Remote physical device fingerprinting

Tadayoshi Kohno
CSE Department, UC San Diego
tkohno@cs.ucsd.edu

Andre Broido
CAIDA, UC San Diego
broido@caida.org

kc claffy
CAIDA, UC San Diego
kc@caida.org

Abstract

We introduce the area of remote physical device fingerprinting*, or fingerprinting a physical device, as opposed to an operating system or class of devices, remotely, and without the fingerprinted device's known cooperation. We accomplish this goal by exploiting small, microscopic deviations in device hardware: clock skews. Our techniques do not require any modification to the fingerprinted devices. Our techniques report consistent measurements when the measurer is thousands of miles, multiple hops, and tens of milliseconds away from the fingerprinted device, and when the fingerprinted device is connected to the Internet from different locations and via different access technologies. Further, one can apply our passive and semi-passive techniques when the fingerprinted device is behind a NAT or firewall, and also when the device's system time is maintained via NTP or SNTP. One can use our techniques to obtain information about whether two devices on the Internet, possibly shifted in time or IP addresses, are actually the same physical device. Example applications include: computer forensics; tracking, with some probability, a physical device as it connects to the Internet from different public access points; counting the number of devices behind a NAT even when the devices use constant or random IP IDs; remotely probing a block of addresses to determine if the addresses correspond to virtual hosts, e.g., as part of a virtual honeynet; and unanonymizing anonymized network traces.*

1 Introduction

There are now a number of powerful techniques for *remote operating system fingerprinting*, i.e., techniques for remotely determining the operating systems of devices on the Internet [2, 3, 5, 27]. We push this idea further and introduce the notion of *remote physical device fingerprinting*, or remotely fingerprinting a physical device, as opposed to an operating system or class of devices, without the fingerprinted device's known cooperation. We accomplish this goal to varying degrees of precision by exploiting microscopic deviations in device hardware: clock skews.

CLASSES OF FINGERPRINTING TECHNIQUES. We consider three main classes of remote physical device fingerprinting techniques: passive, active, and semi-passive. The first two have standard definitions — to *apply* a *passive* fingerprinting technique, the *fingerprinter* (*measurer*, *attacker*, *adversary*) must be able to *observe* traffic from the device (the *fingerprintee*) that the attacker wishes to fingerprint, whereas to apply an *active* fingerprinting technique, the fingerprinter must have the ability to *initiate* connections to the fingerprintee. Our third class of techniques, which we refer to as *semi-passive* fingerprinting techniques, assumes that *after* the fingerprintee *initiates* a connection, the fingerprinter has the ability to interact with the fingerprintee over that connection; e.g., the fingerprinter is a website with which the device is communicating, or is an ISP in the middle capable of modifying packets en route. Each class of techniques has its own advantages and disadvantages. For example, passive techniques will be completely undetectable to the fingerprinted device, passive and semi-passive techniques can be applied even if the fingerprinted device is behind a NAT or firewall, and semi-passive and active techniques can potentially be applied over longer periods of time; e.g., after a laptop connects to a website and the connection terminates, the website can still continue to run active measurements.

METHODOLOGY. For all our methods, we stress that the fingerprinter does not require any modification to or cooperation from the fingerprintee; e.g., we tested our techniques with default Red Hat 9.0, Debian 3.0, FreeBSD 5.2.1, OpenBSD 3.5, OS X 10.3.5 Panther, Windows XP SP2, and Windows for Pocket PC 2002 installations.[1] In Table 1 we summarize our preferred methods for fingerprinting the most popular operating systems.

Our preferred passive and semi-passive techniques exploit the fact that most modern TCP stacks implement the

[1] Our techniques work for the default installs of other versions of these operating systems; here we just mention the most recent stable versions of the operating systems that we analyzed.

