

# Camouflaging Virtual Honeypots

Xinwen Fu, Bryan Graham, Dan Cheng, Riccardo Bettati, and Wei Zhao  
Department of Computer Science, Texas A&M University  
College Station, TX 77843 - 3112  
E-mail: {xinwenfu, bryan, dcheng, bettati, zhao}@cs.tamu.edu

Technical Report 2005-7-3

## Abstract

*Honeypots are decoys designed to trap attackers. Once deployed, we can use honeypots to log an attacker's activities, analyze its behavior and design new approaches to defend against it. A virtual honeypot can emulate multiple honeypots on one physical machine, and so provide great flexibility in representing one or more networks of machines. In order to operate effectively, a honeypot needs to hide itself, analogously to an experienced sniper needing camouflage. In this paper, we address issues related with designing an appropriate "camouflage" for virtual honeypots, in particular for Honeyd, which can emulate any size of network on one or multiple physical machines. We found that an attacker may remotely fingerprint honeyd by measuring the link latency of the network links emulated by honeyd. We design a camouflaged honeyd by rewriting a small part of the honeyd toolkit code and by appropriately patching the operating system. Our experiments demonstrate the effectiveness of our approach to hide honeyd.*

## 1 Introduction

In this paper, we propose to camouflage virtual honeypots, in this case honeyd, from fingerprinting. 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 open Internet. First proposed in "The Cuckoo's Egg" by Cliff Stoll in 1990 [20], the concept of Honeypots was rapidly popularized and applied in the Internet security area. A honeypot was redefined by the Honeynets Project as "*an information system resource whose value lies in unauthorized or illicit use of that resource*" [18]. This definition encompasses every different manifestation of honeypots, ranging from large-scale computational frameworks to simple files. The unique characteristics of honeypots reside in the fact that there is no routine production traffic introduced by legitimate users on the honeypot, and therefore any activities taking place on it are inherently suspicious and likely a probe, attack or compromise. Consequently, the rate of false alarm will be greatly reduced compared with other IDS.

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 to a relatively low degree, such as TCP/IP stack, which effectively limits the scope of the attacker's actions and keeps the compromise risk minimal. On the other hand, high-interaction honeypots expose the whole system to an attacker, and even the operating system could be under the full control of the attacker.

Honeypots can also be categorized with respect to their implementation. A physical honeypot is a realistic machine with its own operating system and IP address, while a virtual honeypot is a machine with emulated system behavior and fictitious IP addresses. A virtual honeypot has a number of benefits: a single physical system can emulate more than one virtual honeypot, more than one type of operating system can be emulated, and different IP addresses can be assigned. Virtual honeypots require much less computation resources than a physical one, and they provide much more flexibility in simulating various operating systems.

*Honeyd* [16] is a low-interaction virtual honeypot. It is implemented to emulate computer systems at the network level. Specifically, *Honeyd* emulates the TCP/IP stack of an operating system in order to convince the attackers that they are exploring a “real” system. *Honeyd* is capable of responding to a network request according to the services the virtual honeypot is configured to provide. *Honeyd* modifies the response packets in order to match the configured operating system’s network behavior. Besides the system activities, *Honeyd* is also simulating the network behavior, including the 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 discuss issues on designing an appropriate “camouflage” for a virtual honeypot. In particular, we found that *honeyd* leaves opportunities for attackers to identify and then intentionally avoid it. We will investigate how to camouflage *honeyd* effectively. Camouflage is widely used in nature as a means to evade detection. Animals use camouflage in order to disguise and hide themselves from something or someone in plain sight. A chameleon, for instance, changes color to blend in with its environment. Soldiers often wear special camouflage clothing to become less visible during war. We propose to camouflage our cyberspace infrastructure such as honeypots from cyber attacks by designing proper technologies and developing the corresponding theories.

Although honeypots at first sight appear to be naturally camouflaged (by virtue of their function), they often are not. We will show in this paper that virtual honeypots may demonstrate distinct temporal signatures that make them easily detectable. Hence, the need for temporal “camouflage” of such honeypots is pressing. Our contributions can be summarized as follows:

1. We found that virtual honeypots may display a clear temporal signature: by examining link latency within a network emulated by *honeyd*, an attacker can discover whether the corresponding links connecting routers are virtual ones. This stems from the fact that honeypots are not typically designed to emulate the temporal behavior of node at high fidelity. *Honeyd* is not designed to support arbitrary link latency resolution. Ordinary OSes, for which *honeyd* is designed, cannot support the necessary timing resolution either. As a result, the virtual honeypot displays a significantly different temporal signature. For example, the virtual links are statistically very different from the surrounding physical network links.
2. We proposed means to temporally camouflage virtual honeypots such as *honeyd*. To illustrate the approach, we designed a camouflaged *honeyd*, which supports a link latency in the order of one microsecond instead of the original one millisecond default. Correspondingly, we generated a new patch to Linux 2.6.10 in order to support the new *honeyd* implementation. Alternative camouflaging schemes are also discussed.
3. We developed the actual toolkits for camouflaging *honeyd* and rewrote a minimum part of the original *honeyd* toolkit code. These toolkits and instructions can be found at [9]. We give the set of 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 give an overview of how an attacker may discover *honeyd* by measuring the link latency and the related issues. We discuss those issues in the following sections. We introduce how an attacker derives the link latency in Section 5, discuss how the attacker classifies a suspect link as either a virtual link or a physical link in Section 6 and evaluate the attack’s threat in Section 7. In Section 8,

we discuss approaches to hide honeyd from the attacker. We conclude this paper and discuss the future work in Section 9.

