SkewMask: Frustrating Clock Skew Fingerprinting Attempts

Ben Ransford and Elisha Rosensweig

December 12, 2007

Abstract

Previous work has shown that *clock skew* can be considered an identifying feature of a physical system, independent of time of measurement, network location of subject and observer, and other factors. Under certain reasonable assumptions, it is possible to *fingerprint*, or uniquely identify within some anonymity set, a remote machine by observing some of its network traffic and observing changes in the timestamps it sends in outgoing TCP headers.

In this work we present and implement a simple scheme by which a machine can *mask*, or intentionally misrepresent, its clock skew, as a defense against fingerprinting by remote machines. We demonstrate the manner in which this masking scheme can be used to disguise a single machine or a group of machines, and we explore the limitations of our masking defense.

1 Introduction

Anonymity on a network is a desirable property in many situations. Kohno, Broido, and Claffy [2] describe a surprisingly robust mechanism for *fingerprinting* physical machines based on information they leak via the optional (but almost universally enabled) *TSval*, or *timestamp value*, TCP header field. A machine that includes TSval in its TCP headers reveals something about its notion of *time*, and therefore something about the operation of its clock. A remote observer with its own clock can watch the evolution of the target machine's timestamp values and measure the difference between its own clock's speed and that of the target machine. This technique allows an observer to fingerprint, and in some cases identify, hosts. The designers of this technique showed that fingerprinting is robust to changes in network location, latency, and measurement times.

This fingerprinting technique, however, has a drawback in that it relies on information explicitly published by the fingerprintee (the machine being fingerprinted), the aforementioned *TSval*. In order to protect itself from being fingerprinted, a machine could simply disable the timestamp option (usually

called *TSopt*) in its TCP stack, though this would come at the cost of decreased functionality, as certain useful network timing mechanisms would no longer be available. A more subtle and functional solution would be to *alter* the TSval field of outgoing packets in such a way that the machine's clock skew (of which it must be aware) is somehow hidden. This is the approach we take in this work. As we shall show, this defense against fingerprinting can be achieved for machine m by a fairly simple mechanism which involves multiplying each *TSval* by a constant $Mask(m)$, chosen by m. We call our defense *masking* and its implementation *SkewMask*.

Since *SkewMask* modifies a value used by the TCP protocol, it is important to verify that all applications that rely on this value will not malfunction as a result of this change. More importantly, the basic operations of the TCP stack—as defined by the TCP protocol—must not break. The *TCP Timestamps Option* (TSopt) field is an option specified in RFC 1323 [1]. According to the RFC, *TSopt* is used for two purposes: round trip time measurement (RTTM) and protection against wrapped sequence numbers (PAWS). Our mechanism does not break the fundamental operations of the TCP stack, and we consider its effects on the performance of the TCP stack to be acceptable. We discuss its performance properties later in this paper.

Kohno *et al.* make several assumptions, all of which we inherit. First, if we want to identify some host using the clock skew fingerprinting method, we assume that we have an de-anonymized—that is, uncensored—recording of packet headers from that host. Second, we assume that the host to be identified has the TSopt TCP option enabled. Third, we assume that no device between the fingerprinter and the fingerprintee has changed the TSopt value in the fingerprintee's outgoing packets (this assumption is valid in practice, since most network equipment offers no facility for this kind of TCP header manipulation).

TSopt and TSval. *TSopt* refers to the 10-byte block of that name in TCP headers. *TSval* is the 4-byte field in TSopt that leaks information over (and about) time. Section 5 discusses the TSval field—and our treatment of it—in more detail.

Organization of this paper. We begin (section 2) by discussing related work on the topic of clock skew measurement. We then proceed to give a basic description of our proposed masking mechanism (section 3), and we underscore its usefulness by listing scenarios in which it would work. With these applications in mind, we move on (section4) to discuss in depth the manner in which masking parameters—or *masks*—can be chosen, analyzing the effectiveness of specific masks and possible *unmasking* attacks that can be mounted against certain types of masks. This discussion leads us to suggest possible modifications and add mask choice policies to the basic masking mechanism. We describe our testbed and the manner in which we implemented the masking mechanism for this work (section 5), and we display simulation results (section 6). We conclude (section 7) with a discussion of our solution's efficacy and possible side-effects it may have.

2 Related work

Previous work has discussed the cause and measurement of clock skew [7, 4]. Several synchronization techniques, designed to eliminate a cumulative offset from true time, have been suggested as well, in order to help in network synchronization. Common protocols exist for synchronizing the clocks of networked machines, such as NTP [3], though not all machines support these protocols.

In general, clock skew is considered to be a per-machine constant; this is one assumption of the fingerprinting technique in [2]. This assumption is not always valid: Murdoch [5] asserts that increased hardware temperature can affect clock skew in a real and measurable manner. For remote fingerprinting to work in all cases, the fingerprinter needs to take this temperature dependence into consideration and allow for some variance in each target machine's perceived skew. There are also additional factors that can affect the apparent clock skew of a target machine. Specifically, if the packets the target machine transmits vary in size while the connection route remains fixed, the transmission delay of those packets will gradually increase, a fact which may wrongly be incorporated into skew measurements [4]. For simplicity, we assume constant packet size in this paper.

In reaction to the fingerprinting technique suggested in [2], Pang *et al.* [6] discuss this problem specifically in the context of donating packet traces to research labs. They propose to replace the TCP timestamps in network traces with monotonically increasing counters. This is a sensible approach to anonymizing packet traces, but is may come at the cost of disabling certain operations (such as *RTTM*) which require accurate information in TCP timestamp fields. They also consider "fuzzing the timestamps by random amounts," but then reject this idea because "fuzzing" may affect research relying on timing information. Our technique aims to improve upon this strategy while preserving the usefulness of the TSopt TCP header field while still misrepresenting the fingerprintee's clock skew to the fingerprinter, with a goal being that the fingerprinter cannot uniquely identify the fingerprintee based on the latter's apparent clock skew.

