

On Recognizing Virtual Honeypots and Countermeasures

Xinwen Fu, Wei Yu, Dan Cheng, Xuejun Tan, and Steve Graham*

Abstract

Honeypots are decoys designed to trap, delay, and gather information about attackers. We can use honeypot logs to analyze attackers' behaviors and design new defenses. A virtual honeypot can emulate multiple honeypots on one physical machine and provide great flexibility in representing one or more networks of machines. But when attackers recognize a honeypot, it becomes useless. In this paper, we address issues related to detecting and "camouflaging" virtual honeypots, in particular Honeyd, which can emulate any size of network on physical machines. We find that an attacker may remotely fingerprint Honeyd by measuring the latency of the network links emulated by Honeyd. We analyze the threat from this fingerprint attack based on the Neyman-Pearson decision theory and find that this class of attack can achieve a high detection rate and low false alarm rate. In order to counter this fingerprint attack, we make virtual honeypots behave like their surrounding networks and blend in with their surroundings. We design a camouflaged Honeyd by revising a small part of the Honeyd toolkit code and by appropriately patching the operating system. Our experiments demonstrate the effectiveness of our approach to camouflaging Honeyd.

1 Introduction

Honeypots have recently been proposed as a means to complement the traditional defenses such as firewalls and Intrusion Detection Systems (IDS) for securing the computer networks that connect with the Internet. A honeypot can be defined as "an information system resource whose value lies in unauthorized or illicit use of that resource" [28]. In other words, a honeypot is just a trap in cyberspace and no legal users are supposed to use it. Any activities taking place on it are inherently suspicious and likely a probe, attack or compromise. In this way, honeypots have few false alarms in detecting intrusions, while other IDS solutions often suffer from high false alarm rates since they are trying to isolate intrusions from within a high volume of routine network traffic.

Honeypots take different forms to target different situations. According to the level of interaction an attacker is allowed to have with a honeypot, there are *low-interaction* and *high-interaction* honeypots, respectively. Low-interaction honeypots restrict the interaction between an attacker and the honeypot. This effectively limits the scope of the attacker's actions and minimizes the risk of compromise. On the other hand, high-interaction honeypots expose the whole system, including the operating system, to an attacker. On high interaction systems, attackers can gain full control of the system.

Honeypots can also be categorized with respect to their implementations. A physical honeypot is a real machine with its own operating system and IP address, while a virtual honeypot is a machine which emulates system behavior and IP addresses. Virtual honeypots have a number of benefits: a single physical system

*Xinwen Fu and Steve Graham are with the College of Business and Information Systems, Dakota State University, 820 N. Washington Ave. Madison, SD 57042 (Email:{Xinwen.Fu, skg}@dsu.edu). Wei Yu and Dan Cheng are with the Department of Computer Science, Texas A&M University, 301 Harvey R. Bright Bldg, College Station, TX 77843 (Email:{weiyu, dcheng}@cs.tamu.edu). Xuejun Tan is with the Department of Electrical Engineering, University of California, Riverside, 900 University Avenue - Riverside, CA, 92521 (Email:xtan@vislab.ee.ucr.edu).

can emulate more than one virtual honeypot; they can emulate more than one operating system; and they can emulate different IP addresses. Virtual honeypots require far less computational and network resources than physical honeypots, and they provide far greater flexibility in emulating various operating systems. In this research, we focus on studying virtual honeypots.

Honeyd [26] is a low-interaction virtual honeypot. It emulates computer systems at the network level. Honeyd emulates the TCP/IP stack and network services of an operating system in order to convince the attackers that they are exploring a “real” system. Honeyd can emulate network services such as HTTP and Telnet. It responds to a network request with response packets modified to match the network behavior of the operating system it has been configured to emulate. Besides emulating multiple services on multiple machines, Honeyd also emulates network topology and link characteristics such as link latency and packet loss.

To perform its function, a virtual honeypot such as Honeyd has to avoid being discovered. A virtual honeypot that cannot hide itself is analogous to a sniper without any camouflage. In this paper, we will show that virtual honeypots such as honeyd may demonstrate distinct temporal signatures that make them easily detectable. By examining link latency within a network emulated by Honeyd, an attacker can determine whether the corresponding links are virtual or real. This stems from the fact that virtual honeypots are not typically designed to emulate the temporal behavior of nodes at high fidelity. Honeyd is not designed to support arbitrary link latency resolution. Ordinary operating systems, for which Honeyd is designed, typically do not support the necessary timing resolution either. As a result, the virtual honeypot displays a significantly different temporal signature from the physical systems and networks they are emulating, and the virtual link characteristics differ statistically from those of surrounding physical links. Using the Texas A&M University campus network as an example, we analyze the threat from this fingerprint attack using Neyman-Pearson decision theory and find that this class of fingerprint attack can achieve a high detection rate with a low false alarm rate.

To counter this class of fingerprint attack against virtual honeypots such as Honeyd, we design a camouflaged Honeyd, which supports a link latency on the order of one microsecond instead of the original one millisecond default. Correspondingly, we generate a new patch to Linux 2.6.10 in order to support the new Honeyd implementation. Using the Texas A&M University campus network as an example again, we demonstrate the effectiveness of our camouflage approach to hide Honeyd within a metropolitan network (MAN, a network comprised of multiple LANs). We develop the actual toolkits for camouflaging Honeyd and rewrite a small part of the original Honeyd toolkit code. These toolkits and instructions can be found in [14]. We give the set of brief instructions in Appendix A.

The rest of this paper is organized as follows. Section 2 presents the related work. We discuss the network model and threat model in Section 3. In Section 4, we investigate how an attacker may discover Honeyd by measuring the link latency and evaluate its threat against Honeyd. In Section 5, we discuss the necessary approaches to camouflage Honeyd from the attacker. We conclude this paper in Section 6.

2 Related Work

In this section, we will survey various implementations of honeypots and then follow that by exploring work related to attacks and defenses for honeypots. Honeypots are classified into physical and virtual honeypots according to their implementation. In addition, they can be classified as low-interaction or high-interaction.

User-Mode Linux (UML) [9] is designed as a safe, secure way of running Linux versions and Linux processes on one machine. UML provides a virtual machine that may have more virtual hardware and software resources than a physical computer. The virtual machine is an image stored as a file on the physical machine and can be configured to have limited hardware access. VMware Workstation is powerful desktop virtualization software for software developers/testers and IT professionals who want to streamline software

development, testing and deployment in their enterprise. VMware Workstation [34] allows users to run multiple x86-based operating systems, including Windows, Linux, NetWare and their applications simultaneously on a single PC in fully networked, portable virtual machines without hard drive partitioning or rebooting required. Both UML and VMware can be used as virtual high interaction honeypots since they run full-fledged services on one machine.

Honeyd [26] is categorized as a low-interaction virtual honeypot and is implemented to emulate computer systems at the network level. In particular, Honeyd emulates the TCP/IP stack of a target operating system in order to convince attackers that they are exploring a “real” system. Honeyd is capable of responding to network requests according to services the virtual honeypot is configured to provide. It personalizes the response packets in order to match the configured operating system network behavior. In addition to system activities, Honeyd also emulates the network behavior, including the network topology and link characteristics such as latency and packet loss.

A Honeynet [15] is a high-interaction physical honeypot, a research effort originated by the Honeynet Project organization with the goal “to learn the tools, tactics, and motives of the blackhat community and share these lessons learned.” A honeynet may contain one or more honeypots and is highly monitored because of the likelihood of being compromised. Sebek [2] is a data capture tool designed to capture an attacker’s activities within a honeypot. It has two components. The first is a client that runs on the honeypots; its purpose is to capture all of attackers’ activities (keystrokes, file uploads, passwords) then covertly send the data to the server. The second component is the service which collects the data from the honeypots. The data collection service normally runs on a system, called the *Honeywall gateway*, which both captures and controls all inbound and outbound honeynet activity [16].

Dornseif, Holz and Klein [10] presented an approach for attacking a *honeynet*. The attack is essentially an exploit of Sebek. Due to the unique characteristics of Sebek, which is a kernel-based toolkit capable of recording all data accessed by users via the system call *read()* on the honeynet, the proposed NoSEBrEaK tool tries to detect the existence of Sebek by measuring the increase of round-trip time when Sebek is in a process of transferring a large amount of data over the network to the logging host. However, it is also recognized in the paper that the effectiveness of this attack can be mitigated by hiding Sebek’s traces more carefully. Actually some updated versions of Sebek have addressed the issue. The paper also presents ways to circumvent logging by Sebek on a Honeynet. The idea is to avoid using the *read()* system call which is used by Sebek to log the user activities.

