from heavy attack activity can be seen on November 25th.

the feasibility of exploiting timing characteristics to identify decoy services. We then discuss the effectiveness of application-layer probing and out-of band data collection. Finally, we take a look at the possibility of passively monitoring traffic to identify legitimate services.

6.1 Denial of Service

The most straightforward way of attacking OpenFire is to overload the decoy machines to the point where they can no longer accept any TCP connections. Subsequent network scans will reveal all of the real services because the decoys will not be able to respond to probes. The key to this attack is being able to target the decoys. Although it is difficult to tell if a particular {IP, port} destination maps to a decoy, an evenly distributed attack against all ports and IP addresses will overload the decoys first because they cover more address space per machine. In our experimental setup with 39 IP addresses, for example, a denial of service attack on SSH evenly distributed across 39 destinations would overload the two decoys first. This is because they are servicing requests for 37 addresses, while the two normal machines are only servicing requests for two.

The first and most preferable way of countering a denial-of-service attack is to use a protection system that is already available [4, 7, 12, 20]. If such a system is unable to stop a DoS attack, there are other OpenFire-specific countermeasures available to mitigate the attack. One way to deal with such an attack is to have each decoy only accept traffic to a few {IP, port} destinations, similar to a normal computer. With each decoy accepting less traffic, the normal machines and the decoys will become overloaded at the same point, making a DoS attack ineffective for fingerprinting. This approach, however, requires a lot of resources for the decoy machines, which may be too costly.

Another way of counteracting denial-of-service attacks is to automatically shift traffic away from one decoy onto another it if starts to become overloaded. This method will work well for attacks that are focused on a small portion of the address space that would normally bring down only a few decoys. If enough firepower is available to overload all of the decoys, however, then load balancing will not be able to stop the attack. We believe that a combination of current protection mechanisms and decoy load balancing for traffic spikes will make it very difficult to successfully run a denial-of-service attack against OpenFire. In the future, we plan to evaluate OpenFire's resilience against real DoS attacks and test the effectiveness of these protection methods.

Service Type and Location	**Mean Connect Time (ms)**	**Std Dev Connect Time (ms)**
Local - Dummy Service	1.83	1.48
Local - Decoy SSH	1.21	0.43
Local - Real SSH	0.76	0.20
Remote - Dummy Service	29.7	14.6
Remote - Decoy SSH	29.4	12.3
Remote - Real SSH	28.4	12.2

Table 3: Average and standard deviation of 100 TCP connect times for different services after removing outliers caused by dropped packets. Local probes were run from the University network. Remote probes were run from a different ISP.

6.2 Timing Analysis

When deploying OpenFire, it is important that the decoys and the legitimate machines respond to probes with approximately the same speed. If response times vary significantly between decoy and normal machines, then an attacker can map the real services using timing analysis. To test our implementation for timing vulnerabilities, we ran a large number probes both from a machine inside the University network (but still external to OpenFire) and from a home machine with a different internet service provider (ISP).

The results of running 100 TCP connect probes from two different locations on dummy, decoy, and real services can be seen in Table 6.2. Probes from the local network were answered quickly because they did not have to travel very far. Because the network latency was low, differences in application response time stood out. We observed that the decoy SSH server responded 500 s slower on average than the real SSH server for our implementation. We attribute this difference to inefficiencies in virtual machines, especially because the decoy VMs were running on the same physical machine as the gateway. To verify this hypothesis, we deployed a physical decoy machine that accepted traffic to multiple IP addresses and observed response times similar to those of the real SSH service.

The results from running the timing analysis on the dummy service and the decoy SSH service also differ slightly, which brings up an important issue. Different applications will have different response times, even to simple connect requests. This is because the connection latency partially depends on how the particular program is written. If the application continuously polls a large number of sockets, then connections may take longer to complete on average and experience greater variance in completion time. On the other hand, if the application uses asynchronous event callbacks, then connections will probably be accepted faster and with less deviation in completion time.

For the remote scans, the mean connection time was much higher at around 30 milliseconds. The standard deviation of the delay times was also higher, which means that more samples are required to get a good approximation of the mean. Although the distribution of connect times does not completely follow a normal distribution, it is approximately normal if outliers caused by queuing delays and dropped packets. After removing outliers, which account for approximately 25% of the samples, the standard deviation of the remote connection times for decoy and real SSH servers is approximately 1.7. Because the actual mean connection times are only 0.45 ms apart, the margin of error for each mean calculation should be no more than about 0.20 ms. Using a confidence interval of 90% (being 90% certain that the actual mean is within 0.20 ms of the measured mean), the number of good samples required to determine if a service is real or a decoy would be approximately 200. Given that only 75% of the samples are good, one would need to run about 250 probes on a service to be able to tell whether it is a decoy with 90% confidence. Doing this requires significantly more work than a typical scan.