Technique and section	Class	NTP	Red Hat 9.0	OS X Panther	Windows XP
TCP timestamps, Section 3	passive	Yes	Yes	Yes	No
TCP timestamps, Section 3	semi-passive	Yes	Yes	Yes	Yes
ICMP tstamp requests, Section 4	active	No	Yes	No	Yes

Table 1. This table summarizes our main clock skew-based physical device fingerprinting techniques. A "Yes" in the NTP column means that one can use the attack regardless of whether the fingerprintee maintains its system time with NTP [19]. One can use passive and semi-passive techniques when the fingerprintee is behind a NAT or current generation firewall.

TCP timestamps option from RFC 1323 [13] whereby, for performance purposes, each party in a TCP flow includes information about its perception of time in each outgoing packet. A fingerprinter can use the information contained within the TCP headers to estimate a device's clock skew and thereby fingerprint a physical device. We stress that one can use our TCP timestamps-based method even when the fingerprintee's system time is maintained via NTP [19]. While most modern operating systems enable the TCP timestamps option by default, Windows 2000 and XP machines do not. Therefore, we developed a trick, which involves an intentional violation of RFC 1323 on the part of a semi-passive or active adversary, to convince Microsoft Windows 2000 and XP machines to use the TCP timestamps option in Windows-initiated flows. In addition to our TCP timestamps-based approach, we consider passive fingerprinting techniques that exploit the difference in time between how often other periodic activities are supposed to occur and how often they actually occur, and we show how one might use a Fourier transform on packet arrival times to infer a device's clock skew. Since we believe that our TCP timestamps-based approach is currently our most general passive technique, we focus on the TCP timestamps approach in this paper.

An active adversary could also exploit the ICMP protocol to fingerprint a physical device. Namely, an active adversary could issue ICMP Timestamp Request messages to the fingerprintee and record a trace of the resulting ICMP Timestamp Reply messages. If the fingerprintee does not maintain its system time via NTP or does so only infrequently and if the fingerprintee replies to ICMP Timestamp Requests, then an adversary analyzing the resulting ICMP Timestamp Reply messages will be able to estimate the fingerprintee's system time clock skew. Default Red Hat 9.0, Debian 3.0, FreeBSD 5.2.1, OpenBSD 3.5, and Windows 2000 and XP and Pocket PC 2002 installations all satisfy the above preconditions.

PARAMETERS OF INVESTIGATION. Toward developing the area of remote physical device fingerprinting via remote clock skew estimation, we must address the following set of interrelated questions:

(1) For what *operating systems* are our remote clock skew estimation techniques applicable?

(2) What is the *distribution* of clock skews across multiple fingerprintees? And what is the *resolution* of our clock skew estimation techniques? (I.e., can one expect two machines to have measurably different clock skews?)

(3) For a single fingerprintee, can one expect the clock skew estimate of that fingerprintee to be relatively *constant* over long periods of time, and through reboots, power cycles, and periods of down time?

(4) What are the effects of a fingerprintee's *access technology* (e.g., wireless, wired, dialup, cable modem) on the clock skew estimates for the device?

(5) How are the clock skew estimates affected by the *distance* between the fingerprinter and the fingerprintee?

(6) Are the clock skew estimates *independent of the fingerprinter*? I.e., when multiple fingerprinters are measuring a single fingerprintee at the same time, will they all output (approximately) the same skew estimates?

(7) How much *data* do we need to be able to remotely make accurate clock skew estimates?

Question (6) is applicable because common fingerprinters will probably use NTP-based time synchronization when capturing packets, as opposed to more precise CDMA- or GPS-synchronized timestamps. Answers to the above questions will help determine the efficacy of our physical device fingerprinting techniques.

EXPERIMENTS AND HIGH-LEVEL RESULTS. To understand and refine our techniques, we conducted experiments with machines that we controlled and that ran a variety of operating systems, including popular Linux, BSD, and Microsoft distributions. In all cases we found that we could use at least one of our techniques to estimate clock skews of the machines, and that we required only a small amount of data, though the exact data requirements depended on the operating system in question. For the most popular operating systems, we observed that when the system did not use NTP- or SNTP-based time synchronization, then the TCP timestamps-based and the ICMP-based techniques yielded

approximately the same skew estimates. This result, coupled with details that we describe in the body, motivated us to use the TCP timestamps-based method in most of our experiments. We survey some of our experiments here.