## 2 Related Work

Honeyd is 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) [7] 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 [26] 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 are virtual high interaction honeypots since they run full-fledged services on one machine.

Honeyd [16] 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 the attackers that they are exploring a “real” system. Honeyd is capable of responding to a network request according to the services the virtual honeypot is configured to provide. It is also trying to personalize the response packets in order to match the configured operating system network behavior. In addition to system activities, Honeyd is also simulating the network behavior, including the network topology and link characteristics such as latency and packet loss.

The Honeynet [10] is a research oriented, high-interaction honeypot, which is the research effort 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 server which collects the data from the honeypots. The server normally runs on the Honeywall gateway, which both captures and controls all inbound and outbound honeynet activity [11].

Dornseif *et al.* [8] presented an approach for attacking *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 [14], a Hacker magazine, about honeypot identification. One is entitled “Local Honeyd Identification” [5] 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 [6] 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.

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.

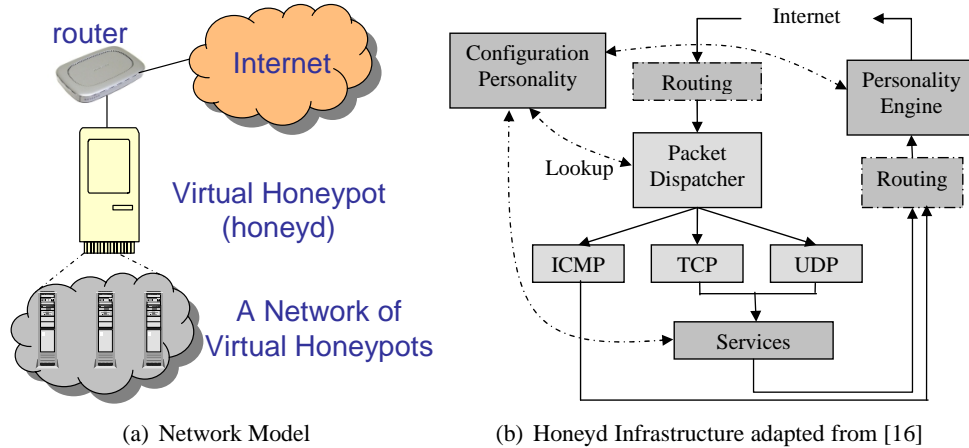


Figure 1. Models

### 3 Models

In this section, we introduce the network model and threat model (the capabilities of the attacker) used in this paper.

#### 3.1 Network Model

Honeyd is a decoy network 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 the threats we face. Compared with physical honeypots, virtual honeypots are more resource efficient, easy to build, and simple to maintain since a virtual honeypot may only need one single physical machine to emulate multiple honeypot servers. 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 captures a variety of usage of high-interaction virtual honeypots such as User-Mode Linux (UML) and low-interaction virtual honeypots such as honeyd for the objective of intrusion detection. In this paper, we will focus on analyzing a low-interaction virtual honeypot, honeyd.

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* [15]. 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 finding whether a network is a virtual network created by honeyd or a real physical network.

1. The attacker is an active one. It can inject probing traffic into the Internet remotely as normal users do. It can adjust the frequency and intensity of the probing traffic in order to prevent itself from being discovered.
2. The attacker can utilize tools to discover the network topology [12]. Then the attacker can choose one suspect link to test whether it is a real link or a virtual one.
3. The attacker knows the emulation strategies and algorithms of virtual honeypots. This assumption is commonly used in security research.

## 4 Recognizing Honeyd

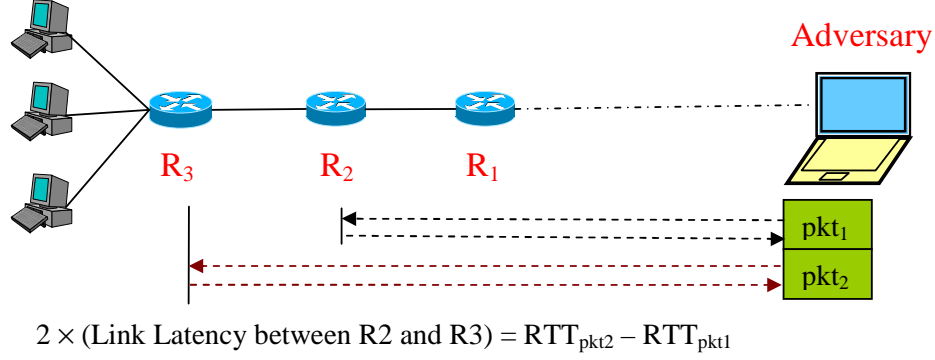
In this section, we first give an overview on how an attacker may fingerprint a virtual network emulated by honeyd. We define the problem formally and introduce issues related to the fingerprinting attack.

### 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, a large portion of events in the system must be triggered by 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 [4].