Kohno *al.* describe additional physical-machine fingerprinting mechanisms. One of these is identical to the one discussed in this work, only focusing on ICMP packets instead of TCP packets, and the masking suggestions made here could be similarly applied. Another takes a completely different approach, and attempts to determine clock skew when observing traffic of known periodic transmissions. If a specific type of traffic transmits a packet every x ticks, an observer may infer from this information the clock speed of the observed machine and thereby deduce its clock skew. Such attacks are limited in their applicability as they require knowledge of the type of traffic, but defending against them is more difficult because the information leak is no longer explicitly published by the observed machine. The masking suggestions presented here are specifically not designed to combat this latter type of attack.

2.1 Remote Physical Device Fingerprinting

We present here a short description of the clock skew estimation technique described in [2]. We refer the reader to that paper for a full description of the technique and its results.

It is well-known that clocks on computers have *clock skews*, which are slight deviations from true time. This is due to manufacturing anomalies at the hardware level. RFC 1323 specifies the TCP timestamps option which is implemented by and enabled on most operating systems by default. This option requires that the host (machine) inserts into each outgoing TCP packet header a value which it believes is related to the true time. Another machine, even a machine that receives only a trace of the target machine's packets, can observe the rate at which the timestamps change and thereby infer the target machine's clock skew (relative to its own idea of true time).

Given large, anonymized TCP traces, Kohno *et al.* used an approximation method to determine the clock skew of each distinct endpoint. Their method, which is based on Graham's convex hull algorithm for sorted data, chooses a single number to represent the "slope" of a machine's clock skew as it drifts away from true time. This approach appears in [4] as well. We used the same technique—in fact, the same estimation program—as Kohno *et al.* to verify that our clock skew measurements were within reasonable bounds.

3 Applications of Clock Skew Masking

Our approach to protecting a machine against clock skew fingerprinting is to modify the TSval field of outgoing TCP packets. This modification takes the form of multiplication by a constant. We do not modify the *source* of timestamp values; we alter the values that come from that source.

Clock skew comes in two flavors: real clock skew, which refers to the skew of a clock compared to true time, and relative or perceived clock skew, which is the skew of the clock on machine A relative to the clock on machine B. We denote $\mathsf{RSkew(m)}$ to be the actual clock skew of a machine m, and $\mathsf{PSkew(m,f)}$ to be the clock skew of m as measured by a fingerprinter f. *SkewMask* multiplies the timestamp by a constant masking parameter $\mathsf{Mask(m)}$, giving the following formula:

$$\mathsf{PSkew(m,f)} = \frac{\mathsf{RSkew(m)}}{\mathsf{RSkew(f)}} \cdot \mathsf{Mask(m)}$$

By employing *SkewMask*, a machine can control the clock skew it *appears* to have simply by choosing its masking parameter. The specific value of $\mathsf{Mask(m)}$ may be chosen in several ways, depending on the required result. Possible masking strategies include:

- *Random shift.* Even if a machine is unaware of its clock skew, it can mask its clock skew by randomly selecting a masking parameter and applying it. To flummox an observer whose fingerprinting technique depends on

constant clock skew, the machine could choose a new random masking parameter periodically.

One drawback of this approach is that as long as a machine is running, a fingerprinter that is continuously observing traffic might be able to detect each change in the machine's relative clock skew. For example, if only one machine masks its clock skew, and does so via choice of random masking parameters, an attacker could take note of each individual change and map the target machine to the new perceived clock skew.

- *Mirror another machine.* Additional functionality is available when a machine can gather information about other observable machines. For example, machine a may independently determine its clock skew relative to another machine b (perhaps using the method of Kohno *et al.*) and then select $\mathsf{Mask(a)} = \mathsf{PSkew(b,a)}$. This will cause both machines to appear to have the same clock skew.

  Such an approach could be used in combination with the random shift approach described above: machine a could adopt the same apparent clock skew as machine b for a short while, then switch to the random shift strategy. This procedure would leave the attacker in doubt as to which of the two machines has diverged. The effectiveness of mirroring other machines increases as the number of machines using the same apparent skew grows.

- *Group masking.* If several machines decide to coordinate, a process similar to group skew-correction mechanisms (as proposed by Paxson [7]) may be put to action. In these cases, a group of machines might join together in a collaborative effort to adopt a single clock skew[1]. Such a synchronization would be useful when trying to conceal not only machine identities but also the number of machines, as in the case of a NAT.

- *Protect against sophisticated fingerprinting attacks.* In the experiments discussed in [5], the effects of temperature on clock skew were shown to be approximately linear. If the exact linear dependency is known, a machine could adjust the mask it uses periodically in order to compensate for the effects of temperature on clock skew, thus evading fingerprinting.

- *Skew Correction.* If a machine discovers its real clock skew, *SkewMask* could be used to correct the exposed time so that it shows the correct time.

[1] An efficient and distributed way to do this would be for all machines to measure their clock skews relative to the rest of the group, and then select a mask that will shift the perceivable skew to the mean skew of all the machines

4 Masking parameter properties

In this section we discuss in depth how to select a masking parameter for *SkewMask*, considering the possible steps an attacker may take in order to discover that masking is taking place or unmask a machine, revealing its true clock skew. As we shall see, on some systems there may be a tradeoff between runtime overhead of the masking process and its effectiveness. We also explore some possible uses of randomness in a masking scheme. We discuss these issues separately for incrementing masks ($\mathsf{Mask(m)} > 1$) and decrementing masks ($\mathsf{Mask(m)} < 1$), for each of these has different properties.