There are a few possible methods for mitigating these timing attacks. First, adding functionality to OpenFire so that it automatically responds to TCP SYN packets with a SYN/ACK will create a constant connection response time for all services. This method, however, does not help with timing attacks later in the life of a connection. For example, it would not prevent analysis of HTTP response times for a web server. Another way of eliminating differences in application response time is to run decoys on dedicated physical machines, or run real services on virtual machines. This solution, however, is not always possible in real network with limited resources. Finally, the most effective way of dealing with timing attacks is latency adjustment. OpenFire could balance the latency of packets coming from different sources by adding a delay to packets from faster services. Adding this delay would be acceptable for most services because it is only on the order of a few hundred microseconds, which is small compared to typical network delay. In the future, we plan to add this capability to the OpenFire system and re-evaluate its susceptibility to timing analysis.

6.3 Application-Layer and Out-of-Band Fingerprinting

Another way of fingerprinting systems protected by OpenFire is to collect data at the application layer or from an out-of-band source. For example, if instructions are posted on a public web page telling employees how to connect to an SSH server from home, then an attacker can assume that the server names listed on the site are not decoys. Furthermore, once an attacker discovers targets in this manner, OpenFire does not prevent him or her from fingerprinting the target's application or OS version by sending probes to an individual port. If a hacker identifies a real HTTP server, for example, OpenFire cannot prevent that person from obtaining its application version by sending an HTTP request and looking at the "Server:" field in the reply. In general, OpenFire is unable to protect publicly known services, such as an organization's www web server, from focused probing and attack.

6.4 Passive Monitoring

The final vulnerability we consider here is a weakness in all honeypot and decoy systems. If an adversary wishes to determine which services are real and which are not, then he or she could passively monitor traffic going in and out of the network. If an attacker has access to SSH traffic, for example, then it may be possible to identify real SSH servers by the number of successful and failed login attempts. In general, if a particular service has a higher ratio of legitimate activity to malicious activity, then it is less likely to be a decoy or honeypot.

Actually obtaining all the traffic going in and out of a network, then being able to process such a large amount of data to discover decoys is not a trivial task. Successfully executing such an attack would require a great deal of skill and resources. For some networks, however, attackers with these capabilities may be a serious concern. A possible solution to this problem, which we plan to consider in the future, is to inject crafted legitimate network activity to prevent passive decoy identification. If the legitimate traffic is mixed with bogus data, it would be much harder for an attacker to discover real services through passive monitoring.

7 Future Work

In the future, we hope to deploy OpenFire in a variety of networks with different levels of service diversity and IP address concentration. This would allow us to determine the effects of address space sparseness on the number of attacks seen at the decoys and at the real machines. We would also like to experiment with installing services on the decoys that do not exist on legitimate machines. We would like to compare the number of attacks on legitimate machines in this configuration to the current OpenFire results to see if this setup provides extra protection. Running this configuration would also provide insight into the number of false positives that will occur if the decoys are running extra services not originally present in the network.

We also plan on investigating activity injection to prevent passive decoy identification. This problem has wider applications for many types of honeypots on the internet today. Injecting additional traffic into the network to make decoys or honeypots look ”more real” may help to lure more skilled hackers and teach us more about their attack patterns.

8 Conclusion

In contrast to traditional firewall systems, OpenFire aims to secure networks from remote attack by allowing traffic to all destinations. Instead of permitting the traffic to go to its intended destination, however, OpenFire forwards unwanted messages to a cluster of decoy machines. These decoys differ slightly from traditional honeypots because their goal is to protect other machines, not just to lure malicious attackers. During a three-week evaluation period, OpenFire was able to reduce attacks on legitimate machines by 57%, while increasing attacks on decoys by over three times when compared to a competing honeypot configuration.

We also tested OpenFire against popular scanning tools and found that it made their results much less reliable. Transport-layer scanning revealed that all ports on all destinations were open. Application-layer scanning was able to identify so-called dummy services that just accepted traffic but did not respond to messages. Operating System fingerprinting attempts were able to identify the OS running on a majority of the machines in the OpenFire network, but were unable to reliably determine the OS of any particular host.

Finally, we looked at limitations and possible attacks on OpenFire. Even though OpenFire does a great job of protecting machines from transport layer scans, it is unable to prevent application-layer and out-of-band fingerprinting. If an attacker is able to determine which hosts inside a network are running real servers by reading the organization’s web page, for example, then OpenFire is unable to protect those servers from attack. Furthermore, once an adversary has identified a real service, OpenFire cannot prevent application-specific probes from gaining information about the target. It would also be possible for an attacker to run a DDoS or timing attack to identify decoy services on the current implementation of OpenFire. In the paper, we discussed methods for countering these attacks.