Some related work is published within the blackhat community. Two articles have been published in two issues of Phrack [23], a Hacker magazine, about honeypot identification. One is entitled “Local Honeypot Identification” [7] and presents a method of disabling Sebek by overwriting its *read()* system call with the original value. Also, it proposes an approach to detect the existence of Sebek by challenging the outgoing connection limit used on most Honeynets. Corey [8] proposed searching through the memory and reconstructing several sensitive parameters used by Sebek to hide the traces of the exploit. However, this has been addressed in the updated version of Sebek.

Kohno, Broido and claffy developed a method to fingerprint physical devices based on the skews of the devices’ physical clocks [19]. That approach can be used to determine whether different addresses correspond to virtual hosts on the same physical machine. However, that approach would not work if the hosts being fingerprinted do not provide timestamps (e.g., with TCP timestamp option disabled). In this paper, we provide a complementary approach that does not rely on timestamp information from the target hosts.

In this paper, we will discuss techniques to measure link latency. Accurately measuring link latency is still a challenging research topic. A few network measurement tools (pathchar [17], clink [12, 11], pchar [21] and pipechar [5]) use a similar philosophy for the link latency measurement. Those tools utilize the field of *TTL* (time to live) in ICMP or UDP packets. We provide a complete set of approaches to measure link latency in our context, including the use of TCP packets.

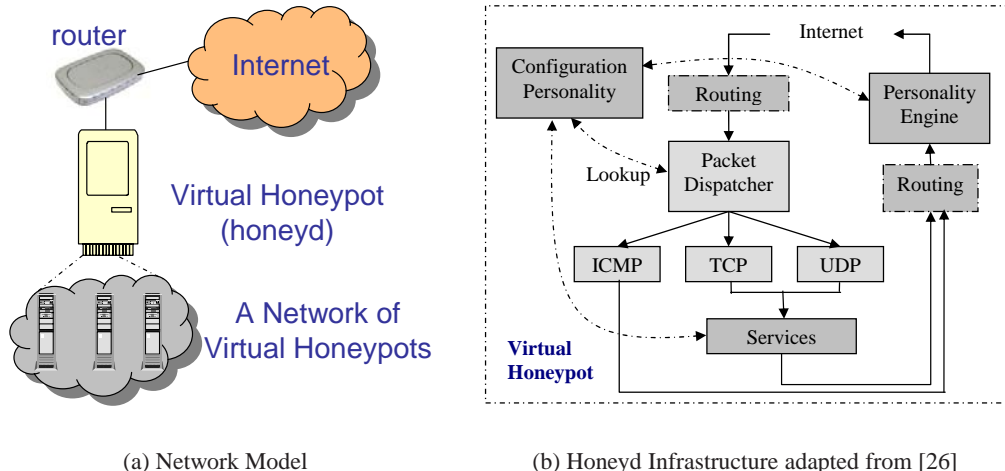


Figure 1: Virtual Honeypot Models

In the following, we will illustrate how the lack of temporal fidelity in the emulation of hosts by virtual honeypots leads to timing signatures that can be very easily exploited to identify honeypots with high accuracy.

3 Models

In this section, we introduce the network model and threat model used in this paper.

3.1 Network Model

Honeypots are decoys deployed in a network in order to trap attackers (and attack) and to learn their tools, tactics and motives. With this information, we can better understand and protect against threats. Figure 1(a) illustrates the network model for a virtual honeypot. The virtual honeypot can emulate either a single host or a network, whose setting and characteristics are configurable. We denote the emulated network as a *virtual network*. The virtual honeypot connects to the open network, such as the Internet, through a physical network interface. This physical interface can connect to a security gateway, which logs, monitors and limits activities on the virtual honeypot. In this way, the virtual honeypot is open to hacking, worms, spams and any other attack. This model is a general framework and can represent both high-interaction virtual honeypots such as User-Mode Linux (UML) and low-interaction virtual honeypots such as Honeyd for the objective of intrusion detection.

Virtual honeypots are the focus of this paper. Figure 1(b) illustrates the infrastructure of Honeyd. A virtual network emulated by Honeyd consists of a variety of components such as routers and end hosts. The emulated hosts may behave like a machine installed with Windows, Unix, Linux or other operating systems. The characteristics of these network units are delimited by specific scripts. Packets to the virtual network are processed in the following way: when the packet dispatcher receives a packet, it traverses the correct entry routing tree from the root node to the destination node. The packet latency on all edges of the path is accumulated to determine how long its delivery should be delayed. The packet delivery uses the event scheduling mechanism provided by *libevent* [24]. The libevent APIs are a wrapper of the underlying operating signal APIs and provide an alternative mechanism to support callbacks due to signals or regular timeouts.

3.2 Threat Model

Before we proceed, we need to clarify the capabilities of an attacker who is interested in determining whether a network is a virtual one created by Honeyd or a real physical one.

1. The attacker is an active one. They can inject probing traffic into the Internet remotely as normal users do. They can adjust the frequency and intensity of the probing traffic in order to prevent it from being discovered.
2. The attacker can utilize tools to discover the network topology [20]. Then, the attacker can choose one suspect link to test whether it is real or virtual. We will discuss how an attacker may use a tool to discover a metropolitan network topology.
3. The attacker knows the emulation strategies and algorithms of virtual honeypots. This assumption is commonly used in security research.

4 Recognizing Honeyd

In this section, we first give an overview on how an attacker may fingerprint a virtual network emulated by Honeyd. We then define the problem formally and introduce issues related to the fingerprint attack.

4.1 Overview and Problem Definition

Since large portions of the operational nodes (for example, users, server programs and network devices) are missing from virtual honeypots, emulating their operations requires triggering by timer events or signals. Packets to the virtual network are dispatched by timers. The accuracy of event scheduling is determined by the timing accuracy of the underlying operating system. In modern operating systems, timer interrupts are generated by the system's timing hardware at regular intervals. At boot time, the operating system kernel sets this interval according to the kernel parameter *HZ*, which is architecture-dependent. In this paper, we focus on studying Honeyd deployed on Linux, for which *HZ* is defined in `<linux/param.h>` or a subplatform file. For example, `/usr/src/linux2.6.10/include/asm-i386/param.h` is the file on the popular x86 PC. Default values of *HZ* range from 50 to 1200 ticks per second on real hardware, down to 24 for software simulators. Most platforms run at 100 or 1000 interrupts per second. The x86 PC defaults to 1000 in Linux kernel 2.6 and 100 in the previous versions of Linux kernel up to 2.4 [6].

Without loss of generality, we will only discuss the popular x86 PC platform in the remaining parts of this paper, but the results in this paper and our analysis approach are readily applied to other platforms. The above interrupt rates mean that the link latency emulated by systems like Honeyd can only achieve an accuracy of 10ms and 1ms for Linux 2.4 and 2.6 kernels respectively. Therefore, the link latency within a virtual network will always be around a multiple of 10ms or 1ms. There are two security concerns raised by this observation: (1) a link latency with multiples of 10ms or 1ms can be a fingerprint of Honeyd; (2) with today's network hardware and bandwidth capacity, the link latency within a wired network can barely reach 1ms, let alone 10ms. This discrete latency gives rise to a timing signature of a virtual network emulated by Honeyd. Thus, if an attacker can recognize that a link exhibits such a signature, they can conclude that this is an emulated link and the corresponding network is virtual rather than physical.