4.1 Incrementing Masks

We begin by slightly clarifying the masking process. Given a binary number X, let $(X)_k$ be the number resulting from looking only at the k LSB of X. When incrementing masks are being used, the published timestamp is $(TSval \cdot \mathsf{Mask(m)})_{32}$. Thus, in the process of masking, there is some information lost regarding the exact values of the original timestamp, while the rest can be retrieved from the masked published timestamp when the masking parameter is known. This fact is of critical importance since in some cases, such as when measuring RTTM, the masking machine itself may wish to unmask previously published timestamps and use this information. Given a masked timestamp TS_{mask} and a known mask $a = \mathsf{Mask(m)} \in \mathbb{N}$, a machine can determine that, regarding the original timestamp TS,

$$(\frac{TS_{mask}}{a})_{32-\lceil \log_2 a \rceil} = (TS)_{32-\lceil \log_2 a \rceil} \qquad (1)$$

While for $a \in \mathbb{R}$ some rounding errors are possible, the recovered value would be very close to the original TS, minus the MSBs. This fact shall be of crucial importance also when estimating the power of an attacker determined to unmask a machine running *SkewMask*.

Ideally, the masking procedure would be fast enough so as not impede TCP traffic. As masking is simply the multiplication of *TSval* by some constant , the runtime overhead will be a function of the specific masking parameter chosen. By choosing masks which are easy to multiply (e.g. $\mathsf{Mask(m)} = 2^k$ for some $k \in \mathbb{N}$), the overhead should be very slight, if at all noticeable. However, limiting the masking parameters to any specific set of values comes with a price of reduction in effectiveness and applicability. For starters, once an attacker is aware of the set of n masks which might be used, it may use reverse engineering to achieve a set of n possible values for the actual skew: Using Eq. (1), for mild skews most of the information can be recovered once the masking parameter is known and the skews can be determined. This idea is expressed in Figure 1.

The severity of this problem depends on the number of observed machines and/or optional mask values. For both of these, an attacker is forced to consider many more possibilities as one or both parameters increase. Skew re-

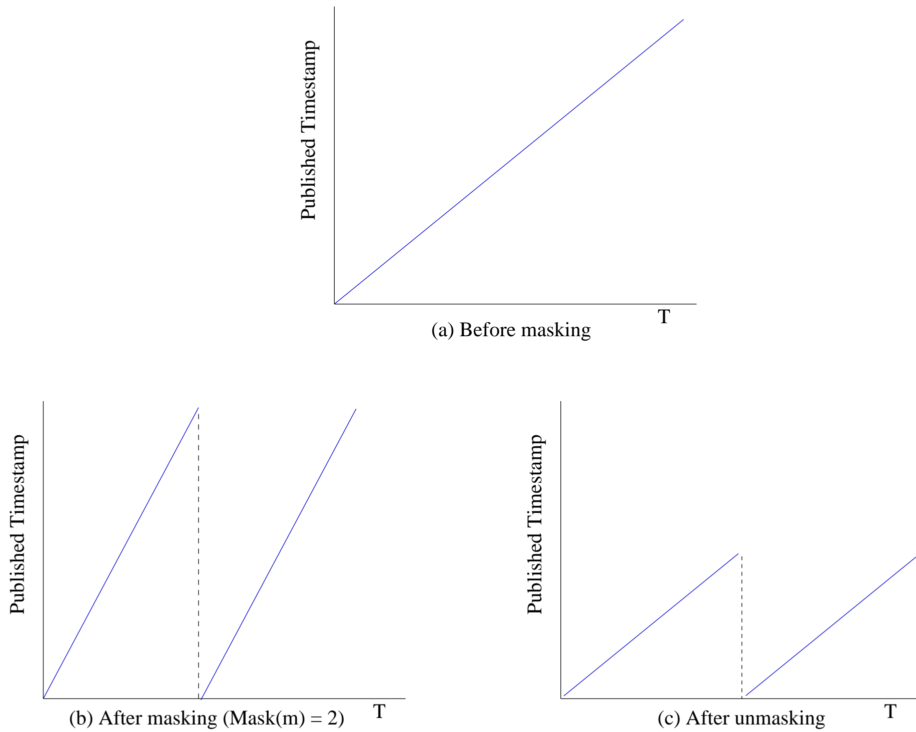

Figure 1: Loss of data when using incrementing masks: (a) The original time measured by a machine. (b) Time as published with mask of value 2. (c) Time as recovered after unmasking. Note that unmasking preserves the skew, but shortens the cycles of time.

covery depends on a mild mask (that is, a masking parameter close to 1) being used, though this is a reasonable assumption since information recovery is also an option the masking machine would likely prefer to conserve. If strong masks (that is, masking parameters far from 1) are being considered for security reasons, disabling the timestamp option might be a safer and more practical option than masking.

Another drawback of using a set of predefined masking parameters is that the applicability of the system decreases as well. For example, in order to effectively mirror another machines' (m') skew, a machine m would need to use a specific mask, specifically $\mathsf{Mask(m)} = \mathsf{PSkew(m',m)}$. If this mask is not included in the set of "fast" masking values, such mirroring cannot be effectively achieved.

Such a tradeoff is unfortunate, and on machines where floating-point multiplication can be done fast enough it can be avoided by allowing for full flexibility in selecting the mask. When limitations on complex multiplication are required, however, it is inevitable that certain operations may not be available, such as skew mirroring, as previously described. In what remains of this sec-

tion, we discuss only the effectiveness of the masking system itself and possible techniques to recover the true clock skew of a machine employing *SkewMask*. This can be thought of as measuring the effectiveness of random selection of a masking parameter.

A simple unmasking attack can be mounted by observing that, prior to cutting off the MSBs, all the masked timestamps have a common denominator $Mask(m)$ - a pattern which is not assumed to exist for standard TCP traffic[2]. Once the most significant bits have been removed, the pattern may not be as clear, but for certain parameters most if not all of the skew information may be discovered. For example, if $\mathsf{Mask(m)} = 2^k$, all the timestamps will have all their k LSBs equal to 0, thus making it clear to a finger-printer which masking parameter was used, and once again the skew can be discovered. Once again, such an attempt may be thwarted in part by using floating-point masking parameters: floating point common denominators are more difficult to determine due to rounding policies (which enter a degree of randomness into the published timestamp) and due to the large number of possible values.