Without loss of generality, we only discuss the popular x86 PC platform in the remaining parts of this paper. This means 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 be always around a multiple of 10ms or 1ms. There are two issues related to this observation: (1) a link latency with multiples of 10ms or 1ms can be a fingerprint of honeyd, which creates the virtual network; (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 timing signature of the virtual network emulated by honeyd. Thus, if an attacker can recognize that a link exhibits such a signature, it 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 physical link or one emulated by honeyd?*



**Figure 2. Principle for recognizing Honeyd**

#### 4.2 Issues Related to Recognizing Virtual Honeypots

From the discussion above, we know the success of recognizing honeyd lies in accurately measuring link latency within the virtual network. Figure 2 demonstrates the principle method of deriving link latency<sup>1</sup>. To find the link latency of a network link connecting routers  $R_2$  and  $R_3$ , an attacker can send a pair of packets  $\langle \text{pkt}_1, \text{pkt}_2 \rangle$  of minimal size.  $\text{pkt}_1$  is addressed to router  $R_2$  while  $\text{pkt}_2$  is addressed to router  $R_3$ . Intuitively,

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

$$= \text{RTT}_{\text{pkt}_2} - \text{RTT}_{\text{pkt}_1} \quad (2)$$

where  $\text{RTT}_{\text{pkt}_n}$  is the Round Trip Time (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.)

This approach is robust and insensitive to network noise. When a packet traverses the Internet, the intermediate routers may disturb the timing of the packet. This forms the network noise, which disturbs the measurement of RTT. The approach in Figure 2 uses packet pairs of minimum size that are sent out sequentially. Their timings and thus the measurements of RTT will very likely experience the same noise from the Internet. Since the link latency is the subtraction of the two RTTs, the noise is cancelled each other out and the approach is not sensitive to network noise.

Therefore, 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 thus the link latency?
2. How can an attacker recognize a virtual network using the measured link latency?
3. How effective is this class of attack?
4. How can we hide honeyd from this form of fingerprinting attacks?

We will answer these questions in the following sections.

<sup>1</sup>We use a virtual network topology similar to the one defined in *config.sample*, shipped with honeyd 1.0 [17].

## 5 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 thus 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 generate 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.

### 5.1 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 [19].

### 5.2 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 obtained a reply packet and can calculate the RTT.

To correlate sent and received TCP packets, the attacker can utilize the TCP sequence number and acknowledgement number. Thus, 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.

### 5.3 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 [19]. Thus, the attacker can send a bogus UDP packet to the victim router. The bogus UDP packet uses a port not in use on the victim router, which will then 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.

Finally, it must be pointed out that any one 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 effectively usable for attack classification, they must at least engage the attacker in some packet exchange, at which point some RTT information can be gathered.

## 6 Recognition of Honeyd

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. In this section, we answer the second question from Section 4.2: “How can an attacker recognize a virtual network using the measured link latency?”

### 6.1 Phases in Recognition of Virtual Honeypots

In order to recognize a virtual link the attacker builds a profile in terms of link latency of a honeyd link in advance and then compares the suspect link with the profile. If the link latency of the suspect link is found to be statistically equal to the profile, the attacker classifies the suspect link as a virtual link, otherwise a physical link. Therefore, an attacker uses a two phase approach:

1. Offline training phase: (a) the attacker sets up honeyd simulating a virtual network. The virtual network has the same topology as the victim network that the attacker targets. We assumed that the attacker can use tools to derive the topology of a network; (b) the attacker collects a series of link latency data of a specific virtual link and builds its profile.
2. Online recognition phase: (a) the attacker collects a series of corresponding link latency data from the real network and builds its profile; (b) based on the virtual link profile and the suspect link profile, the attacker chooses a decision rule to decide if the suspect link is virtual or physical.

### 6.2 T-test as Decision Rule

The decision whether a suspect link is virtual or physical is binary, and can therefore be formulated as a hypothesis testing problem. In our scenario, the hypothesis can be that “the suspect link is a virtual link”, and an attacker needs to determine the probability that this hypothesis is true. If the hypothesis testing tells us that the probability of this hypothesis being true is high enough, the attacker cannot reject the hypothesis and will classify the suspect link as a virtual link emulated by honeyd. On the other hand, if the hypothesis testing tells us that the probability of this hypothesis being true is sufficiently low, the attacker rejects the hypothesis and classifies the suspect link as a physical link. In other words, the question of interest is simplified into two competing hypotheses between which we make a choice: the null hypothesis that “the suspect link is a virtual link”, versus the alternative hypothesis that “the suspect link is a physical link”.

In this paper, we use t-test, which uses statistics to determine the probability that a given hypothesis is true. In the following we describe how an attacker applies t-test to determine if a suspect link is physical or virtual [21]. The description follows standard practice for hypothesis testing.

1. *Formulate a null hypothesis (denoted  $H_0$ ) and the alternative hypothesis (denoted  $H_1$ ).* As discussed above, we define these two hypotheses as follows:

$$\begin{aligned} H_0 &: \text{the suspect link is a virtual link} \\ H_1 &: \text{the suspect link is a physical link} \end{aligned} \tag{3}$$

2. *Identify a test statistic that can be used to determine the probability that the null hypothesis is true.*

In this paper, we assume that an attacker uses  $t$  statistic<sup>2</sup>. T-test is used to determine if the means of two samples are equal. Therefore, the original problem of recognizing a virtual link is transformed to the problem of determining if the means of the two link latency samples are equal or not.

