Spring 5-7-2010

Improving the measurement of system time on remote hosts

Michael Christopher Smith
James Madison University

Follow this and additional works at: https://commons.lib.jmu.edu/master201019

Part of the Computer Sciences Commons

Improving the Measurement of System Time on Remote Hosts

Michael C. Smith

A thesis submitted to the Graduate Faculty of

JAMES MADISON UNIVERSITY

In

Partial Fulfillment of the Requirements

for the degree of

Master of Science

Computer Science

May 2010

Dedication

This thesis document is the culmination of a seven year journey to learn a new skill set and begin a new career as a software developer. I dedicate this work to my lovely wife Lanette – who provided unending support, encouragement, patience, and an unshakable confidence that I could accomplish this – and to my three wonderful children who came into this world along the way – Zachary, Isaiah, and Laurel. You endured many nights when Daddy didn't see you because I was in class, and many hours I couldn't spend with you because I had too much work to do. It took a little longer than we thought it would, but now we can say, "Thank God that it is all done!"

Acknowledgements

I would like to take a moment to acknowledge the contributions of my thesis committee members – Dr. Brett Tjaden, Dr. Florian Buchholz, and Dr. Steve Wang – without whose help this document could not have been produced. My interest in the field of measuring system time on remote hosts was stimulated by previous work conducted jointly by Drs. Tjaden and Buchholz. Both contributed greatly to my comprehension, definition, and solution of the problem addressed by my work, the development of my experiments, my direction in researching of related work, and the editing of this document. All three members likewise made significant contributions to my understanding of the technologies and concepts involved and provided critical assistance in getting past coding challenges.

Additionally, I would like to express my appreciation for all of the dedicated and highly qualified professors in the James Madison University Computer Science Department whose classes I had the great fortune to take. Not only did I learn a great deal, but I also enjoyed every class, soaking up the wealth of knowledge offered in a stimulating , challenging, and supportive atmosphere. I am forever grateful and indebted to you: Dr. Charles Abzug, Dr. Mohamed Aboutabl, Dr. Ralph Grove, Dr. Brett Tjaden, Dr. Bob Tucker, and Dr. Steve Wang.

Table of Contents

List of Tables

List of Figures

Abstract

The tools and techniques of digital forensics are useful in investigating system failures, gathering evidence of illegal activities, and analyzing computer systems after cyber attacks. Constructing an accurate timeline of digital events is essential to forensic analysis, and developing a correlation between a computer's system time and a standard time such as Coordinated Universal Time (UTC) is key to building such a timeline.

In addition to local temporal data, such as file MAC (Modified, Accessed, and Changed/Created) times and event logs, a computer may hold timestamps from other machines, such as email headers, HTTP cookies, and downloaded files. To fully understand the sequence of events on a single computer, investigators need dependable tools for building clock models of all other computers that have contributed to its timestamps.

Building clock models involves measuring the system times on remote hosts and correlating them to the time on the local machine. Sending ICMP or IP timestamp requests and analyzing the responses is one way to take this measurement. The Linux program *clockdiff* utilizes this method, but it is slow and sometimes inaccurate. Using a series of 50 packets, *clockdiff* consumes an average of 11 seconds in measuring one target. Also, *clockdiff* assumes that the time difference between the local and target hosts is never greater than 12 hours. When it receives a timestamp showing a greater difference, it manipulates this value without alerting the user, reporting a result that could make the target appear to be more tightly synchronized with the local host than it actually is. Thus, *clockdiff* is not the best choice for forensic investigators.

As a better alternative, we have designed and implemented a program called *clockvar*, which also uses ICMP and IP timestamp messages. We show by experiment that *clockvar* maintains precision when system times on the local and target hosts differ by twelve to twenty-four hours, and we demonstrate that *clockvar* is capable of making measurements up to 1400 times faster than *clockdiff*.