To address the problem when floating point arithmetic is too time-consuming, one solution could be to inject some randomness into every masked timestamp. For example, instead of publishing $(TSval \cdot \mathsf{Mask(m)})_{32}$, we might instead use $(TSval \cdot \mathsf{Mask(m)} + rnd\{0, ..., 2^k\})_3 2$. k is selected in such a way that will ensure the monotonic nature of the published timestamp, by requiring that $Mask(m) > 2^k$. Such a system would thwart any attempt to classify a machine based on a common denominator. Note that division by the mask will eradicate all randomness, ensuring that this patch does not hinder timestamp recovery when needed. However, this raises the potential of unmasking the machine based on timestamp variance, as is discussed next.

Variance in the perceived skew refers to cases when packets do not abide by the skew of their originating machine, i.e. when packets sent close together arrive with a large time gap between them, and vice versa. If no skew variance existed, the following would be true for all packets:

$$PSkew(m, f) = \frac{Sent_m(p2) - Sent_m(p1)}{Arrive_f(p2) - Arrive_f(p1)} \tag{2}$$

Each variable represents the registered time of the event (sending, arrival), and the subscript specifies who is measuring time. Due to changing network conditions, such as changes in the route taken and, mostly, queueing delays, skew variance does exist even without any masking [4]. As can be easily seen, simple masking by multiplication will change the perceived mask by modifying the numerator, but not the skew variance. On the other hand, once randomness is injected as part of the masking process, this will increase the skew variance, possibly to a degree in which it is noticeable[3]. Once such an anomaly is

[2]In [2], the authors present a fingerprinting attack which can be applied only when, due to some application-level reason, TCP packets are sent into the network every t clock ticks. The attack presented here might be fooled by such traffic into believing that masking is occurring.

[3]Determining at what point this becomes noticeable is beyond the scope of this work. Here we only concern ourselves with how this noticeable variance might be used in an unmasking attempt.

detected, the observable variance might be used in order to (partially) unmask a machine, meaning that a closer estimate of the machines clock skew can be made. This is achieved since randomness is injected only in the LSBs, so a simple division of the observable timestamp will get rid of most of the added variance by removing the bulk of random data. If the amount of injected randomness is proportionate to the size of masking parameter, an attacker could divide the timestamps by a factor which would bring the variance back to normal, closing in on the original skew.

4.2 Decrementing Masks

Decrementing masks are similar to incrementing masks, but have several important differences. To begin with, it is clear that in order to use $Mask(m) < 1$, floating point arithmetic is required, which may limit the applicability of such masks on certain machines. Since most of the problems which arose for incrementing masks were problematic mainly when floating point multiplication is unavailable, decrementing masks are not a possible solution to these problems.

In contrast with incrementing masks, the loss of information when using decrementing masks is focused in the LSBs, while the MSBs can be recovered. This, on the one hand, makes the masking more effective, as the LSBs are crucial for determining the skew with any precision, but at the same time may cause trouble for timestamp based applications such as RTTM, as discussed in a later section. In essence, as the mask becomes smaller, the range of possible skews after recovery grows (Fig. 2).

As in the case of incrementing masks, it is important to asses the ability of an observer to discover which mask was used. Assuming $\mathsf{Mask(m)} = \frac{1}{a}$, applying it will ensure that all timestamps are of a value smaller than $\frac{2^{32}}{a}$. This means that for all timestamps, the $\lfloor \log_2 a \rfloor$ MSBs will always be 0, a very easily distinguishable phenomenon. The only way to disguise this is to inspect each timestamp and increment the value in these top MSBs whenever a new cycle of timestamps has been completed. This may increase the overhead of applying the masking.

5 SkewMask Implementation

Our implementation goals were twofold: first, confirm via automated experiments that clock skew measurement of the type proposed by Kohno *et al.* is feasible. Second, demonstrate that a machine's outgoing TCP timestamps can be manipulated without hindering its normal operation.

5.1 Measuring Clock Skew

It is possible to measure the clock skew of a machine *relative to another machine* by receiving TCP packets from each. Our method, which employs a *server*

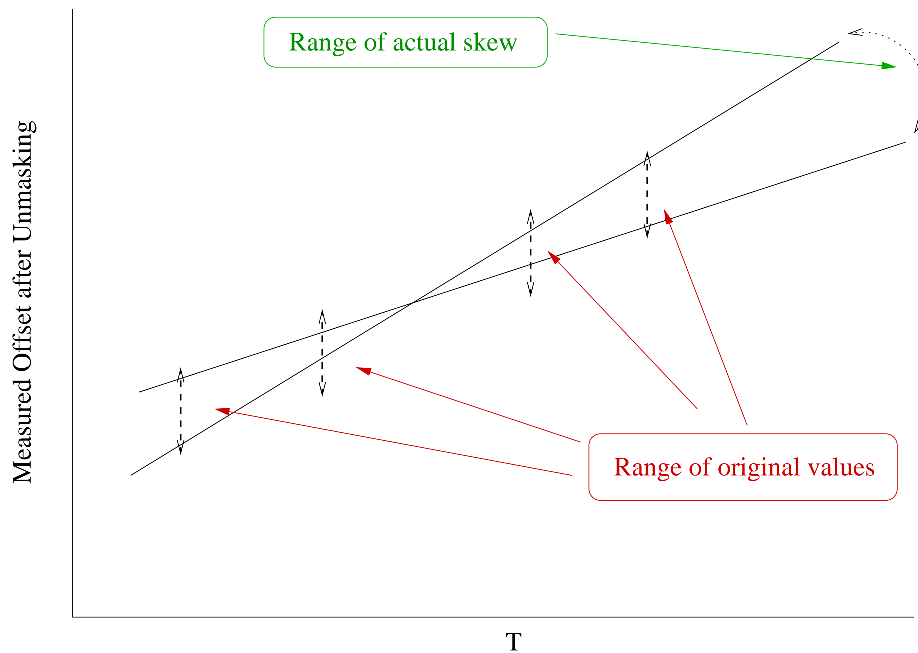

Figure 2: Possible skew after recovery from decrementing mask. The smaller the mask, the greater the range of possible original timestamp values.