Suppose we have two link latency samples  $X = \{X_1, \dots, X_n\}$  of size  $n$  and  $Y = \{Y_1, \dots, Y_m\}$  of size  $m$ .  $X$  is the sample corresponding to the profile of the virtual link while  $Y$  is the sample corresponding to the profile of the suspect link. The two samples can have different variance. The following  $t$  statistic is used to test the hypothesis in (3).

$$t = \frac{\bar{X} - \bar{Y}}{\sqrt{\frac{s_x^2}{n} + \frac{s_y^2}{m}}} \quad (4)$$

$$df = \frac{(s_x^2/n + s_y^2/m)^2}{\frac{(s_x^2/n)^2}{n-1} + \frac{(s_y^2/m)^2}{m-1}} \quad (5)$$

where  $\bar{X}$  and  $\bar{Y}$  are sample means of the corresponding samples,  $df$  is the degree of freedom of the  $t$  test and  $s_x^2, s_y^2$  are the sample variances.

$$\bar{X} = \frac{\sum_{i=1}^n X_i}{n} \quad (6)$$

$$s_x^2 = \frac{\sum_{i=1}^n (X_i - \bar{X})^2}{n-1} \quad (7)$$

3. *Compute the P-value, which is the probability that a test statistic at least as significant as the one observed would be obtained, assuming that the null hypothesis were true.* The smaller the P-value, the more unlikely the null hypothesis is true, and the stronger the evidence against the null hypothesis. The P-value for the t-test can be derived from the sampling distribution of  $t$  statistic at the corresponding degree of freedom. The sampling distribution of  $t$  statistic is often made into tables for references.
4. *Compare the P-value to an acceptable significance value  $\alpha$ .* If  $P \leq \alpha$ , the observed effect is deemed as statistically significant, and then the null hypothesis  $H_0$  is rejected. That is, an attacker would classify the suspect link as a physical one. Otherwise, the null hypothesis cannot be rejected. An attacker then classifies the suspect link as a virtual link. Obviously, the significance value  $\alpha$  is an acceptance threshold. In this paper, we choose  $\alpha = 0.01$ .

### 6.3 Evaluation Metrics

In this paper, in order to measure how well an attacker can recognize a virtual link, we define *detection rate*  $P_D$  as the probability that the null hypothesis cannot be rejected. In our case, the detection rate  $P_D$  is the probability that an attacker correctly recognizes a virtual link when the suspect link is indeed a virtual link. We use  $P_D$  as the metric to evaluate the t-test based recognition approach. We define *rejection rate* as the probability that the null hypothesis is ruled out. That is, rejection rate is the probability that an attacker recognizes a physical link when the suspect link is a physical link. We define *false alarm rate* as the probability when an attacker incorrectly identifies a physical link as a virtual link emulated by honeyd. Thus, we can see that

$$\text{false alarm rate} = 1 - \text{rejection rate} \quad (8)$$

---

<sup>2</sup>T-test has different versions under difference cases/assumptions. We choose the case of T-test with different variances in two link latency sample

## 7 Evaluation of the Fingerprinting Attack

In this section, we evaluate the effectiveness of the attack described above. For this, we use a testbed and measure the accuracy at which an attacker who applies t-test can distinguish a honeyd emulated virtual link from a physical link. We selected honeyd for these experiments. We expect, however, that other virtual honeypot implementations would lead to similar results. So we will answer the third question from Section 4.2: “How effective is this class of attack?”

### 7.1 Experiment Setup

Figure 3 illustrates our experiment setup. We deploy a virtual honeyd network in our laboratory<sup>3</sup>. The topology corresponds to a routing path from the Department of Computer Science to the university homepage. The objective of the adversary is to decide whether the link between  $R_2$  and  $R_3$  is a virtual link created by honeyd or a physical link connecting to the university homepage. Between the honeyd virtual network and the attacker’s machines, we setup a machine with NISTnet [3] to emulate the influence from cross traffic or intermediate networks. The attacker can get access to the Internet from our lab and knows the topology of the campus network. This type of setup represents a broad range of use for honeyd within a metropolitan network (MAN) including company networks.