I. Introduction

Computers are very good at doing a multitude of operations quickly. However, to keep track of time, they almost universally rely on inexpensive quartz crystals of unreliable quality. Thus, they are not inherently good keepers of civil, or real-world, time. Knowing the relationship between the time maintained by a computer and civil time is extremely useful in many situations, some of which are listed in an article entitled "Why is NTP Important?", which appears on the NTP (Network Time Protocol) Public Services Project home page:

> "In a commercial environment, accurate time stamps are essential to everything from maintaining and troubleshooting equipment and forensic analysis of distributed attacks, to resolving disputes among parties contesting a commercially valuable time-sensitive transaction. In a programming environment, time stamps are usually used to determine what bits of code need to be rebuilt as part of a dependency checking process as they relate to other bits of code and the time stamps on them, and without good time stamps your entire development process can be brought to a complete standstill. Within law enforcement, they are essential for correlation of distributed communication events, forensic analysis, and potential evidentiary use in criminal proceedings. In essence, all debugging, security, audit, and authentication is founded on the basis of event correlation (knowing exactly what happened in what order, and on which side)."

Computer scientists have conducted a great deal of research on computer timekeeping, especially as it relates to digital forensic science. Because many computers – even hosts

seemingly far removed from a potential digital crime scene – can be involved in a security incident, it may be helpful or even necessary to use time data from these machines in constructing or refining a timeline of events on a particular host. In order to use temporal data from remote hosts in this timelining process, we must understand the correlation between the time on the remote hosts and time on the machine which is the target of the investigation. Our analysis of existing tools and techniques for measuring time on computers across the Internet has revealed a need for improvement in this field, and it has led us to develop our own tool for measuring the difference between system times on local and remote hosts.

1.1 The Significance of Establishing a Timeline of Digital Events

The ability to correlate computer events to real-world events is an essential element of digital investigations. Establishing a timeline of events helps us to understand how the events relate to one another; that is, which events are causes or effects of other events. Building a timeline has been useful in the prosecution of individuals for crimes that they have committed using a computer. In the event of an intrusion into a network, multiple computers may be involved, and if their system clocks are not tightly synchronized, building an accurate timeline is even more critical to understanding the flow of events. Timelining has also proven extremely useful in analyzing major system failures such as the widespread U.S. and Canadian blackout in 2003. In both system failures and security incidents, rarely is only one computer affected; thus it is often necessary to understand how multiple computers' perceptions of time relate to a standard civil time, even when these hosts are dispersed across the Internet.

1.2 Determining System Time on Local and Remote Hosts

A computer's operating system maintains data about every object, such as a file or directory, on the local hard drive. This data includes the Modified, Accessed, and Changed/Created (MAC) timestamps associated with each object. (Regarding the Changed/Created time, Unix systems keep track of any time a file's metadata, such as ownership or permissions, "changes"; Windows systems, however, preserve only the time a file was "created.") A close examination of the MAC times on a particular computer is foremost in importance to developing and understanding a timeline of events on that machine. Additional temporal information may also be found on the computer, including timestamps embedded in documents, HTTP cookies, and email headers. The MAC times themselves may show time data from other computers, as files downloaded from other sources may carry the timestamps from the computer on which they were created.

When temporal information from a remote host is found on a local computer, a forensic investigator needs to determine the correlation between the time on the local and remote machines. Although he cannot make a comparison between the local and remote hosts at some point in the past, he can take multiple careful measurements of the time on the remote host in order to determine how that computer's clock performs in the present, and then use this information to build a model of the clock's behavior (Stevens 2005). If a very precise model of the clock's behavior can be constructed, the investigator can form an educated hypothesis about the clock's past behavior.

One method of determining the current system time on a remote host is to send it either an ICMP timestamp request or a packet with IP timestamp options set and then

parse the response. If the local host is tightly synchronized with standard time, the response can be used to determine how close to standard time the remote clock is. Even if the local host is not synchronized, this method may be used to determine the correlation between timestamps generated by the local and remote machines. The program *clockdiff*, which is a component of the Linux *iputils* package, uses this method to calculate the system time difference between two computers. Having used this program to measure the time difference between a local host and very many servers across the Internet, we have observed that it is exceedingly slow; also, when the system times on the local and target host differ by more than 12 hours, it generates output that does not correspond to the raw timestamp data the program receives.