Therefore, the problem of recognizing Honeyd can be defined as follows: *Given a link connecting two routers within a network, how can an attacker determine by measuring the link latency whether the link is a real physical link or one emulated by Honeyd?*

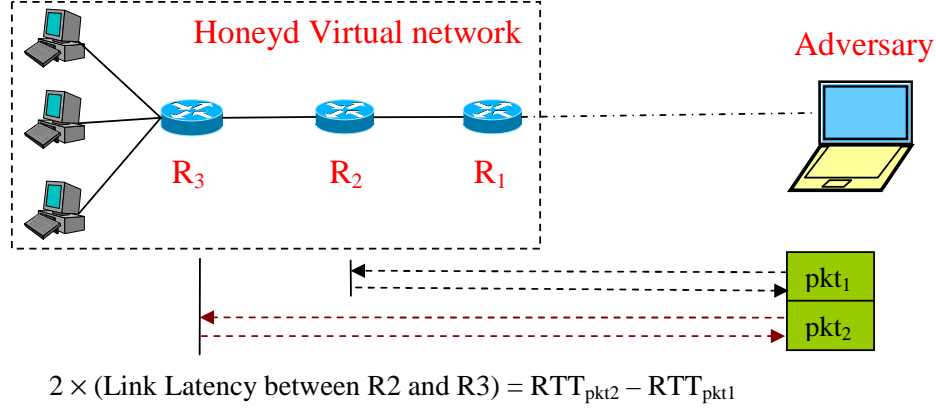


Figure 2: Principle for recognizing Honeyd

4.2 Issues Related to Recognizing Virtual Honeyd

From the discussion above, we know the success of recognizing Honeyd lies in accurately measuring link latency of a remote network. Figure 2 demonstrates the principle method of deriving link latency¹. In Figure 2, RTT refers to Round Trip Time. To find the link latency of a network link connecting routers R_2 and R_3 , an attacker can send a pair of packets $\langle pkt_1, pkt_2 \rangle$ of minimal size. pkt_1 is addressed to router R_2 while pkt_2 is addressed to router R_3 . Intuitively,

$$\text{two-way link latency} = 2 \times LL_{R_2, R_3} \quad (1)$$

$$= RTT_{pkt_2} - RTT_{pkt_1} \quad (2)$$

where RTT_{pkt_n} is the RTT of packet n and LL_{R_2, R_3} is the one-way link latency between R_2 and R_3 . In the following discussion, when we mention link latency, we refer to the two-way link latency for brevity.

Network noise may influence the accuracy of link latency measurement [12]. When a packet traverses the Internet, the intermediate routers and transmission media may disturb the timing of the packet. This forms the network noise, which disturbs the measurement of RTT. In our experiments, we use the same statistical analysis as the one used by *clink* [12, 11] for deriving link latency². Interested readers may refer to [12] for the details. Accurately measuring link latency is still a challenging research topic.

Therefore, from the discussion above, we can see that from the viewpoints of both attackers and defenders, there are four critical issues:

1. What kind of packets can the attacker utilize to derive the round trip times and the link latency?
2. How can an attacker recognize a virtual network using the measured link latency?
3. How effective is this form of fingerprint attack?
4. How can we hide Honeyd from this form of fingerprint attack?

We will answer these questions in the following sections.

¹We use a virtual network topology similar to the one defined in *config.sample*, shipped with Honeyd 1.0 [25].

²*clink* utilizes UDP traffic as the probing traffic. We also apply its measurement principle to ICMP and TCP traffic.

4.3 Measurement of Round Trip Times

In this section, we answer the first question in Section 4.2: “What kind of packets can the attacker utilize to derive the round trip times and the link latency?” The key to successfully recognizing a virtual network is to accurately derive the link latency. For the approach illustrated in Figure 2, an attacker has to measure the RTT of a packet sent from the attacker’s probing machine to the victim router. This requires that the router generates a reply packet to a probing packet. The RTT can then be calculated as the difference between the sending time of the probing packet and the receiving time of the reply packet.

There are several ways for an attacker to force a reply packet from nodes (routers or end hosts), depending on what services are provided on the node.

4.3.1 Ping Based Approach

This might be the most straightforward approach. Ping utilizes the Internet Control Message Protocol (ICMP), which is part of the IP layer. ICMP communicates error messages and other condition messages. These messages can be processed at either IP layer or transport layer protocols (TCP or UDP). The Ping program sends an ICMP echo request message to a host and expects an ICMP echo reply to be returned in order to check if a network unit is alive [29].

4.3.2 TCP Based Approach

There are two main classes of TCP based approaches to derive the RTT. First, if the victim host supports some well-known TCP services, the attacker can exploit these services to get the RTT. For example, if the victim supports *Telnet*, an attacker can exploit the procedure of establishing TCP connections (the three way hand shake). When an attacker sends a SYN packet to the victim router, the TCP service will return an ACK/SYN packet in response to the attacker’s SYN packet.

Even without any well-known TCP service on the victim router, the attacker may still be able to exploit the TCP protocol to derive the RTT. A *reset* (RST) is sent by TCP whenever an arriving TCP segment doesn’t appear correct for the corresponding connection. Thus, the attacker can send a bogus TCP packet (i.e., a packet with a bogus TCP source or destination port number) to the victim router. Then the router will return a TCP RST packet to the attacker. Thus, the attacker obtains a reply packet and can calculate the RTT. We use this approach in our experiments.

To correlate sent and received TCP packets, the attacker can utilize the TCP sequence number and acknowledgement number. The attacker can send a series of bogus TCP packets with different sequence numbers, and the router will return TCP RST packets with corresponding acknowledgement numbers.

4.3.3 UDP Based Approach

Similarly, RTT information can be gathered with UDP as well. If the victim network unit supports some well-known UDP services, the attacker can exploit these services to get the RTT. If no such services are supported on the router, the attacker may still be able to exploit the UDP protocol to get RTT information.

Similarly to the case of TCP, the victim router may respond to erroneous UDP packets. For example, when UDP receives a datagram and the destination port does not correspond to a port that some process is using, UDP responds with an ICMP “port unreachable” packet [29]. Thus, the attacker can send a bogus UDP packet to the victim router. The bogus UDP packet specifies a port not in use on the victim router, which will return an ICMP “port unreachable” packet. The attacker is also able to correlate a sent UDP packet to a received ICMP packet by using the source or destination port because the ICMP packet contains the UDP header. We use this approach in our experiments.

Finally, it must be pointed out that any of the methods mentioned above (and others) can be easily foiled by not having the virtual honeypot respond to the request or to the erroneous packet. This, however, would render the honeypot “silent”, in which case it can only be used as a port knocking detector. In order for honeypots (physical or virtual) to be useful for gathering information about attacks, they must at least engage the attacker in some packet exchange, at which point some RTT information can be gathered.

4.4 Recognizing Virtual Honeypots

Once the RTT information has been gathered, and the link latency determined, a decision must be made on whether the link is part of a virtual honeypot. This section answers the second question in Section 4.2: “How can an attacker build a profile and recognize Honeyd based on the profile?”

4.4.1 Framework of Detecting Virtual Honeypots

The above problem is naturally a decision problem. We summarize the framework of detecting virtual honeypots based on pattern recognition [13] in Figure 3. Generally speaking, the goal of the pattern recognition process is to use classifiers to *classify* an unknown pattern as belonging to one of several existing patterns, *classes*, with the help of some distinguishing feature. In this paper, we use link latency as the feature and leave other features to exploit in our future work..

In the problem of detecting virtual honeypots, there are only two classes of events:

$$\begin{aligned} \omega_0 : & \text{ A suspect link is a real link} \\ \omega_1 : & \text{ A suspect link is a virtual link} \end{aligned} \tag{3}$$

Therefore, following the common practice, a pattern recognition system for detecting virtual honeypots consists of two phases: (a) off-line training and (b) on-line recognition.

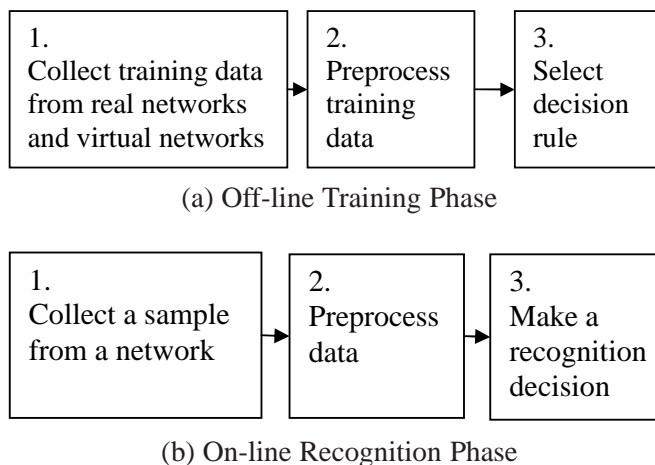


Figure 3: Framework for Recognizing Virtual Honeypots

Figure 3 (a) is the procedure for the off-line training phase.

1. *Collect training data from real networks and virtual networks:* An attacker applies the probing traffic to known real networks and virtual networks. They collect the RTT data samples for both real links and virtual links.