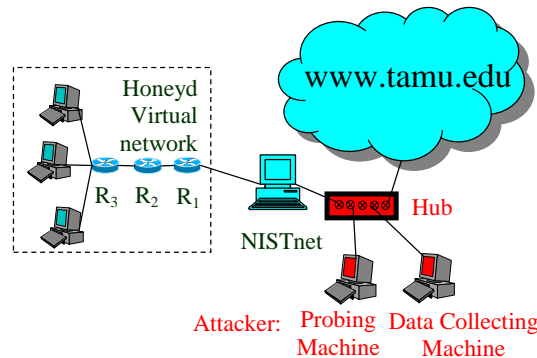


Figure 3. Experiment Setup

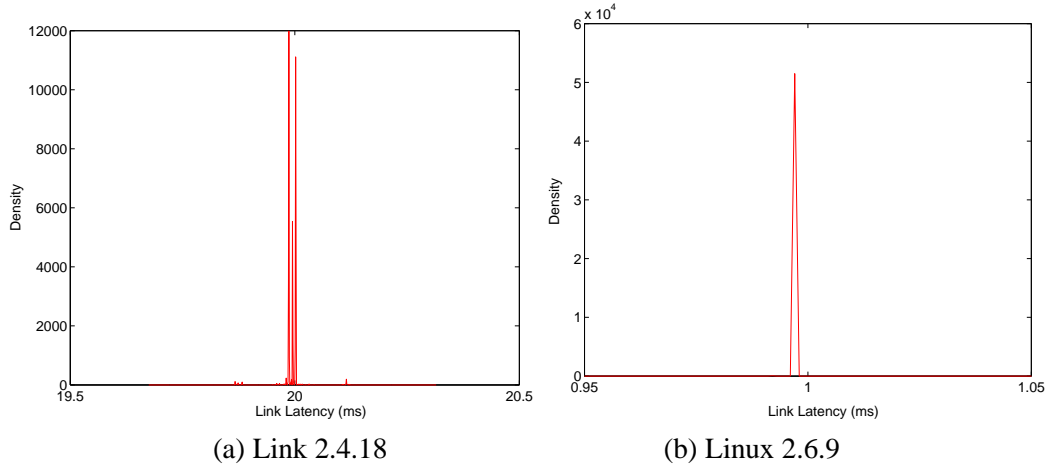
We use as a honeyd virtual network hosting machine a Dell Dimension 8400 Intel desktop configured with Pentium 4 processor 640 supporting Hyper-Threading (HT) Technology (3.20GHz CPU featuring 800 MHz front side bus - FSB). The attacker controls two machines: the probing machine generating the TCP/UDP/Ping probing traffic and the data collecting machine. The attacker installs tcpdump [22] on the data collecting machine to dump traffic on the hub, which is connected to 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.

### 7.2 Failure of Current Honeyd Implementation

Figure 4 demonstrates the probability density function (PDF) of link latency when an attacker uses the TCP probing approach. 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. Moreover, the PDF curve is very compact. This means that a packet pair based probing approach in this case generates very accurate measurement of link latency.

Figure 5 shows the results of t-test to recognize a virtual link emulated by honeyd. We have the following observations:

<sup>3</sup>Because of the university administration issues, we cannot deploy honeyd at a campus scale



**Figure 4. PDFs of Link Latency**

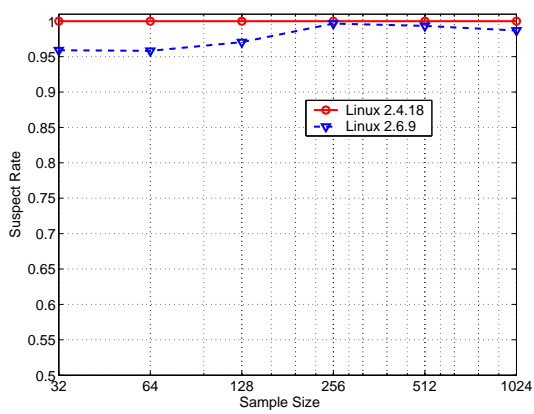
1. An attacker can recognize a virtual link emulated by honeyd very effectively by using a TCP, UDP or Ping based packet pair probing approach. We can see that for all these approaches, the attacker achieves a detection rate of 100% if the link is a virtual link. The attacker achieves a rejection rate of 100% if the link is a physical link. A rejection rate of 100% implies a false alarm rate of 0%.
2. An attacker can recognize a virtual link very efficiently by using the TCP, UDP or Ping based packet pair probing approach. For a honeyd running on either Linux 2.4.18 or Linux 2.6.9, an attacker achieves a detection rate of over 95% with 32 packet pairs under each of these three approaches. The attacker achieves a rejection rate of 100% (false alarm of 0%) when the sample size is 32.

## 8 Camouflaging Honeyd

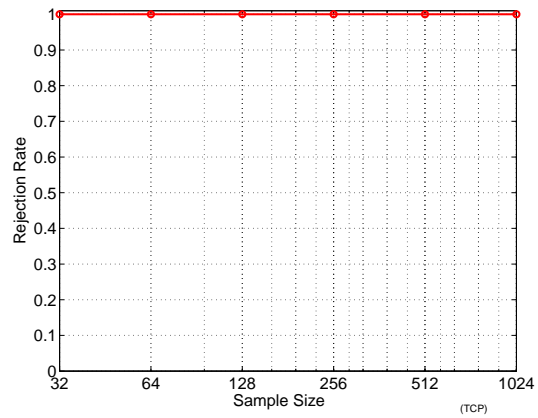
In the previous section, we provided evidence that virtual honeypot implementations like honeyd are highly vulnerable to simple timing attacks. In this section, we want to identify means to protect virtual honeypots against these types of attacks. Camouflaging is widely used by animals in order to hide themselves within their environment and avoid being discovered and attacked. We perceive camouflaging also as a very general way of providing a variety of security services in cyberspace. For example, encryption can be viewed as one kind of camouflaging to provide confidentiality. Below we discuss how to camouflage honeyd into its surrounding networks and prevent its discovery. So we will answer the fourth question in Section 4.2: How can we hide honeyd from this form of fingerprinting attacks?