To understand the effects of topology and access technology on our skew estimates, we fixed the location of the fingerprinter and applied our TCP timestamps-based technique to a single laptop in multiple locations, on both North American coasts, from wired, wireless, and dialup locations, and from home, business, and campus environments (Table 3). All clock skew estimates for the laptop were close — the difference between the maximum and the minimum skew estimate was only 0.67 ppm. We also simultaneously measured the clock skew of the laptop and another machine from multiple PlanetLab nodes throughout the world, as well as from a machine of our own with a CDMA-synchronized Dag card [1, 9, 11, 17] for taking network traces with precise timestamps (Table 4). With the exception of the measurements taken by a PlanetLab machine in India (over 300 ms round trip time away), for each experiment, all the fingerprinters (in North America, Europe, and Asia) reported skew estimates within only 0.56 ppm of each other. These experiments suggest that, except for extreme cases, the results of our clock skew estimation techniques are independent of access technology and topology.

Toward understanding the distribution of clock skews across machines, we applied the TCP timestamps technique to the devices in a trace collected on one of the U.S.'s Tier 1 OC-48 links (Figure 2). We also measured the clock skews of 69 (seemingly) identical Windows XP SP1 machines in one of our institution's undergraduate computing facilities (Figure 3). The latter experiment, which ran for 38 days, as well as other experiments, show that the clock skew estimates for any given machine are approximately constant over time, but that different machines have detectably different clock skews. Lastly, we use the results of these and other experiments to argue that the amount of data (packets and duration of data) necessary to perform our skew estimation techniques is low, though we do not perform a rigorous analysis of exactly what "low" means.

APPLICATIONS AND ADDITIONAL EXPERIMENTS. To test the applicability of our techniques, we applied our techniques to a `honeyd` [24] virtual honeynet consisting of 100 virtual Linux 2.4.18 hosts and 100 virtual Windows XP SP1 hosts. Our experiments showed with overwhelming probability that the TCP flows and ICMP timestamp responses were all handled by a single machine as opposed to 200 different machines. We also applied our techniques to a network of five virtual machines running under VMware Workstation [4] on a single machine. In this case, the clock skew estimates of the virtual machines are significantly different from what one would expect from real machines (the skews were large and *not* constant over time; Figure 5). An application of our techniques, or natural extensions, might therefore be to remotely detect virtual honeynets.

Another applications of our techniques is to count the number of hosts behind a NAT, even if those hosts use random or constant IP IDs to counter Bellovin's attack [7], even if all the hosts run the same operating system, and even if not all of the hosts are up at the same time. Furthermore, when both our techniques and Bellovin's techniques are applicable, we expect our approach to provide a much higher degree of resolution. One could also use our techniques for forensics purposes, e.g., to argue whether or not a given laptop was connected to the Internet from a given access location. One could also use our techniques to help track laptops as they move, perhaps as part of a Carnivore-like project (here we envision our skew estimates as one important component of the tracking; other components could be information gleaned from existing operating system fingerprinting techniques, usage characteristics, and other heuristics). One can also use our techniques to catalyze the unanonymization of prefix-preserving anonymized network traces [28, 29].

BACKGROUND AND RELATED WORK. It has long been known that seemingly identical computers can have disparate clock skews. The NTP [19] specification describes a method for reducing the clock skews of devices' system clocks, though over short periods of time an NTP-synchronized machine may still have slight clock skew. In 1998 Paxson [22] initiated a line of research geared toward eliminating clock skew from network measurements, and one of the algorithms we use is based on a descendent of the Paxson paper by Moon, Skelly, and Towsley [20]. Further afield, though still related to clock skews, Pásztor and Veitch [21] have created a software clock on a commodity PC with high accuracy and small clock skew. One fundamental difference between our work and previous work is our goal: whereas all previous works focus on creating methods for eliminating the effects of clock skews, our work exploits and capitalizes on the effects of clock skews.