2. *Preprocess training data*: The attacker derives two classes of link latency data sample from the RTT data by using approaches described in Section 4.3: real link latency sample and virtual link latency sample. From these two classes of link latency sample, the attacker is able to derive two distributions: real link latency distribution, denoted as $f_0(x)$, and virtual link latency distribution, denoted as $f_1(x)$. In this paper, we assume that the attacker uses kernel based density estimation approach, which will be introduced in a following section, to derive the distribution numerically.
3. *Select decision rule*: The attacker selects an appropriate classification rule based on the training data and the two trained link latency distributions. In this paper, we assume that the attacker uses a classifier based on the Neyman Pearson theory, which will be introduced in a following section.

Figure 3 (b) is the procedure for the on-line recognition phase. The procedure is similar to the off-line training phase. An attacker collects a link latency sample from a suspect network. Then the attacker uses the trained classifier to decide whether the link is real (class ω_0) or virtual (class ω_1).

4.4.2 Link Latency PDF Estimation

From the framework of detecting virtual honeypots in Figure 3, we can see that an attacker should be able to estimate the link latency probability density function (PDF) from link latency measurement samples. Among the many ways that can be used to estimate density functions, we assume that an attacker uses the *non-parametric Gaussian Kernel based density estimation* method [27]. The major advantage of this approach is its flexibility to represent very complicated densities.

The kernel based density estimation uses the superposition of a normalized window function centered on a set of random samples, such that

$$f(x) \approx \frac{1}{N} \sum_{i=1}^N G_{\psi}(x - x_i) \quad (4)$$

where $f(x)$ is the estimated density function of features, N the sample size, $\{x_i : 1 \leq i \leq N\}$ the random sample of link latency, $G_{\psi}(x)$ the window function, and ψ the window width. N must be big enough to capture the class (population) characteristics. In this paper, we assume G_{ψ} to be a normalized Gaussian (Normal) density distribution, that is

$$G_{\psi}(x - x_i) = \frac{1}{\sqrt{2\pi}\psi} e^{-\frac{(x-x_i)^2}{\psi^2}} \quad (5)$$

We need to carefully choose ψ and N to control the smoothness and bias of the estimated density function. Silverman recommends an optimal window width in [27] and Beardah [3] implemented those approaches in Matlab. We use the Kernel Density Estimation Toolbox (Matlab) by Beardah [3] for all the density estimation in this paper.

4.4.3 Classifier Based on Neyman-Pearson Decision Theory

In this paper, we assume that an attacker uses a classifier based on the Neyman-Pearson (NP) decision theory [13] for recognizing virtual honeypots. The advantage of this theory is that Neyman-Pearson classification does not assume knowledge of or about a priori class probabilities such as those used in Bayesian classification theory.

We define *detection rate* P_D as the probability that a link emulated by Honeyd is detected as a virtual link. *False-alarm rate*, P_F , is the probability that a real link is misclassified as a virtual link. We have the following Neyman-Pearson criterion:

$$\text{Given a maximum false alarm rate } \alpha, \text{ maximize detection rate } P_D \text{ such that } P_F \leq \alpha \quad (6)$$

Neyman-Pearson criterion tells us that we should construct our decision rule to have the maximum detection rate while not allowing the false alarm rate to exceed a certain value α . In other words, the optimal detector according to the Neyman-Pearson criterion is the solution to this constrained optimization problem in (6).

This optimization problem has an explicit solution. It is given by the Neyman-Pearson lemma. We limit our discussion to continuous random variables. Recall $f_0(x)$ as the link latency PDF of a real link and $f_1(x)$ as the link latency PDF of a link emulated by Honeyd.

Neyman-Pearson Lemma: Define

$$\Lambda(x) = \frac{f_1(x)}{f_0(x)} \quad (7)$$

For a given maximum false alarm rate α , we can calculate a decision boundary γ' by (8):

$$P_F = \int_{\Lambda(x) > \gamma'} f_0(x) dx = \alpha \quad (8)$$

Then we have the following classifier

$$\Lambda(x) = \frac{f_1(x)}{f_0(x)} \begin{cases} < \gamma', & \text{the suspect link is real} \\ > \gamma', & \text{the suspect link is virtual} \end{cases} \quad (9)$$

Detection rate can then be calculated as follows:

$$P_D = \int_{\Lambda(x) > \gamma'} f_1(x) dx \quad (10)$$

Please note that here we assume that the mean of $f_1(x)$ is greater than the mean of $f_0(x)$. This assumption will be validated from our experimental results³ in Section 4.5.4.

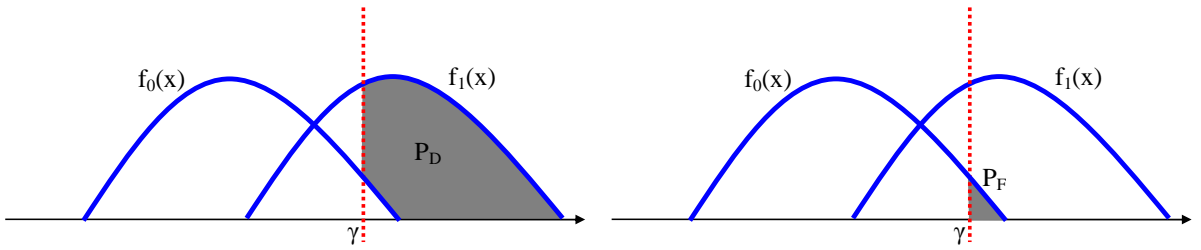


Figure 4: Calculation of P_D and P_F

When we use a classifier based on the Neyman-Pearson theory, because we know $f_0(x)$ and $f_1(x)$, the classifier in (9) is often translated, either theoretically or numerically, to a simpler classifier as shown in (11):

$$x \begin{cases} < \gamma, & \text{the suspect link is real} \\ > \gamma, & \text{the suspect link is virtual} \end{cases} \quad (11)$$

We will use (11) as the classifier in this paper. Figure 4 illustrates how to calculate detection rate P_D and false alarm rate P_F in this paper, where γ is the derived decision boundary.

We implemented the above detection rate and false alarm rate calculation in Matlab and the code is available upon email request.

³That is, since the virtual link latency is greater than the real link latency for most cases, we will be able to detect virtual networks based on the link latency measurement.

4.5 Evaluation

In this subsection, we evaluate the threat from an attacker who applies Neyman-Pearson decision theory to distinguish a Honeyd emulated virtual link from a real link.

4.5.1 Evaluation Metrics

In this paper, we use detection rate P_D and false alarm rate P_F as our evaluation metrics for detecting honeyd. From Figure 4, we can see that detection rate P_D and false alarm rate P_F have an interesting relationship. Both P_D and P_F decrease to zero as γ increases, while both P_D and P_F increase to one as γ decreases. A common means of displaying the relationship between P_D and P_F is with a *Receiver Operating Characteristic (ROC)* curve, which is a plot of P_D versus P_F . In reality, when an attacker tries to find a virtual honeypot by using the Neyman-Pearson theory, they want a high detection rate and a low false alarm rate. If we want to hide virtual honeypots, the ideal scenario is that an attacker will get a false alarm rate as high as the detection rate.

4.5.2 Experiment Setup

Figure 5 illustrates our experimental setup. We deploy a virtual Honeyd network in our laboratory⁴. The objective of the attacker is to decide whether the link between R_2 and R_3 is a Honeyd emulated link (the same configuration as the one in Figure 2. We have similar results for other virtual network topologies). Between the Honeyd virtual network and the attacker’s machines, we set up a machine with NISTnet [4] to emulate the influence from cross traffic or intermediate networks on the detection of Honeyd.

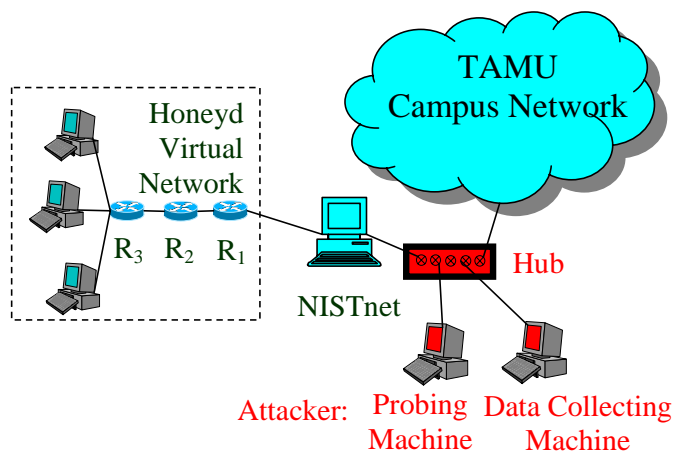


Figure 5: Experiment Setup

The machine we use for hosting the Honeyd virtual network is a Dell Dimension 8400 Intel desktop configured with Pentium 4 processor 640 supporting Hyper-Threading (HT) Technology (3.20GHz CPU featuring 800 MHz front side bus - FSB). The attacker controls two machines: the probing machine generating the TCP/UDP/Ping probing traffic and the data collecting machine. The attacker installs tcpdump [30] on the data collecting machine to dump traffic through the hub, which is connected to both the Honeyd virtual network and the Internet. We run Honeyd on Redhat 7.3 (Linux 2.4.18) and Fedora Core 3 (Linux 2.6.9), which are representatives of Linux 2.4 and 2.6 kernels.