### 8.1 Overview of the Principle

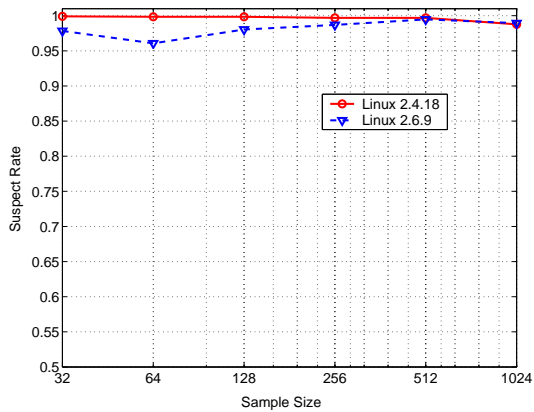
The key reason for the failure of the current honeyd implementation (and likely other honeypot implementations as well) is the low fidelity of emulation (in the time domain) of the system components by the honeypot. These components may represent users, applications and servers in the case of physical honeypots, in addition to hardware resources in the case of virtual honeypots. The processing delay caused by these components is emulated by timers in the honeypot. When these timers are either carelessly defined in the 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 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 under a t-test attack.



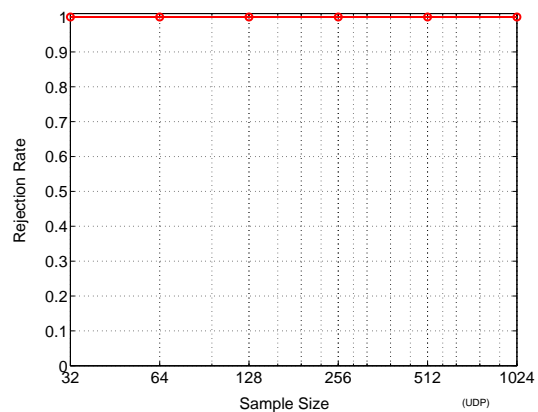
(a) TCP Detection Rate



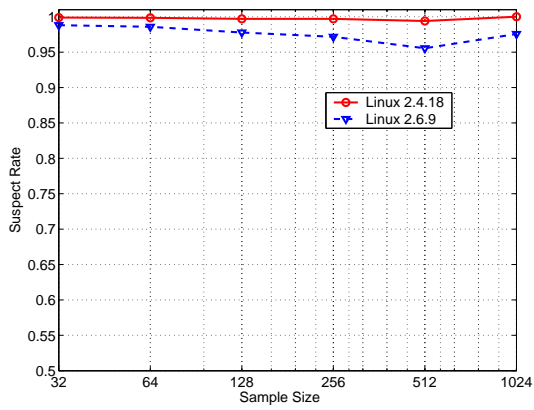
(b) TCP Rejection Rate



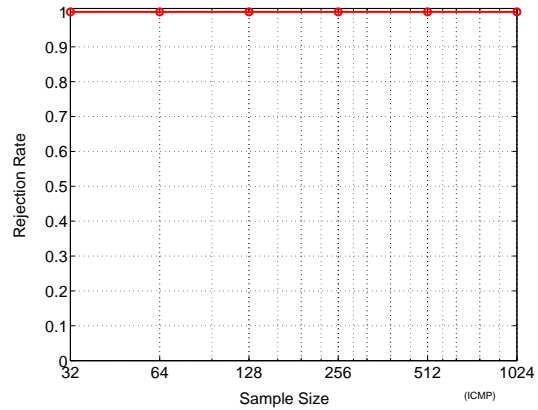
(c) UDP Detection Rate



(d) UDP Rejection Rate



(e) Ping Detection Rate



(f) Ping Rejection Rate

Figure 5. Evaluation by t-test

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 high-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 meaningful profile for honeyd. We camouflage honeyd in the following ways:

1. We change the honeyd code to make it support the timing resolution of microseconds  $\mu s$ . 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 [23] 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  $HZ$ . We ported the HZ patch [25] to kernel 2.6.10. In this way, we don’t need to change libevent API at all while we may sacrifice some CPU efficiency. However, since the physical machine running honeyd should not run any other useful applications, we don’t lose much here.

Therefore, honeyd can emulate link latency comparable to that of its surrounding networks, and an attacker cannot determine whether a link is emulated by honeyd or not. We discuss the overhead and achievable timing accuracy by this approach in the following.

## 8.2 System Performance under Different HZ Settings

Figure 6 demonstrates the CPU performance in terms of the timer timeout accuracy (i.e.,  $1/HZ$ ). We utilize the “speed” function in OpenSSL [13] to benchmark the system’s performance under different HZ settings. OpenSSL tests how many operations it can perform in a given time. Figure 6 (a) shows how many sign/verify operations with a 1024 bit RSA public key pair it can perform in 10s, while Figure 6 (b) shows the performance degradation with the performance at  $HZ=100$  (timeout accuracy 10ms in Linux 2.4) defined as 100%.