Anagnostakis et. al. [6] use ICMP Timestamp Requests to study router queuing delays. It is well known that a network card's MAC address is supposed to be unique and therefore could serve as a fingerprint of a device assuming that the adversary can observe the device's MAC address and that the owner of the card has not changed the MAC address. The main advantage of our techniques over a MAC address-based approach is that our techniques are mountable by adversaries thousands of miles and multiple hops away. One could also use cookies or any other persistent data to track a physical device, but such persistent data may not always be available to an adversary, perhaps because the user is privacy-conscious and tries to minimize storage and transmission of such data, or because the user never communicates that data unencrypted.

See [15] for the full version of this paper.

2 Clocks and clock skews

When discussing clocks and clock skews, we build on the nomenclature from the NTP specification [19] and from Paxson [22]. A *clock* C is designed to represent the amount of time that has passed since some initial time $\mathsf{i}[\mathsf{C}]$. Clock C's *resolution*, $\mathsf{r}[\mathsf{C}]$, is the smallest unit by which the clock can be incremented, and we refer to each such increment as a *tick*. A resolution of 10 ms means that the clock is designed to have 10 ms granularity, not that the clock is always incremented *exactly* every 10 ms. Clock C's *intended frequency*, $\mathsf{Hz}[\mathsf{C}]$, is the inverse of its resolution; e.g., a clock with 10 ms granularity is designed to run at 100 Hz. For all $t \geq \mathsf{i}[\mathsf{C}]$, let $\mathsf{R}[\mathsf{C}](t)$ denote the time reported by clock C at time t, where t denotes the true time as defined by national standards. The *offset* of clock C, $\mathsf{off}[\mathsf{C}]$, is the difference between the time reported by C and the true time, i.e., $\mathsf{off}[\mathsf{C}](t) = \mathsf{R}[\mathsf{C}](t) - t$ for all $t \geq \mathsf{i}[\mathsf{C}]$. A clock's *skew*, $\mathsf{s}[\mathsf{C}]$, is the first derivative of its offset with respect to time, where we assume for simplicity of notation that $\mathsf{R}[\mathsf{C}]$ is a differentiable function in t. We report skew estimates in microseconds per second ($\mu s/s$) or, equivalently, parts per million (ppm). As we shall show, and as others have also concluded [22, 20, 26], it is often reasonable to assume that a clock's skew is constant. When the clock in question is clear from context, we shall remove the parameter C from our notation; e.g., $\mathsf{s}[\mathsf{C}]$ becomes s.

A given device can have multiple, possibly independent, clocks. For remote physical device fingerprinting, we exploit two different clocks: the clock corresponding to a device's *system time*, and a clock internal to a device's TCP network stack, which we call the device's *TCP timestamps option clock* or *TSopt clock*. We do not consider the hardware bases for these clocks here since our focus is not on understanding why these clocks have skews, but on exploiting the fact these clocks can have measurable skews on popular current-generation systems.

THE SYSTEM CLOCK. To most users of a computer system, the most visible clock is the device's *system clock*, $\mathsf{C}_{\mathsf{sys}}$, which is designed to record the amount of time since 00:00:00 UTC, January 1, 1970. Although the system clocks on professionally administered machines are often approximately synchronized with true time via NTP [19] or, less accurately, via SNTP [18], we stress that it is much less likely for the system clocks on non-professionally managed machines to be externally synchronized. This lack of synchronization is because the default installations of most of the popular operating systems that we tested *do not* synchronize the hosts' system clocks with true time or, if they do, they do so only infrequently. For example, default Windows XP Professional installations only synchronize their system times with Microsoft's NTP server when they boot and once a week thereafter. Default Red Hat 9.0 Linux installations do not use NTP by default, though they do present the user with the option of entering an NTP server. Default Debian 3.0, FreeBSD 5.2.1, and OpenBSD 3.5 systems, at least under the configurations that we selected (e.g., "typical user"), do not even present the user with the option of installing `ntpd`. For such a non-professionally-administered machine, if an adversary can learn the values of the machine's system clock at multiple points in time, the adversary will be able to infer information about the device's *system clock skew*, $\mathsf{s}[\mathsf{C}_{\mathsf{sys}}]$.