⁴Because of the university administration issues, we cannot deploy Honeyd at a campus scale.

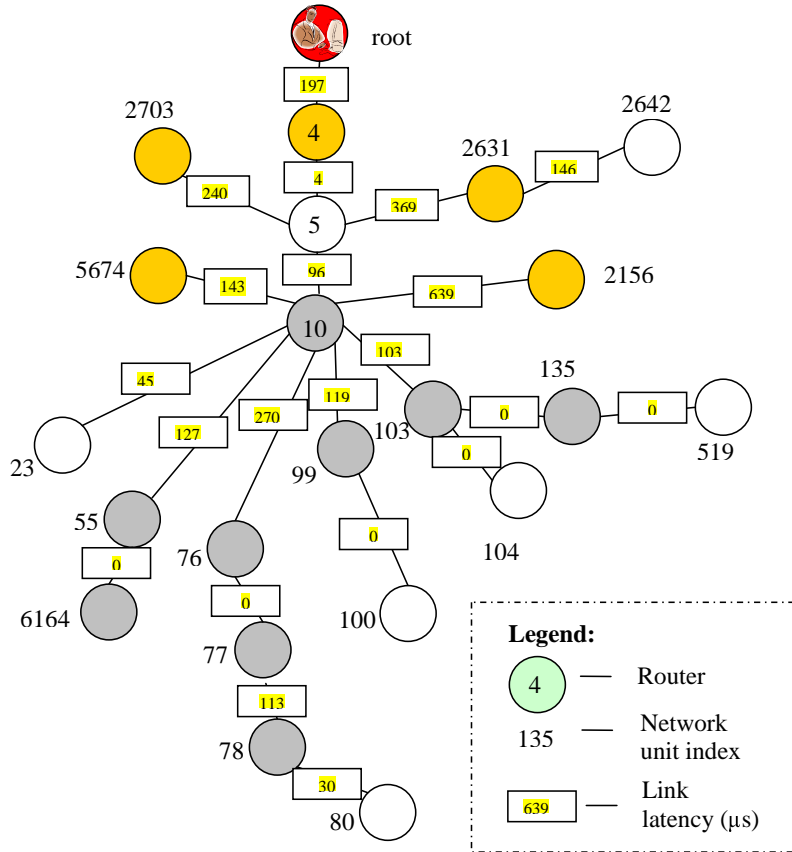


Figure 6: Part of Texas A&M University Campus Network Topology Discovered by Argus

4.5.3 Deriving Link Latency Distributions

In order to detect a Honeyd emulated link, the attacker will use the pattern recognition based approach described above. During offline training, attackers will derive the latency PDFs for both real and virtual links.

To derive the link latency distributions for the real surrounding network links, an attacker can first derive the network topology of the target network and then use one of the approaches in Section 4 to derive the link latency of all the links. Then, from the measured link latency samples, the attacker can derive the link latency distribution for the network. Recall the attacker uses the Gaussian Kernel based density estimation approach to derive the empirical density function in this paper, .

In our experiments, we use a network topology discovery tool called *Argus* [18]. This tool uses an ICMP/traceroute-based topology discovery methodology. It is flexible and can be used for most networks. We applied Argus against Texas A&M University’s campus network and found nearly 20,000 running network units. We give part of the campus network topology as shown in Figure 6. A circle (with or without shading) represents a router⁵. The number beside the circle is the network unit index created by Argus. The number in a square is the link latency of the corresponding link.

Figure 6 gives the spanning tree of the campus network from the perspective of the person at the root, i.e., an attacker who tries to derive the topology. If we choose more roots at different locations across

⁵Texas A&M University controls two class B IP address ranges. Router shading refers to which of the two class B networks a node belongs to and is irrelevant to this study.

the campus, we will derive other spanning trees of the same network. By combining those spanning trees together, we can derive a more complete network topology since a single spanning tree ignores the crossover links.

Figure 6 also labels the link latency of each link. We can use any approach in Section 4.3 to find the link latency. Many existing tools, such as *Pathchar* [17], *Clink* [12, 11], *pchar* [21] and *pipechar* [5] utilize a subset of those approaches. As a benchmark for this paper, we simply used *clink* in its default mode to derive the real link latency. When we specify an end host, *clink* will measure the latency of each link on the path to the end host. For each link latency measurement, *clink* uses 8×93 probes to derive a single link latency. Therefore, to measure the latencies of all the links within a spanning tree, we specify each of those leaf nodes in Figure 6 as the end host for *clink*. The whole process of scanning one spanning tree took about three hours. Most link latency measurement tools cannot deal with a few types of links such as parallel links. This will cause errors in computing link latencies. When we detect such errors, we label the latency as “0”, and such errors are not used when deriving link latency distributions.

Thus, by discovering the network topology and measuring the corresponding link latency, an attacker can derive the link latency distribution by using the Gaussian Kernel based density estimation approach as discussed in Section 4.4.2. Figure 7 shows the real link latency distribution. Negative values in the distribution are caused by the kernel based density estimation approach and can be ignored.

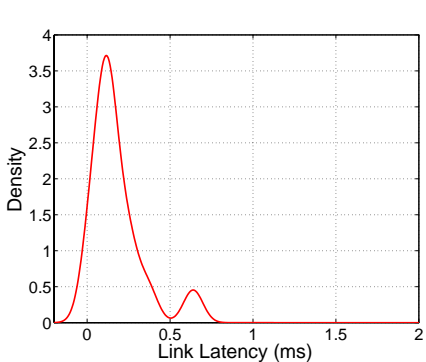
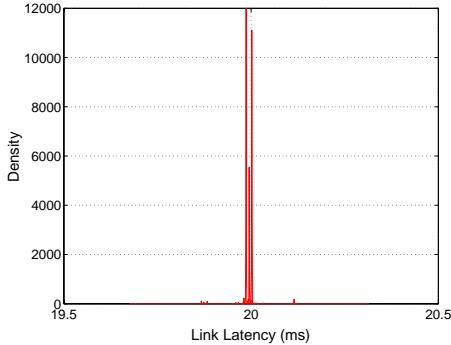
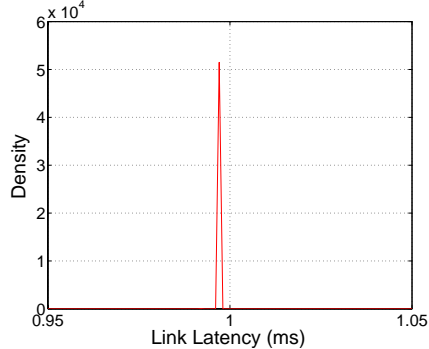


Figure 7: Campus Network Link Latency Distribution



(a) Link 2.4.18



(b) Linux 2.6.9

Figure 8: PDFs of Virtual Link Latency (Note the finer x-axis scale compared with Figure 7)

To derive link latency distributions of Honeyd emulated links, the attacker can use the setup in Figure 5 and derive link latency distributions for Linux 2.4 and 2.6 kernels. Figure 8 demonstrates the probability density function (PDF) of link latency when an attacker uses the TCP probing approach.

4.5.4 Failure of Current Honeyd Implementation

We have a few observations from Figure 7 and Figure 8.

1. We can see that the link latency emulated by Honeyd is roughly a multiple of 10ms on Linux 2.4.18 and 1ms on Linux 2.6.9, as we analyzed in Section 4.1. Clearly, the mean of each virtual link latency density function is larger than the mean of the real link latency distribution. This validates the assumption made in Section 4.4.3. It is a quirk of OS and honeyd scheduling policies that Linux 2.6.9 yields a factor of one times the default interrupt rate while Linux 2.4.18 yields a factor of two times the default interrupt rate.