machine that observes the timestamps on packets it receives from clients, is as follows:

1. Start a web server that serves up some small text files.
2. Run `tcpdump` on the web server to capture packets from a certain "interesting" set of machines.
3. From each member the set of "interesting" machines, run a shell script that repeatedly (using `wget --recursive`) fetches the entire content of the web server's document root. Stop this after some amount of time.
4. Run the resulting `tcpdump` file through a Python script that records two pieces of information for each packet: the time `tcpdump` saw it, and the value of the `timestamp` field from the packet's header. These two values are written to a file named after the packet's source. (Therefore, with n "interesting" machines, there are $n + 1$ separate files.) `tcpdump`'s time is recorded as the number of seconds since midnight.
5. Load the data files from all of the "interesting" machines into Matlab. Calculate apparent skew as follows. To calculate the number of seconds since the beginning of the capture, for each packet p_i compute

$$x_i = t_i - t_0 \tag{3}$$

where t_i is the time `tcpdump` saw the packet and t_0 is the time `tcpdump` saw the first packet in the capture. Calculate a similar difference for the TSopt value:

$$v_i = T_i - T_0 \tag{4}$$

where T_i is the TSopt value of packet p_i and T_0 is the TSopt value of the first packet in the capture. Since Hz, the TSopt clock frequency, is unknown a priori, compute it for a given host h from a sequence of m packets with that host as source, as follows:

$$\Delta_T = v_m = T_m - T_0$$

$$\Delta_t = x_m = t_m - t_0$$

$$\mathsf{Hz}_h = 10^{round(log_{10}(\Delta_T - \Delta_t))} \tag{5}$$

Finally, compute the estimated offset $\delta_{h,i}$ for a host h at packet i:

$$\delta_{h,i} = \frac{v_i}{\mathsf{Hz}_h} \tag{6}$$

Values of Hz. Using Equation 5, we computed Hz values for various operating systems; see Figure 4.

Results. Figure 3 shows the result of measuring 4 machines in the UMass EdLab for about 3 hours using the above procedure.

5.2 Linux kernel modifications

We have modified the Linux 2.6 kernel to change the relationship between the system clock and the values that appear in the TSval field of outgoing TCP packets.

Modifications to the Linux kernel can, as ours do, take two forms: modifications (or *patches*) to existing kernel code, and separate *modules* that can be loaded and unloaded on demand. The former approach requires recompilation of the kernel after patching, which is unappealing for the time it takes; however, an advantage of the patching approach is that a patch may be shorter in length than a module intended to perform the same function. Kernel modules do not require kernel recompilation and are therefore more convenient to build and test.

Altering TSval via a kernel patch. The Linux kernel counts timer interrupts in a kernel variable called `jiffies`. A kernel *jiffy*, as its name implies, is a small amount of time; how small depends on the value of a machine-dependent constant called `HZ`. (We refer to this variable as Hz, following the notation of Kohno *et al.*; see Section 5.1 for discussion of Hz measurement.) By design, exactly Hz jiffies elapse every second. The `jiffies` counter starts at zero at boot time. In Linux 2.6, `jiffies` is an unsigned 32-bit integer; the Linux Cross Reference[4] shows where this number is defined and used.

[4] `http://lxr.linux.no/ident?i=tcp_time_stamp`

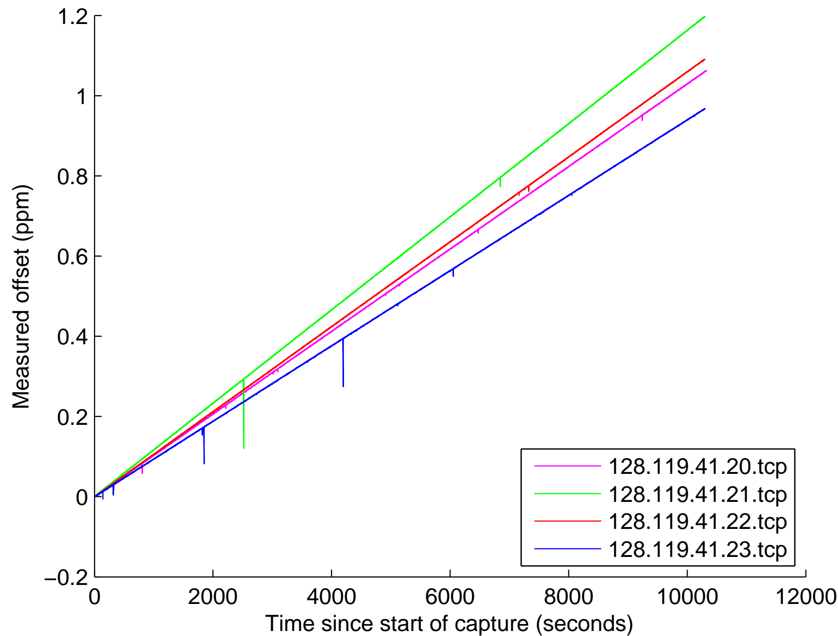

Figure 3: Skew estimates for 4 identically configured machines in the UMass EdLab.

Linux's TCP stack simply inserts the value of the `jiffies` variable into the TSopt field of outgoing TCP packets. The line that defines this relationship appears in the kernel's `net/tcp.h`:

```
#define tcp_time_stamp          ((__u32)(jiffies))
```

Other parts of the code then refer to the `tcp_time_stamp` macro; see Figure 5 for an example usage.