1.3 Outline of our Work

Knowing that determining the correct system time on remote computers is important to digital investigations, and believing that *clockdiff* is not completely adequate for the task for which it was designed, we have developed an alternative to *clockdiff* called *clockvar*. This program measures the system time on remote hosts significantly faster than *clockdiff* without losing accuracy; furthermore, it displays the delta (system time difference between the hosts) without manipulating the result when the timestamps from the local and target hosts show a difference of between 12 and 24 hours.

In Chapter II of this paper, we examine prior work in the areas of computer timekeeping and digital forensics. After a further discussion of the importance of developing a timeline of digital events, we study how computer clocks work and the factors that affect the accuracy of these clocks. We consider means of synchronizing

computer clocks with standard time sources and several studies on computer synchronization and the measurement of time on remote hosts.

In Chapter III, we examine in detail how *clockdiff* works and highlight the areas in which improvements can be made. We then explain our solution, *clockvar*, which determines the system time on any number of remote computers using the same internet protocols as *clockdiff*, but without the limitations described above. In Chapter IV, we describe the experiments we conducted to test the improvements in the speed of measuring system times on remote hosts without sacrificing the accuracy of these measurements. We also demonstrate situations in which *clockvar* produces a result which more strongly correlates to the raw timestamp data it receives from target hosts than *clockdiff* does. Finally, in Chapter V, we offer our conclusions and discuss some possibilities for future research in this area.

II. Related Work

In order to underscore the relevance of our work to the science of digital forensics, we use this chapter to develop the following concepts: 1) constructing a timeline of digital events is an essential element of digital forensic analysis; 2) time stamped data from remote hosts may play a crucial role in establishing (or disproving) this timeline; 3) forensic investigators have no basis for an assumption that all computer clocks are synchronized with standard time; 4) thus, forensic investigators need reliable tools for understanding the correlation between time on remote hosts and the machine which is the target of an investigation; and, finally, 5) there is significant room for improvement in the current tools for measuring system time on remote hosts.

In this chapter, we provide examples that highlight just how important constructing an accurate timeline of events is to digital forensic investigations. We consider many possible sources of time data available to investigators, including computers that are external to the host or network that is the subject of an investigation. In order to understand why we cannot presume a tight synchronization of computer clocks with civil time, we study how the computer clocks that produce time stamped data work along with factors that limit their accuracy. After a discussion of the most popular protocols for synchronizing computers with a trusted time source, we analyze the results of several studies on the effectiveness of computer synchronization. In the last part of this section, we explore various methods for measuring time on remote hosts, as building models of their clock behavior may prove essential in establishing or confirming a timeline of events. Examining the currently available methods of determining time on

hosts across the Internet motivates the main goal of our research: to improve upon the speed of system time measurement of remote hosts without adversely affecting the accuracy of these measurements, thus providing a more useful tool to assist forensic investigators in building timelines of digital events.

2.1 An Introduction to Digital Forensics

In his technical report, *A Road Map for Digital Forensic Research*, Gary Palmer offers this as a definition of Digital Forensic Science:

"The use of scientifically derived and proven methods toward the preservation, collection, validation, identification, analysis, interpretation, documentation and presentation of digital evidence derived from digital sources for the purpose of facilitating or furthering the reconstruction of events found to be criminal, or helping to anticipate unauthorized actions shown to be disruptive to planned operations" (Palmer 2001).

Constructing an accurate timeline of digital events on a system plays a crucial part in this process of gathering, interpreting, and presenting digital evidence. Stevens states that establishing a timeline may "provide a critical piece of evidence of information relating to the prosecution of involved persons" (Stevens 2005). The reconstruction of events, both in criminal cases and otherwise, is thus one of the main goals of digital forensic investigation. We begin our review of the related work with a discussion of the importance of determining the correct system time on a computer so that events can be ordered in a correct sequence.

2.1.1 The Importance of a Correct Timeline: Police Forensic Investigation

Collecting date and time evidence is often an essential part of digital forensic analysis. This type of evidence is extremely important because it represents a concrete link between the real world and the realm of computer logs and other digital data. Unfortunately, gathering digital time and date evidence is neither straightforward nor guaranteed to yield an accurate result, as we can see from the following legal case analyzed by Boyd and Forster (2004).