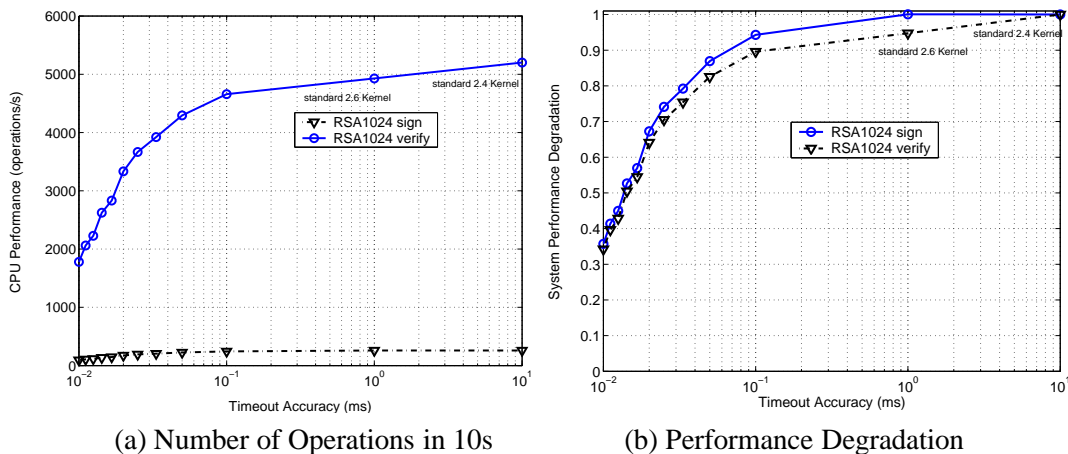


Figure 6. CPU Performance

We have the following observations from Figure 6:

1. When HZ increases (timeout accuracy increases), the system performance decreases. From 6 (b), we can see that the performance degrades to 34% of the best performance (specified as 100%) when the timeout accuracy is  $10\mu s$  (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.

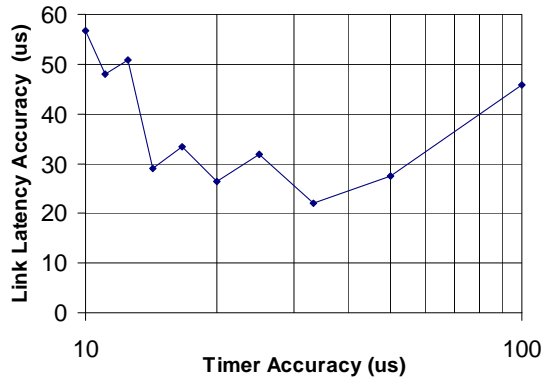
Honeyd is designed to be able to process a large amount of network load. Thus we need to carefully select a HZ. We base our selection of HZ on the accuracy of link latency emulation described below.

### 8.3 Accuracy of Link Latency Emulation

Honeyd provides a configuration file for us to specify a network topology with configurable parameters including link latency. When we deploy honeyd, we prefer that it can 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 accuracy in the following pessimistic way.

1. Set the HZ to a specified value, e.g., 20,000.
2. Increase the link latency of a link (such as the link between  $R_2$  and  $R_3$  in Figure 2) at a step of  $5\mu s$  and measure the corresponding link latency. In our experiments, the link latencies we specified are  $\{1, 5, 10, \dots, 200\}$ . 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 maximal increase of the link latency  $\max(LL_5 - LL_1, \dots, LL_{200} - LL_{195})$  as the measure of link latency emulation accuracy.

Since we use the the maximum increase of the link latency as the emulation accuracy, we are considering a worst case.



**Figure 7. Link Emulation Accuracy**

Figure 7 illustrates the link latency accuracy in terms of the timing accuracy (1/HZ) on Linux. We can see that the link latency emulation accuracy does not monotonically increase with the increase of timer accuracy. We achieve the best link latency accuracy of  $22\mu s$  when the timer accuracy is  $33\mu s$ . When the timer accuracy is  $10\mu s$ , the link latency emulation accuracy is  $57\mu s$ . The reason is when HZ has a bigger value, it generates more

interrupts. When HZ reaches a threshold and gets too large, 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 accuracy increases beyond a threshold, the link latency emulation accuracy becomes worse instead of better.

We recommend deploying honeyd with HZ set at 20,000. Thus the Linux system performance will only degrade maximally around 10% (refer to Figure 6) with respect to the performance of the standard 2.6 kernel while achieving a decent link latency emulation accuracy of  $27\mu s$ .

Therefore, we camouflage virtual honeypots by achieving good accuracy of link latency emulation. There is no way that an adversary can build a profile for such a virtual honeypot and apply the class of fingerprinting attacks described in Section 4.

## 9 Conclusions

In this paper, we propose camouflaging virtual honeypots within their background network. 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 since the 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 for hiding virtual honeypots within its surrounding.