To implement a multiplying mask of the sort we describe above, we needed to modify only one line of code in the kernel, namely the `tcp_time_stamp` macro definition. We redefined the macro as follows:

```
#define tcp_time_stamp  (((__u32)(jiffies)) << 3)
```

After recompiling the kernel, we repeated our remote measurement procedure and verified that the timestamps coming from our test machine were multiples of 8—that is, they had `000` as their three least significant bits.

More complicated masking procedures involving more than simple arithmetic could be implemented with a slightly larger kernel patch: one could implement a function `tcp_time_stamp_fn()` in `tcp.h`, then replace all occurrences of `tcp_time_stamp` in the kernel's source code with calls to the new function. We did not test this modification.

OS	Hz	Notes
CentOS Linux 5.0	1000	UMass EdLab and cusp.cs.umass.edu.
Ubuntu Linux 7.04	100	on a Xen VM in the Midwest
Ubuntu Linux 7.10	100	Northampton, MA
FreeBSD 4.11	100	Syracuse, NY

Figure 4: Hz (TSval clock frequency) values for various operating systems.

```
if (ts) {
    rep.opt[0] = htonl((TCPOPT_NOP << 24) |
                       (TCPOPT_NOP << 16) |
                       (TCPOPT_TIMESTAMP << 8) |
                       TCPOLEN_TIMESTAMP);
    rep.opt[1] = htonl(tcp_time_stamp);
    rep.opt[2] = htonl(ts);
    arg.iov[0].iov_len += TCPOLEN_TSTAMP_ALIGNED;
}
```

Figure 5: Linux's TCP stack inserts the TSopt field into an ACK packet. Code from `tcp_v4_send_ack` procedure in `net/ipv4/tcp_ipv4.c`.

The kernel patch approach presents a distinct advantage: by redefining the *source* of TSval, we need not perform any alteration of already-assembled packets. As we discuss below, the kernel module approach is much more complex in this regard.

Altering TSval via a kernel module. We have written a kernel module for Linux 2.6 that alters the TSopt field of outgoing packets for the purpose of hiding the host machine's clock skew. Our module is a *proof of concept*; it is meant to show that our approach is practicable. The source code of our kernel module, which we called `mod_skewmask.c`, appears in the appendix.

The kernel module is loaded via `insmod mod_skewmask.ko`, optionally followed by a `shift_bits=<n>` parameter. `insmod` is the standard way to load a module into a running Linux kernel. The `shift_bits` parameter allows the `root` user—the only user allowed to load modules into the kernel—to select a number of bits by which each TSval value will be left-shifted; it corresponds to a variable of the same name in the module's source code. The value of the `shift_bits` variable can be read and written via a file at `/proc/sk_shiftbits` whenever the module is loaded in the kernel.

During the initialization routine called when the module is loaded, our module calls a kernel function named `dev_add_pack()` to interpose its own function (`mysendfn()`) into the IP stack's packet processing chain. Because our function is called on every packet the system processes, it quickly returns if its input is not an outgoing TCP packet. Much to the chagrin of this report's authors, the only place a function like our `mysendfn()` can be interposed is

at the *end* of the packet processing chain; our function sees packets *after* they have been assembled by the kernel.

Our kernel module's TSval alteration procedure is straightforward: given a packet, it reads the existing timestamp if one exists, finds the same timestamp in the sequence of bytes representing the TCP header, and overwrites it with a new value. The accounting work our function does after altering a TCP packet's TSval field is less straightforward: we must recompute the packet checksum and replace it in the TCP header. Because of this extra nontrivial step, and because we were interested only in the feasibility of our approach, we did not implement checksum recalculation for the general case; we implemented a simpler version of checksum recalculation in a simpler version of our module. The simpler module merely increments the checksum field if the TSval field was discovered to hold an odd value (in which case our module replaces the TSval field's trailing `1` bit with a `0`).

Limitations of our implementation. We have not yet decided how to deal with integer overflow, which becomes more of a possibility as we multiply TSval by larger numbers.

6 Simulation

In order to test the correctness of our scheme, we estimated the clock skew of several machines in the Edlab based on packet traces taken with `tcpdump`, applied *SkewMask* to one of them, and repeated the same estimation over the same traffic. As can be seen clearly from comparing figures 3 and 6, the masking is complete, as the masked machine appears to have the same skew as another. This proves that multiplication can change the perceived clock skew, and that it gives a machine sufficient control over the exact perceived skew the mask will produce.

As mentioned earlier, floating point arithmetic might cause significant overhead on certain machines. We therefore ran a short program which performed many multiplication operations, both with integer and with float numbers. The measured speeds of each are presented in Figure 7. It is clear from these speed differences that if this calculation were to become a bottleneck in the TCP packet assembly process, allowing floating point calculations may indeed have a noticeable effect on the outgoing traffic flow. However, it is unlikely that this should be a problem in practice, as end-to-end delay is normally several milliseconds, whereas as single floating-point calculations seem to be at least three orders of magnitude faster.

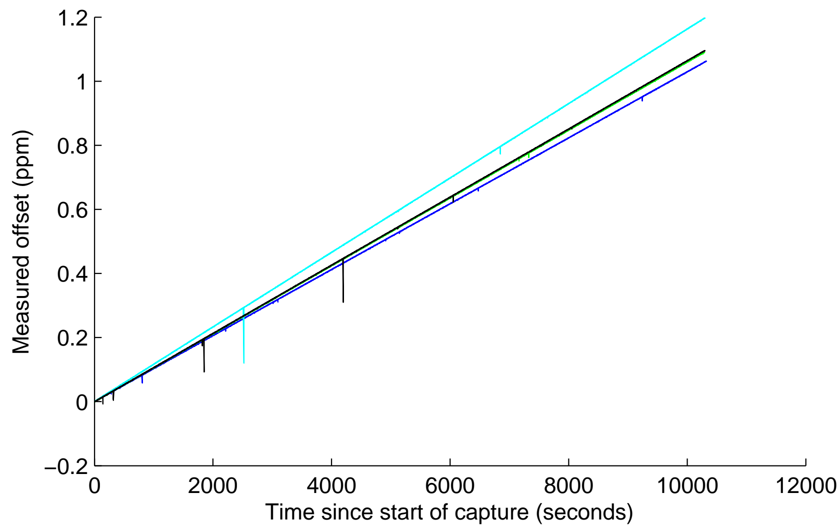

Figure 6: The result of applying a masking function to an EdLab machine. Compare to 3. One of the machines in the group is masking itself, so only three distinct skews can be discerned.