- Moreover, the PDF curve for virtual link latencies is confined to a very narrow range. This means that a packet pair based probing approach in this case generates very accurate measurement of link latency.
- The real campus network link latency distribution in Figure 7 and Honeyd-emulated network link latency in Figure 8 are widely different along the axis of link latency.

The above observations can well explain the ROC curves in Figure 9 in the case of detecting a virtual link on Linux 2.6.9. From Figure 9, we have the following observations:

- An attacker is able to achieve a high detection rate while keeping the false alarm rate low for any of the TCP, UDP or Ping based approaches. For example, when the false alarm rate is around 2%, an attacker may achieve a detection rate of over 98% in all three cases. The reason is the Honeyd-emulated network link latency is much larger than the real network link latency. Moreover, the emulated link latency is narrowly confined to a region around 1ms or 20ms.
- TCP, UDP and Ping probes produce similar ROC curves. We expect this since our experiments are run within a campus network. If an attacker deploys the attack from a remote site, TCP and UDP probes will be able to achieve better performance because the Ping packets' low scheduling priority on the Internet. In the remaining part of this paper, we will not differentiate results from TCP, UDP and ICMP probes.

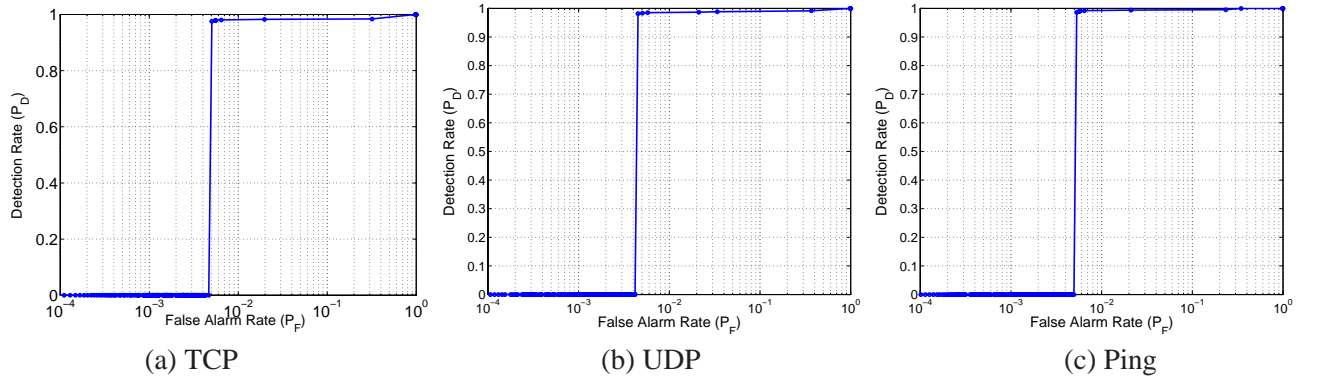


Figure 9: ROC for Fingerprint Attack

5 Camouflaging Honeyd

The key reason for the failure of the current Honeyd implementation (and likely other virtual honeypot implementations as well) is the low fidelity of emulation (in the time domain) of the system components by the virtual honeypot. These components may represent users, applications and servers in the case of physical honeypots, in addition to various hardware resources in the case of virtual honeypots. The processing delay caused by these components is emulated by timers in the virtual honeypot. When these timers are either carelessly defined in the virtual honeypot implementation or are provided at insufficient accuracy by the underlying OS, a timing signature emerges. In this case, an attacker may construct a profile of a virtual honeypot and launch a timing attack. In the case of Honeyd, for example, emulated virtual links are statistically very different from the surrounding physical network links.

5.1 Overview of the Principle

To camouflage Honeyd and defeat the type of timing attacks described above, we have to modify Honeyd and the underlying OS support to allow for a higher-fidelity emulation of events. This requires (a) configurable definition of accurate timing behavior, and (b) support for accurate triggering of events by the underlying OS. In our case, this means (a) accurately configurable link latencies, and (b) high-resolution timers within the OS. In this way, we can assign a reasonable link latency for Honeyd based on its surrounding network link characteristics, and an attacker will not be able to build a useful profile for Honeyd. We camouflage Honeyd in the following ways:

1. We change the Honeyd code to make it support a timing resolution of microseconds μs . This involves modifying both Honeyd and the event management library (libevent).
2. There are a number of ways to improve the OS support for high-fidelity timers. We can use the real-time event scheduling in commercial real-time operating systems [31] or one of various open source projects for high resolution timers [1]. The advantage of this approach is that we may achieve high-resolution timers at low overhead since these approaches typically make use of hardware support for accurate timers. The disadvantage is that we would need to rewrite the event management library (libevent).

In this paper, we improve the operating system’s timing accuracy by increasing the kernel parameter *HZ* (Refer to Section 4.1). We ported the HZ patch [33] to kernel 2.6.10. In this way, we don’t need to change the libevent library at all, however we may sacrifice some CPU efficiency. Since the physical machine running Honeyd should not be running any other applications, this is not crucial.

Thus, if the PDF of the Honeyd emulated link latency can approach the real network link latency PDF in Figure 7, an attacker cannot determine whether a link is emulated by Honeyd or not.

In the following, we will discuss first the CPU performance overhead caused by the increased HZ and then how well a high resolution timer can emulate the link latency. At last, we demonstrate the effectiveness of camouflaging virtual honeypots.

5.2 CPU Performance Overhead Caused by a High Resolution Timer

The use of an increased HZ is simply one approach for us to prevent virtual honeypots from being detected by the fingerprint attack. A better approach that requires much more effort is to rewrite the event library with the help of the Real-Time OS or dedicated high resolution timer API to reduce the overhead introduced by fine grained timer interrupts. This will be left as our future work.

Figure 10 demonstrates the CPU performance in terms of the timer resolution (i.e., $1/HZ$). We utilize the “speed” function in OpenSSL [22] to benchmark the system’s performance under different HZ settings. OpenSSL tests how many operations it can perform in a given time. Figure 10 (a) shows how many sign/verify operations (with a 1024 RSA public key pair) it can perform in 10s, while Figure 10 (b) shows the performance degradation with the performance at $HZ=100$ (timing interval 10ms) defined as 100%.

We have the following observations from Figure 10:

1. When HZ increases (timing interval decreases), the system performance decreases. From 10 (b), we can see that the performance degrades to 34% of the best performance (specified as 100%) when the timer resolution is 10us ($HZ=100,000$).
2. The performance degradation has an exponential curve. When HZ is relatively small, the performance degrades slowly. When HZ is relatively big, the performance degrades sharply.

Honeyd is also supposed to be able to process a large amount of network load. Thus we need to carefully select an HZ in practice.

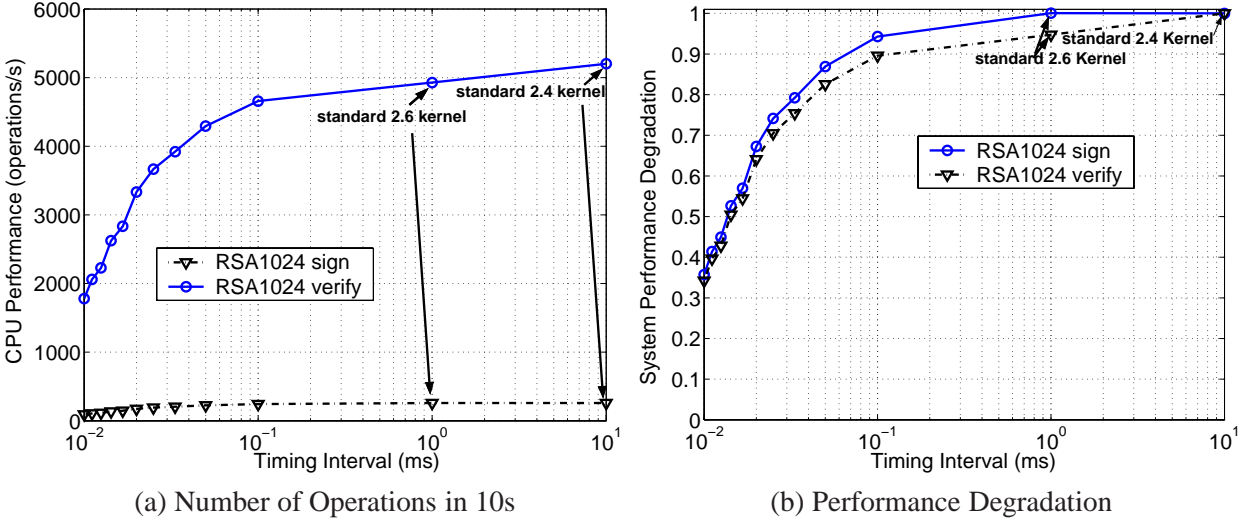


Figure 10: CPU Performance

5.3 Accuracy of Emulating Link Latency by a High Resolution Timer

Honeyd provides a configuration file to specify a network topology including its parameters such as link latency. When we deploy Honeyd, we would prefer that it emulate link latency at arbitrary accuracy. As we know, we cannot achieve this in reality because of the timing accuracy of the operating system. We measure the link latency emulation accuracy in the following pessimistic way: we set the link latency in Honeyd’s configuration file and increase the link latency $5 \mu\text{s}$ each time. Because of timer resolution limitations, at a specific timer resolution, the measured link latency may not change for every $5 \mu\text{s}$ increase we configure and will jump to a new value when the accumulated increase is big enough. Therefore, we use the maximum measured increase of the link latency as our emulation accuracy metric. The detailed experimental procedure is given as follows.