In particular, we found that honeyd cannot emulate network link latency as configured. This results in an emulation accuracy of 1ms or 10ms on a Linux platform, which is too big for a metropolitan network where honeyd might be deployed. The attacker can build a profile of honeyd link latency and remotely fingerprint honeyd by measuring the link latency. We suggested that an attacker could measure the link latency with appropriate packet pairs. The attacker can exploit ICMP, TCP and UDP protocols to derive the link latency even if no well-known Internet services are emulated by honeyd. Our experiments show that the attacker may derive a high recognition rate for honeyd. While the timing attack described in this paper relies on training data, training is not inherently necessary. In fact, training is particularly suited for situations where the signatures cannot be provided beforehand. In this particular case however, we understand what mechanisms (primarily low timer resolution due to periodic timer event dispatching at the OS level) cause the timing signatures of the type of virtual honeypots under consideration. The observed time latency data should therefore show "quantitative" effects due to the periodic timer event dispatching. Even in very noisy environment, these periodicities can be detected and measured using wavelet analysis and entropy measurements. In comparison to training based methods, such a scheme may require careful calibration in order to determine the threshold.

To camouflage honeyd, we improve OS's timing accuracy by changing the kernel parameter of HZ. We show that an optimal HZ exists and give guidelines about how to choose an appropriate HZ in order to get the best timing accuracy. For future work, we plan to rewrite the honeyd code and port it to a real-time system such as TimeSys RTOS [24] or a linux kernel with a high resolution timer [1]. We will compare the accuracy of link latency emulation by honeyd under different implementations. We also believe that other vulnerabilities exist in emulation or emulation based virtual honeypots. We plan to do a thorough investigation of them and design the corresponding 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] M. Carson and D. Santay. Nist net - a linux-based network emulation tool. *Computer Communication Review*, 33, July 2003.

- [4] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers, Third Edition*. O'Reilly, 2005.
- [5] J. Corey. Local honeypot identification. <http://www.phrack.org/fakes/p62/p62-0x07.txt>, 2003.
- [6] J. Corey. Advanced honey pot identification. <http://www.phrack.org/fakes/p63/p63-0x09.txt>, 2004.
- [7] J. Dike. The user-mode linux kernel home page. <http://user-mode-linux.sourceforge.net/>, 2005.
- [8] M. Dornseif, T. Holz, and C. N. Klein. Nosebreak - attacking honeynets. In *IEEE Workshop on Information Assurance and Security*, 2004.
- [9] X. Fu and B. Graham. Toolkits for camouflaging honeyd. [http://students.cs.tamu.edu/xinwenfu/honeyd\\_tamu/](http://students.cs.tamu.edu/xinwenfu/honeyd_tamu/), 2005.
- [10] honeynet.org. Honeyd project. <http://www.honeynet.org/index.html>, 2005.
- [11] honeynet.org. Honeywall cdrom - version roo. <http://www.honeynet.org/tools/cdrom/roo/manual/1-intro.html>, 2005.
- [12] B. Lowekamp, D. R. OHallaron, and T. R. Gross. Topology discovery for large ethernet networks. In *SIGCOMM*, 2001.
- [13] openssl.org. Openssl - speed(1). <http://www.openssl.org/docs/apps/speed.html>, 2005.
- [14] phrack.org. Phrack. <http://www.phrack.org/>, 2005.
- [15] N. Provos. libevent - an event notification library. <http://www.monkey.org/~provos/libevent/>, 2004.
- [16] N. Provos. A virtual honeypot framework. In *USENIX Security Symposium*, 2004.
- [17] N. Provos. Developments of the honeyd virtual honeypot. <http://www.honeyd.org/>, 2005.
- [18] L. Spitzner. The value of honeypots, part one: Definitions and values of honeypots. <http://online.securityfocus.com/infocus/1492>, 2001.
- [19] W. R. Stevens. *TCP/IP Illustrated, Volume 1 The Protocols*. Addison-Wesley Professional, 1999.
- [20] C. Stoll. *Cuckoo's Egg*. Pocket, 1990.
- [21] A. Studenmund. *Using Econometrics: A Practical Guide (4th Edition)*. Addison Wesley, 2000.
- [22] tcpdump.org. tcpdump. <http://www.tcpdump.org/>, 2005.
- [23] TimeSys Corp. . Timesys. <http://www.timesys.com/>, 2005.
- [24] timesys.com. Timesys linux rtos sdk. [http://www.timesys.com/index.cfm?bdy=linux\\_bdy\\_performance.cfm](http://www.timesys.com/index.cfm?bdy=linux_bdy_performance.cfm), 2005.
- [25] J.-M. Valin. Increasing hz (patch for hz > 1000). <http://kerneltrap.org/node/1766>, 2003.
- [26] vmware.com. Vmware workstation 5. [http://www.vmware.com/products/desktop/ws\\_features.html](http://www.vmware.com/products/desktop/ws_features.html), 2005.

## Appendix A - Steps Camouflaging Honeyd

### 1. Make your libevent use “select” for the timing

Comment out the definitions of HAVE\_EPOLL and HAVE\_POLL in config.h, and then remake your libevent library. Under some systems, maybe only one of the two definitions appears. Refer to config.h at [9].

```
-----
/* 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 your honeyd (1.0) support 1us link latency

Copy honeyd.c and lex.l at [9] to your honeyd source code directory, and rebuild your honeyd. The file config.sample provided here is an example of using the new unit (almost the same file as the original one with some bugs fixed).

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

Apply TAMU-2.6.10-hz.patch at [9] 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 [9].

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