THE TCP TIMESTAMPS OPTION CLOCK. RFC 1323 [13] specifies the *TCP timestamps option* to the TCP protocol. A TCP flow will use the TCP timestamps option if the network stacks on both ends of the flow implement the option and if the initiator of the flow includes the option in the initial SYN packet. All modern operating systems that we tested implement the TCP timestamps option. Of the systems we tested, Microsoft Windows 2000 and XP are the only ones that do not include the TCP timestamps option in the initial SYN packet (Microsoft Windows Pocket PC 2002 does include the option when initiating TCP flows). In Section 3 we introduce a trick for making Windows 2000- and XP-initiated flows use the TCP timestamps option.

For physical device fingerprinting, the most important property of the TCP timestamps option is that if a flow uses the option, then a portion of the header of each TCP packet in that flow will contain a 32-bit timestamp generated by the creator of that packet. The RFC does not dictate what values the timestamps should take, but does say that the timestamps should be taken from a "virtual clock" that is "at least approximately proportional to real time [13];" the RFC 1323 PAWS algorithm does stipulate (Section 4.2.2) that the resolution of this virtual clock be between 1 ms and 1 second. We refer to this "virtual clock" as the device's *TCP timestamps option clock*, or its *TSopt clock* $\mathsf{C}_{\mathsf{tcp}}$. There is no requirement that a device's TSopt clock and its system clock be correlated. Moreover, for popular operating systems like Windows XP, Linux, and FreeBSD, a device's TSopt clock may be unaffected by adjustments to the device's system clock via NTP. To sample some popular operating systems, standard Red Hat 9.0 and Debian 3.0 Linux distributions[2] and FreeBSD 5.2.1 machines have TSopt clocks with 10 ms resolution, OS X Panther and OpenBSD 3.5 machines have TSopt clocks with 500 ms resolution, and Microsoft Windows 2000, XP, and Pocket PC 2002 systems have TSopt clocks with 100 ms resolution. Most systems reset their TSopt clock to zero upon reboot; on these systems $\mathsf{i}[\mathsf{C}_{\mathsf{tcp}}]$ is the time at which the system booted. If an adversary can learn the values of a device's TSopt clock at multiple points in time, then the adversary may be able to infer information about the device's *TSopt clock skew*, $\mathsf{s}[\mathsf{C}_{\mathsf{tcp}}]$.

[2] We do not generalize this to all Linux distributions since Knoppix 3.6, with the 2.6.7 experimental kernel, has 1 ms resolution.

3 Exploiting the TCP Timestamps Option

In this section we consider (1) how an adversary might obtain samples of a device's TSopt clock at multiple points in time and (2) how an adversary could use those samples to fingerprint a physical device. We assume for now that there is a one-to-one correspondence between physical devices and IP addresses, and defer to Section 8 a discussion of how to deal with multiple active hosts behind a NAT; in this section we do consider NATs with a single active device behind them.

THE MEASURER. The measurer can be any entity capable of observing TCP packets from the fingerprintee, assuming that those packets have the TCP timestamps option enabled. The measurer could therefore be the fingerprintee's ISP, or any tap in the middle of the network over which packets from the device travel; e.g., we apply our techniques to a trace taken on a major Tier 1 ISP's backbone OC-48 links. The measurer could also be any system with which the fingerprintee frequently communicates; prime examples of such systems include a search engine like Google, a news website, and a click-through ads service that displays content on a large number of websites. If the measurer is active, then the measurer could also be the one to initiate a TCP flow with the fingerprintee, assuming that the device is reachable and has an open port. If the measurer is semi-passive or active, then it could make the flows that it observes last abnormally long, thereby giving the measurer samples of the fingerprintee's clock over extended periods of time.