Machine	integer	float	float/integer ratio
PowerPC (2GHz G5)	0.89	2.66	2.988
Pentium 4 (1.8 GHz)	0.56	3.79	6.767

Figure 7: Median time (sec) of performing 10^8 multiplications of float and integers, on different machines.

7 Discussion

7.1 Side effects: RTTM and PAWS

We return now to analyze the possible effects of *SkewMask* on the classical usage of the timestamp option in TCP traffic.

RTTM. RTTM is accomplished in the following manner. Host A sends a TCP packet p_1 to host B with a value T_A in the TSopt field; host B sends back a TCP packet p_2 with its own T_B in the TSopt field and T_A in the *timestamp echo reply* (TSecr) field. Host A can subtract the time it sent p_1 from the time it received p_2, and use this as an estimation of the RTTM. Since the RTTM is evaluated by the machine which originally sent the first message (A), masking should not have a negative effect on RTTM performance, since the estimating machine is aware of the exact mask being used. This, however, does limit the freedom of the machine in choosing the masking parameter. As discussed in earlier sections, using very large incrementing masks and very small decrementing masks causes timestamp information to be lost in such a way that cannot later be recovered. This can cause RTTM estimations to go off mark. This danger is especially relevant to decrementing masks, which eliminate information regarding the original LSBs, while the MSBs can be assumed to change on a much slower scale that is less relevant to RTTM.

PAWS. PAWS is a mechanism for discarding old packets which have already been sent and received. To quote the RFC directly, "a segment can be discarded as an old duplicate if it is received with a timestamp SEG.TSval less than some timestamp recently received on this connection.". This is simply done by saving timestamps of incoming packets and comparing them to future incoming packets, and discarding packets with timestamps too low (early) to be relevant anymore. Since skew masking retains both the relative order of timestamps and the basic notion of "time", the use of PAWS is not hampered even though the receiving end of the connection is unaware of the masking taking place. It would also seem that, contrary to RTTM, the performance of PAWS would be affected only if extremely large or small masks were to be used.

One-way delay measurement. In addition to RTTM, there are also schemes in which packet delay is measured in a one-way fashion, by simply measuring the difference between TSval and the time the packet was received at the destination. From the above it is clear that such a measurement assumes that no masking is taking place, and that the clocks of both ends of the connection are synchronized. If such measurements need to be computed while masking is in place, the masked machine would need to communicate to the destination host what the masking parameter is. Given that the mask is not too strong (that is, that the masking parameter is not too large), this should give reasonable results.

7.2 Future research

Previous attempts at removing skew have focused on synchronizing machines at a given point in time, ensuring that they show the same time for some period thereafter. However, since clock skew is caused by slight hardware anomalies, it is impossible to eradicate these in the long run. In light of this, *SkewMask* offers an additional level of synchronization: Once a skew $RSkew(m)$ is known, a matching mask $Mask(m) = RSkew(m)^{-1}$ can be applied. Such a mask will lengthen the time for which the machine will *appear* to be synchronized to observing machines. The exact degree to which this approach can be useful depends on the precision of both the known clock skew and the chosen mask. Still, this cannot be a permanent solution due to limited mask precision, rounding policies for float calculation and variations in the clock skews (which can be caused, for example, by temperature changes[5]).

References

[1] D. Borman, B. Braden, and V. Jacobson. TCP extensions for high performance. RFC 1323, Internet Engineering Task Force, May 1992.

[2] T. Kohno, A. Broido, and K. Claffy. Remote physical device fingerprinting. In *IEEE Transactions on Dependable and Secure Computing*, 2005.

[3] D. Mills. Network time protocol (version 3) specification, implementation, 1992.

[4] S. B. Moon, P. Skelly, and D. Towsley. Estimation and removal of clock skew from network delay measurements. Technical Report UM-CS-1998-043, , 1998.

[5] S. J. Murdoch. Hot or not: revealing hidden services by their clock skew. In *Computer and Communications Security*, pages 27–36, 2006.

[6] R. Pang, M. Allman, V. Paxson, and J. Lee. The devil and packet trace anonymization. *Computer Communication Review*, 36(1):29–38, 2006.

[7] V. Paxson. On calibrating measurements of packet transit times. In *Measurement and Modeling of Computer Systems*, pages 11–21, 1998.