1. Set the HZ to a specified value, e.g., 20,000.
2. Increase the configured link latency of a link (such as the link between R_2 and R_3 in Figure 2) at a step of $5\mu\text{s}$ and measure the corresponding link latency. In our experiments, the link latency we specified is $\{1, 5, 10, \dots, 200\}$. For each specified link latency, we derive the median of the measured link latency as $\{LL_1, LL_5, LL_{10}, \dots, LL_{200}\}$.
3. Derive the link latency increase $\{LL_5 - LL_1, \dots, LL_{200} - LL_{195}\}$
4. Use the maximum increase of the link latency $\max(LL_5 - LL_1, \dots, LL_{200} - LL_{195})$ as the measure of link latency emulation accuracy.

Figure 11 demonstrates the link latency emulation accuracy. We can see that the link latency emulation accuracy does not monotonically improve as the timer resolution becomes finer. We achieve the best link latency accuracy of $22\mu\text{s}$ when the timer resolution is $33\mu\text{s}$. When the timer resolution is $10\mu\text{s}$, the link latency emulation accuracy is $57\mu\text{s}$. The reason is when HZ is larger, more timer interrupts are generated. When HZ exceeds a threshold, the processing of too many timeouts interrupts an application (such as Honeyd) too frequently and reduces the application’s performance. In our case, Honeyd cannot process the packets in a timely way. Therefore, when the timer resolution becomes too fine, the link latency emulation accuracy becomes worse instead of better.

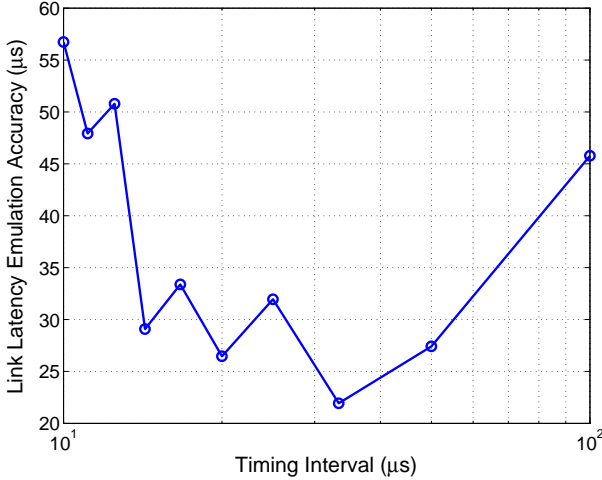


Figure 11: Link Emulation Accuracy (Linux 2.6.10 with Hight Resolution Timer)

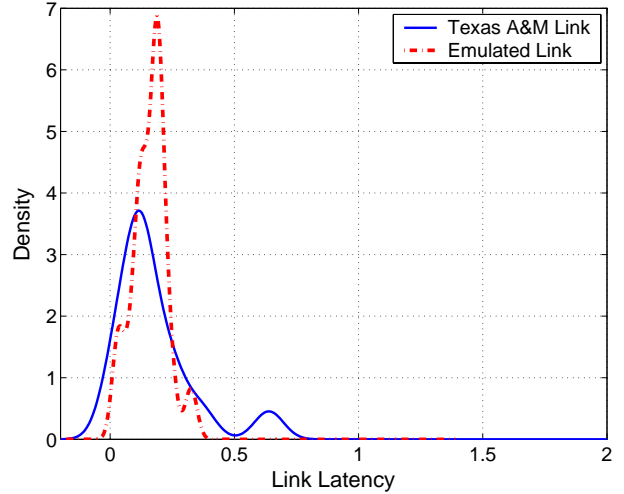


Figure 12: Emulated Network Link Latency Distribution v.s. Real Campus Network Latency Distribution

5.4 Effectiveness of Camouflaging

Recall that in order to camouflage Honeyd from the fingerprint attack as discussed in this paper, we need to make Honeyd capable of emulating real network link latency. For example, in the context of the Texas A&M University campus network, if Honeyd emulated link latency could have a distribution like the one in Figure 7, the attacker will not be able to use the fingerprint attack to tell the difference between emulated links and real links. Therefore, in order to camouflage Honeyd, we determine a link latency distribution for the virtual network that is as close as we can emulate to the real network’s link latency distribution. Then we randomly generate values from our approximate distribution to use when we configure Honeyd emulated link latencies.

From the above discussion, we know that if the PDF of the Honeyd emulated link latency can approach the real network link latency PDF in Figure 7, an attacker cannot determine whether a link is emulated by Honeyd or not. Figure 12 shows the emulated network link distribution we generate to match the real campus network latency distribution. We can see the similarity of the two distributions in Figure 12. In our experiments, we set HZ as 20,000 and adjust link latency in the Honeyd configuration. With few adjustments, we can achieve the effect shown in Figure 12. Actually, since we can adjust parameters HZ in the Linux kernel and link latency in the Honeyd configuration, we may produce a broad range of distribution patterns resembling real network link latency distributions.

Figure 13 shows the curve of detection rate and false alarm rate versus decision boundary. From Figure 13, we can derive Figure 14, which shows the ROC curve for the camouflaged Honeyd. The ROC curve almost follows a 45 degree diagonal, which represents the worst-case for detection, but the best case for camouflaging. That is, when the detection rate is big, the false alarm rate is big. When the detection rate is small, the false alarm rate is small too. For example, our experiments show that when the detection rate is 97%, the false alarm rate is as high as 90%. When the detection rate is 15%, the false alarm rate is around 27%. This shows that when we apply the above camouflaging approach to Honeyd, an attacker can no longer use the fingerprint attack discussed in this paper. This is an intuitive result considering the two overlapping distributions in Figure 12.

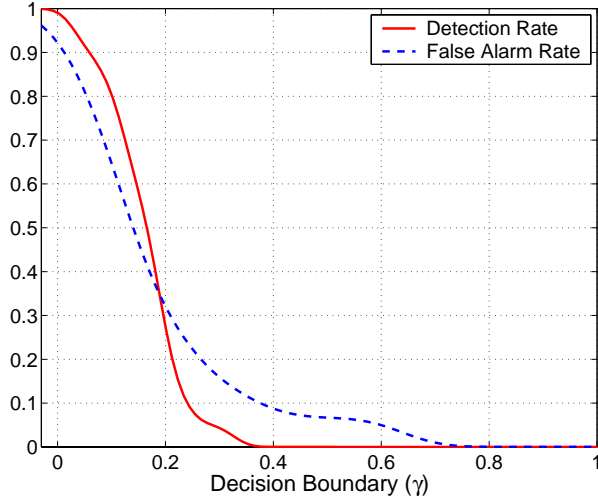


Figure 13: Detection Rate and False Alarm Rate vs Decision Boundary

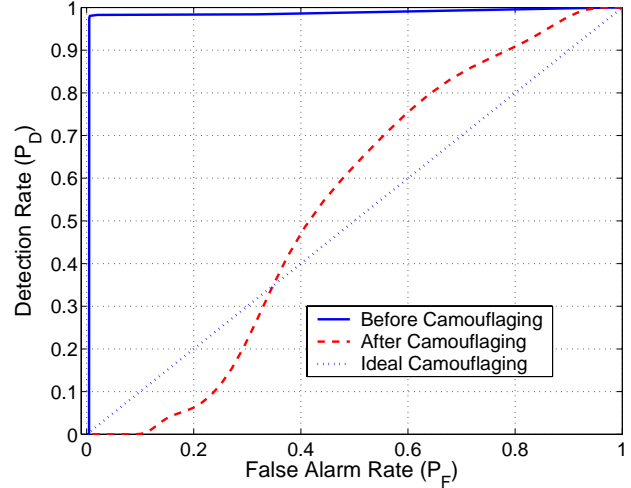


Figure 14: ROC for Camouflaged Honeyd

6 Conclusions

In this paper, we address issues related to detecting and “camouflaging” virtual honeypots. In general, virtual honeypots emulate characteristics of the target operating systems or networks. It is well known that such emulation is often a challenging task. An inappropriate emulation may expose the existence of the virtual honeypot. An attacker may find that the characteristics of a victim network cannot match believable parameters or those of its surrounding networks. We emphasize that great care has to be taken to hide virtual honeypots within their surroundings.