A TRICK FOR MEASURING WINDOWS 2000 AND XP MACHINES. We seek the ability to measure TSopt clock skews of Windows 2000 and XP machines even if those machines are behind NATs and firewalls. But, because of the nature of NATs and firewalls, in these cases we will typically be limited to analyzing flows initiated by the Windows machines. Unfortunately, because Windows 2000 and XP machines do not include the TCP timestamps option in their initial SYN packets, the TCP timestamps RFC [13] mandates that *none* of the subsequent packets in Windows-initiated flows can include the TCP timestamps option. Thus, assuming that all parties correctly implement the TCP RFCs, a passive adversary will not be able to exploit the TCP timestamps option with Windows 2000/XP-initiated flows.

If the adversary is semi-passive, we observe the following trick. Assume for simplicity that the adversary is the device to whom the Windows machine is connecting. After receiving the initial SYN packet from the Windows machine, the adversary will reply with a SYN/ACK, but the adversary will *break the RFC 1323 specification and include the TCP timestamps option in its reply*. After receiving such a reply, our Windows 2000 and XP machines ignored the fact that they did not include the TCP timestamps option in their initial SYN packets, and included the TCP timestamps option in all of their subsequent packets. As an extension, we note that the adversary does not have to be the device to whom the Windows machine is connecting. Rather, the adversary simply needs to be able to mount a "device-in-the-middle" attack and modify packets such that the Windows machine receives one with the TCP timestamps option turned on. If the adversary is the device's ISP, then the ISP could rewrite the Windows machine's initial SYN packets so that they include the TCP timestamps option. The SYN/ACKs from the legitimate recipients will therefore have the TCP timestamps option enabled and, from that point forward, the Windows machine will include the TCP timestamps option in all subsequent packets in the flows.

We applied this technique to Windows XP machines on a residential cable system with a LinkSys Wireless Access Point and a NAT, as well as to Windows XP SP2 machines using the default XP SP2 firewall, and to Windows XP SP1 machines with the Windows ZoneAlarm firewall. (While current firewalls do not detect this trick, it is quite possible that future firewalls might.)

ESTIMATING THE TSOPT CLOCK SKEW. Let us now assume that an adversary has obtained a trace $\mathcal{T}$ of TCP packets from the fingerprintee, and let us assume for simplicity that all $|\mathcal{T}|$ packets in the trace have the TCP timestamps option enabled. Toward estimating a device's TSopt clock skew $\mathsf{s}[\mathsf{C}_{\mathsf{tcp}}]$ we adopt the following additional notation. Let t_i be the time in seconds at which the measurer observed the i-th packet in $\mathcal{T}$ and let T_i be the $\mathsf{C}_{\mathsf{tcp}}$ timestamp contained within the i-th packet. Define

$$\begin{aligned} x_i &= t_i - t_1 \\ v_i &= T_i - T_1 \\ w_i &= v_i/\mathsf{Hz} \\ y_i &= w_i - x_i \\ \mathcal{O}_{\mathcal{T}} &= \{\, (x_i, y_i) \,:\, i \in \{1, \ldots, |\mathcal{T}|\} \,\}\,. \end{aligned}$$

The unit for w_i is seconds; y_i is the *observed offset* of the i-th packet; $\mathcal{O}_{\mathcal{T}}$ is the the *offset-set* corresponding to the trace $\mathcal{T}$. We discuss below how to compute Hz if it is not known to the measurer in advance. As an example, Figure 1 shows the offset-sets for two devices in a two-hour trace of traffic from an Internet backbone OC-48 link on 2004-04-28 (we omit IP addresses for privacy reasons). Shifting the clocks by t_1 and T_1 for x_i and v_i is not necessary for our analysis but makes plots like in Figure 1 cleaner.

If we could assume that the measurer's clock is accurate and that the t values represent true time, and if we could assume that there is no delay between when the fingerprintee generates the i-th packet and when the measurer records the i-th packet, then $y_i = \mathsf{off}(x_i + t_1)$. Under these assumptions, and if we make the additional assumption that

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS
Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS
Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS
Sync your system to PACER to automate legal marketing.