APPENDIX

```
#include <linux/module.h>  /* needed by all modules */
#include <linux/kernel.h>  /* needed for KERN_INFO */
#include <linux/init.h>    /* needed for various convenience macros */
#include <linux/proc_fs.h> /* needed to interact with /proc/ */
#include <asm/uaccess.h>   /* for copy_from_user */
#include <net/tcp.h>
```

```
#include <net/ip.h>

#define PROCFILENAME "sk_shiftbits"
struct proc_dir_entry* procfile;

#define MODULENAME "mod_skewmask"

/*
 * shift_bits parameter.  Install the module with "insmod mod_skewmask
 * shift_bits=n" to set the number of bits to shift TSopt clock values.
 */
static short int shift_bits = 3;
module_param(shift_bits, short, S_IRUSR | S_IWUSR |  S_IRGRP | S_IROTH);
MODULE_PARM_DESC(shift_bits,
        "Shift TSopt clock value left by this many bits");

static struct packet_type myproto;

/*
 * emit the contents of the shift_bits parameter as a string into the
 * provided buffer.
 */
static int
procfile_read(char *buffer, char** buffer_location,
        off_t offset, int buffer_length, int *eof, void *data)
{
    return (offset > 0)
        ? 0
        : snprintf(buffer, buffer_length, "%d\n", shift_bits);
}

/*
 * write a new value to the shift_bits parameter.
 */
static int
procfile_write(struct file* file, const char* __user buffer,
        unsigned long count, void *data)
{
    char buf[6];
    short int sb = shift_bits;
    size_t len = min((unsigned long)sizeof(buf) - 1, count);
    char* nl;

    /* try copying the new sb value from user-space to kernel-space */
    if (copy_from_user(buf, buffer, len))
        return count; /* this is actually bad */
```

```
    buf[len] = '\0';
    for (nl = buf; *nl != '\0'; ++nl) {
        if (*nl == '\n')
            *nl = '\0';
    }
    printk(KERN_INFO MODULENAME ": got '%s'\n", buf);

    /* after some checking, copy the new sb value to our shift_bits
     * parameter */
    if (sscanf(buf, "%hd", &sb) != 1)
        printk(KERN_INFO MODULENAME
                ": %s is not a valid short", buf);
    else if (sb < 0 || sb > 32)
        printk(KERN_INFO MODULENAME
                ": %s is not between 0 and 32", buf);
    else
        shift_bits = sb;

    return 1; /* this value doesn't actually matter */
}

static int
mysendfn (struct sk_buff *skb,
        struct net_device *dv,
        struct packet_type *pt)
{
    u32 stamp_orig, stamp;
    struct tcphdr* th = tcp_hdr(skb);
    struct tcp_options_received tmp_opt;
    u32* tsptr;
    u32* ckptr;

    /* we want to touch only outgoing TCP packets */
    if (ntohs(skb->protocol) != ETH_P_IP) goto cleanup;
    if (skb->pkt_type != PACKET_OUTGOING) goto cleanup;
    if (ip_hdr(skb)->protocol != IPPROTO_TCP) goto cleanup;

    /* read the stamp from the outgoing packet */
    stamp_orig = TCP_SKB_CB(skb)->when;
    if (!stamp_orig) goto cleanup;

    /* XXX comment me */
    tsptr = th;
    tsptr += 11;
    if (ntohl(*tsptr)!=stamp_orig) { tsptr++; } /* try the next byte */
    if (ntohl(*tsptr)==stamp_orig) {
```

```
        if (stamp_orig & 1) {
            printk(KERN_DEBUG MODULENAME
                    ": fixing ts; %08x->%08x\n",
                    ntohl(*tsptr), stamp_orig & ~1);
            *tsptr = htonl(stamp_orig & ~1);
            TCP_SKB_CB(skb)->when = stamp_orig & ~1;
            /* XXX HACK: fix up checksum manually! */
            ckptr = tsptr;
            ckptr -= 3;
            printk(KERN_DEBUG MODULENAME ": orig cksum %04x\n",
                    ntohl(*ckptr));
            *ckptr += htonl(1); /* simply increment the checksum */
            printk(KERN_DEBUG MODULENAME ":  new cksum %04x\n",
                    ntohl(*ckptr));
        }
    }

cleanup:
    kfree_skb(skb);
    return 0; /* XXX this return value doesn't seem to matter */
}


/*
 * Hook into the sending stack.
 */
static void init_proto (void) {
    myproto.type = htons(ETH_P_ALL);
    myproto.func = mysendfn;
    myproto.dev = NULL; /* NULL means get packets from all devices */
    dev_add_pack(&myproto);
}

/*
 * Module initialization routine.
 */
static int __init skewmask_init (void) {
    printk(KERN_INFO "Installing mod_skewmask (shift_bits=%d)\n",
            shift_bits);

    /*
     * Attempt to create /proc/PROCFILENAME.  If this fails, assume
     * we're out of memory.  Otherwise, set various parts so that the
     * result looks like this:
     *
     *   -rw-r--r-- 1 root root 1 2007-12-06 16:27 /proc/sk_shiftbits
```

```
     *
     */
    procfile = create_proc_entry(PROCFILENAME, 0644, NULL);
    if (procfile == NULL) {
        remove_proc_entry(PROCFILENAME, &proc_root);
        printk(KERN_ALERT "Error: unable to create /proc/%s\n",
                PROCFILENAME);
        /* blame an out of memory error (ENOMEM). */
        return -ENOMEM;
    }
    procfile->read_proc = procfile_read;
    procfile->write_proc = procfile_write;
    procfile->owner = THIS_MODULE; /* this is defined, don't worry */
    procfile->mode = S_IFREG | S_IRUGO | S_IWUSR;
    procfile->uid = procfile->gid = 0;
    procfile->size = 1; /* XXX */
    printk(KERN_INFO "Created /proc/%s\n", PROCFILENAME);

    /*
     * Hook into the kernel's sending stack
     */
    init_proto();

    return 0;
}

/*
 * Module exit routine.  Clean up if necessary.
 */
static void __exit skewmask_exit (void) {
    dev_remove_pack(&myproto);
    remove_proc_entry(PROCFILENAME, &proc_root);
    printk(KERN_INFO "Removed /proc/%s\n", PROCFILENAME);
    printk(KERN_INFO "Uninstalling mod_skewmask\n");
}

/*
 * Tell the kernel what our init and exit routines are called.
 */
module_init(skewmask_init);
module_exit(skewmask_exit);

/*
 * Metadata so that the kernel doesn't complain about non-free software.
 */
MODULE_LICENSE("GPL");
```

```
MODULE_AUTHOR("{ransford,elisha}@cs.umass.edu");
MODULE_DESCRIPTION("Masks clock skew by futzing with TSopt values.");
```