In particular, we find that Honeyd can neither configure nor emulate network link latencies at a resolution finer than 1ms. Our measured results demonstrated an emulation accuracy of 1ms or 10ms on a Linux platform, which is too large to accurately emulate a metropolitan network where Honeyd might be deployed. An attacker can build a profile of Honeyd link latency and remotely fingerprint Honeyd by measuring the link latency. The attacker can exploit measured differences in packet pairs using ICMP, TCP and UDP protocols to derive the link latency even if no well-known Internet services are being emulated by Honeyd. Our experiments show that the attacker may derive a high detection rate for Honeyd with a low false alarm rate.

To camouflage virtual honeypots, the key is to make them behave like the surrounding network. In our specific case, if the PDF of the Honeyd emulated link latency can approach the real network link latency PDF, an attacker cannot determine whether a link is emulated by Honeyd or not. To achieve a high fidelity emulation of link latency, we improve the OS timing accuracy by changing the Linux kernel parameter HZ. We show that an optimal timing interval exists and demonstrate an approach for choosing a value for the HZ parameter in order to get the best timing accuracy while maintaining system performance and thus greatly reduce the effectiveness of the fingerprint attack. For future work, we plan to rewrite the Honeyd code and port it to a real-time system such as TimeSys RTOS [32] or a Linux kernel with a higher resolution timer [1]. This study examined latencies to a μs resolution. Given increasing network and component speeds, we plan to explore emulation to nanosecond resolution. We will also compare the accuracy of link latency emulation by Honeyd under different implementations. We also believe that other vulnerabilities exist in emulation based virtual honeypots. We plan to do a thorough investigation of them and design compensating camouflaging approaches.

References

- [1] G. Anzinger. This is the home page for the high resolution timers project. <http://high-res-timers.sourceforge.net/>, 2005.
- [2] E. Balas. Sebek. <http://www.honeynet.org/tools/sebek/>, 2005.
- [3] C. Beardah. Kernel density estimation toolbox. <http://web.ccr.jussieu.fr/ccr/Documentation/Calcul/matlab5v11/docs/00053/05375.htm>, 1999.
- [4] M. Carson and D. Santay. Nist net - a linux-based network emulation tool. *Computer Communication Review*, 33(3), July 2003.
- [5] Computational Research Division, Lawrence Berkeley National Laboratory. Network characterization service (NCS). <http://www-didc.lbl.gov/NCS/>, 2005.
- [6] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly, 2005.
- [7] J. Corey. Local honeypot identification. <http://www.phrack.org/fakes/p62/p62-0x07.txt>, 2003.
- [8] J. Corey. Advanced honey pot identification. <http://www.phrack.org/fakes/p63/p63-0x09.txt>, 2004.
- [9] J. Dike. The user-mode linux kernel home page. <http://user-mode-linux.sourceforge.net/>, 2005.
- [10] M. Dornseif, T. Holz, and C. N. Klein. Nosebreak - attacking honeynets. In *Proceedings of IEEE Workshop on Information Assurance and Security*, June 2004.
- [11] A. B. Downey. Clink: a tool for estimating internet link characteristics. <http://allendowney.com/research/clink/>, 1999.
- [12] A. B. Downey. Using pathchar to estimate internet link characteristics. In *Proceedings of ACM SIGCOMM*, September 1999.
- [13] R. O. Duda and P. E. Hart. *Pattern Classification*. John Wiley & Sons, 2001.
- [14] X. Fu and B. Graham. Toolkits for camouflaging honeyd. http://students.cs.tamu.edu/xinwenfu/honeyd_tamu/, 2005.
- [15] honeynet.org. Honeynet project. <http://www.honeynet.org/index.html>, 2005.
- [16] honeynet.org. Honeywall cdrom - version roo. <http://www.honeynet.org/tools/cdrom/roo/manual/1-intro.html>, 2005.
- [17] V. Jacobson. Pathchar. <http://www.caida.org/tools/utilities/others/pathchar/>, 1997.
- [18] S. Keshav, C. Estan, H. Chan, and W. Chang. Project argus - network topology discovery, monitoring, history, and visualization. <http://www.cs.cornell.edu/boom/1999sp/projects/Network%20Topology/topology.html>, 1999.
- [19] T. Kohno, A. Broido, and kc claffy. Remote physical device fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2005.
- [20] B. Lowekamp, D. R. OHallaron, and T. R. Gross. Topology discovery for large ethernet networks. In *Proceedings of ACM SIGCOMM*, August 2001.
- [21] B. A. Mah. pchar: A tool for measuring internet path characteristics. <http://www.kitchenlab.org/www/bmah/Software/pchar/>, 2005.
- [22] openssl.org. Openssl - speed(1). <http://www.openssl.org/docs/apps/speed.html>, 2005.
- [23] phrack.org. Phrack. <http://www.phrack.org/>, 2005.
- [24] N. Provos. libevent - an event notification library. <http://www.monkey.org/~provos/libevent/>, 2004.
- [25] N. Provos. Developments of the honeyd virtual honeypot. <http://www.honeyd.org/>, 2005.
- [26] N. Provos. A virtual honeypot framework. In *Proceedings of USENIX Security Symposium*, August 2004.
- [27] B. Silverman. *Density Estimation for Statistics and Data Analysis, Monographs on Statistics and Applied Probability*. Chapman & Hall, London, 1986.
- [28] L. Spitzner. The value of honeypots, part one: Definitions and values of honeypots. <http://online.securityfocus.com/infocus/1492>, 2001.
- [29] W. R. Stevens. *TCP/IP Illustrated, Volume 1 The Protocols*. Addison-Wesley Professional, 1999.
- [30] tcpdump.org. tcpdump. <http://www.tcpdump.org/>, 2005.
- [31] TimeSys Corp. . Timesys. <http://www.timesys.com/>, 2005.
- [32] timesys.com. Timesys linux rtos sdk. http://www.timesys.com/index.cfm?bdy=linux_bdy_performance.cfm, 2005.
- [33] J.-M. Valin. Increasing hz (patch for hz > 1000). <http://kerneltrap.org/node/1766>, 2003.
- [34] vmware.com. Vmware workstation 5. http://www.vmware.com/products/desktop/ws_features.html, 2005.

Appendix A - Steps for Camouflaging Honeyd

1. Make libevent use “select” for the timing

Comment out the definitions of HAVE_EPOLL and HAVE_POLL in config.h, and then rebuild the libevent library. Under some systems, only one of the two definitions may appear. Refer to config.h at [14].

```
-----  
/* Define if your system supports the epoll system calls */  
//#define HAVE_EPOLL 1 comment out  
  
/* Define if you have the 'poll' function. */  
//#define HAVE_POLL 1  
-----
```

2. Make honeyd (1.0) support 1us link latency

Copy honeyd.c and lex.l at [14] to your honeyd source code directory, and rebuild your honeyd. The file config.sample provided at [14] is an example of using the new unit (almost the same file as the original one with some bugs fixed).

3. Patch Linux kernel (2.6.10) to support high resolution timing

Apply TAMU-2.6.10-hz.patch at [14] to Linux 2.6.10 kernel. By changing the parameters of “Z” and “USER_HZ” in the file /usr/src/linux2.6.10/include/asm-i386/param.h, you can alter OS’s timing resolution. We recommend a timing resolution of 50us. Refer to param.h at [14].

```
-----  
#define HZ                20000    /* Internal kernel timer frequency */  
#define USER_HZ          2000     /* .. some user interfaces are in "ticks" */  
-----
```