In summary, OpenFire provides a new method of protection for networked computers from remote attack. Although it does not divert all malicious traffic, OpenFire significantly reduces the number of attacks on legitimate machines in a network. Furthermore, it provides a better early warning for remote attacks than previous honeypot systems because it accepts and diverts more traffic to the decoy machines. OpenFire does not make it impossible to attack a network, but it does significantly hinder malicious traffic from getting through to its intended destination, while at the same time making it much harder avoid attacking decoy machines.

References

[1] Yair Bartal, Alain J. Mayer, Kobbi Nissim, and Avishai Wool. Firmato: A Novel Firewall Management Toolkit. In *Proceedings of IEEE Symposium on Security and Privacy*, 1999.

[2] Cisco Intrusion Prevention System. http://www.cisco.com/en/US/products/sw/secursw/ps2113/.

[3] Evan Cooke, Michael Bailey, Farnam Jahanian, and Richard Mortier. The Dark Oracle: Perspective-Aware Unused and Unreachable Address Discovery. In *Proc. of the 3rd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, May 2006.

[4] T. Darmohray and R. Oliver. Hot spares for ddos attacks. http://www.usenix.org /publications/login/2000-7/apropos.html.

[5] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. A Sense of Self for Unix Processes. In *Proc. IEEE Symposium on Security and Privacy*, pages 120–128, 1996.

[6] Fyodor. Remote OS detection via TCP/IP Stack FingerPrinting. http://www.insecure.org/nmap/nmap-fingerprinting-article.txt.

[7] A. Garg and A. L. N. Reddy. Mitigation of DoS attacks through QoS Regulation. In *Proc. of IWQOS workshop*, May 2002.

[8] Internet Security Systems, Proventia Intrusion Prevention Appliance, 2006. http://www.iss.net/ products_services/enterprise_protection/proventia/g_series.php.

[9] Sotiris Ioannidis, Angelos D. Keromytis, Steven M. Bellovin, and Jonathan M. Smith. Implementing a Distributed Firewall. In *Proceedings of ACM Conference on Computer and Communication Security*, November 2000.

[10] Netfilter/Iptables Project. http://www.netfilter.org.

[11] Junpier Networks, Juniper Networks IDP 50, 2006. http://www.juniper.net/products/intrusion/idp50.html.

[12] F. Kargl, J. Maier, and M. Weber. Protecting web servers from distributed denial of service attacks. In *Proc. of 10th International World Wide Web Conference*, May 2001.

[13] G. Kim and E. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In *Proc. of the Conference on Computer and Communication Security*, 1994.

[14] O. Kolesnikov and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. Technical Report Technical Report: GIT-CC-04-15, GeorgiaTech, 2004-2005.

[15] C. Kreibich and J. Crowcroft. Honeycomb: Creating Intrusion Detection Signatures Using Honeypots. In *Proc. of the 2nd Workshop on Hot Topics in Networks (HotNets-II)*, Nov. 2003.

[16] M. Krzywinski. Port Knocking: Network Authentication Across Closed Ports. *SysAdmin Magazine*, 12:12–17, 2003.

[17] Nessus Vulnerability Scanner. http://www.nessus.org.

[18] Nmap - Network Mapper. http://www.insecure.org/nmap.

[19] V. Paxon. Bro: A system for detecting network intruders in real-time. In *Proc. of the 7th USENIX Security Symposium*, 1998.

[20] The peakflow platform, arbor networks. http://www.arbornetworks.com.

[21] N. Provos. Honeyd - A Virtual Honeypot Daemon. In *Proc. of the 10th DFN-CERT Workshop*, Feb. 2003. Hamburg, Germany.

[22] Niels Provos. A Virtual Honeypot Framework. In *Proc. of the 13th USENIX Security Symposium*, August 2004.

[23] Realsecure, internet security systems, 2006. http://www.iss.net/products_services/ enterprise_protection/rsnetwork/sensor.php.

[24] M. Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proc. of the USENIX LISA '99 Conference*, Nov. 1999.

[25] L. Spitzner. The Honeynet Project: Trapping the Hackers. *IEEE Security and Privacy Magazine*, 1(2):15–23, 2003.

[26] Lance Spitzner. *Honeypots: Tracking Hackers*. Addison Wesley Professional, 2002.

[27] J. Vijayan. Zero-day attacks seen as growing threat. In *ComputerWorld*, Dec. 2003. http://computerworld.com/securitytopics/security/story/0,10801,88109,00.html?SKC=news88109.

[28] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *Proc. of the Symposium on Operating Systems Principles*, Oct. 2005.

[29] David Watson, Matthew Smart, G. Robert Malan, and Farnam Jahanian. Protocol scrubbing: network security through transparent flow modification. *IEEE/ACM Transactions on Networking*, 12(2):261–273, 2004.