In this case, several emails linked to a man in the United Kingdom led police to suspect him of involvement in the electronic transmission of images of child abuse and child pornography. The police then arrested the suspect and seized his computer. The computer crime unit of the local police department recovered an indecent image involving children from the media they had seized. The suspect was charged with the relevant offences and pleaded "not guilty". The defense team hired a computer forensics expert to analyze the digital evidence provided by the prosecution, including a forensic image of the seized computer and the police forensic statement. After the defense completed their analysis, they submitted a report containing allegations that the police had planted evidence on the suspect's computer. Their report claimed that the computer had been used to access the Internet while it was in police custody; in fact, their report cited around 750 records of internet access time stamped during the 6 hours immediately after the seizure. The accessed sites included one that may have displayed indecent images depicting children. The defense team alleged that the computer had been "altered" while in police custody, and that the police had planted the indecent image on the computer (Boyd and Forster 2004).

Boyd and Forster point out that, when conducting a forensic analysis of a computer, it is important to know whether the timestamps on a system reflect the local time or have been converted to a standard time such as Coordinated Universal Time (UTC). In this particular case, the seized computer's time zone was set to Pacific Standard time (GMT +480 minutes). This information was readily available to both the prosecution and the defense. The registry contained the entry:

```
ActiveTimeBias        REG_DWORD        0x000001e0
StandardName          REG_SZ           PacificStandardTime
```

Although the defense's expert had extracted this registry data, he neglected to configure the forensic analysis software (which was used to analyze activity involving internet access) so that it subtracted the 8-hour time difference. Thus, the report that this software package generated failed to account for the difference between the actual local time and the system time on the computer being analyzed. After the prosecution analyzed the defense report, they discovered the error and issued their own report explaining this facet of the evidence. The defense was then forced to retract their allegations that the police had planted evidence, and the defendant shortly thereafter pleaded guilty to the charges. The authors conclude, "From an ethical viewpoint this case has shown the importance of establishing exactly what is happing forensically before anyone, prosecution or defense, commit themselves to a line of reasoning or a strong opinion" (Boyd and Forster 2004).

2.1.2 The Importance of a Correct Timeline: Analysis of the 2003 Blackout

Digital forensic investigative techniques can be applied in situations other than those imagined by Palmer. On August 14, 2003, a power grid failure occurred in eight states and in the province of Ontario, Canada, disrupting electric service to over 50

million people. Shortly after this massive blackout, U.S. Energy Secretary Spencer

Abraham revealed the complexity of the initial investigation, stating that thousands of

events related to the blackout occurred across the network within a time span of only nine

seconds. Early on in the investigation, North American Electric Reliability Council

(NERC) President Michehl R. Gent anticipated that it would take between 15 and 30

NERC specialists several weeks to analyze the data collected from every component of

the grid that lost power, thus enabling them to reach a conclusion about the causes of the

power failure (McAlpin 2003).

One of the primary purposes of the investigation following this incident was to

determine the specific causes in order to prevent similar outages in the future. This task

was exceedingly difficult due to the initial inability of power operators to determine the

timeline of events after the failure. The investigators had to calculate the time of each

individual event and relate it to the authoritative time kept by an atomic clock.

Unfortunately, due to the lack of synchronization of the all the pertinent system clocks, it

actually took the investigators several months to construct an accurate timeline of events

(Symmetricom 2004).

2.2 Constructing a Timeline of Digital Events

The examples provided above are intended to highlight just how crucial a part the

construction of an accurate timeline of events plays in digital forensic investigations.

Willasen points out that the chief end of most investigations is to identify the person or

persons directly responsible for the crime or incident. Finding the exact times when

various events have occurred is often a critical part of the investigation, especially in

cases when a host or network is attacked from the outside, as internet addresses are frequently assigned dynamically. Building an accurate timeline by pinpointing the precise times of events thus allows an investigator to determine which computer was using a particular IP address at a certain point in time (Willasen 2008).

Having established the importance of developing a timeline of digital events in a forensic investigation, we now explore some of the ways in which time stamped data on a system can be used to determine such a timeline. Carrier and Spafford define a process for reconstructing the relevant events within a digital crime scene. This process focuses on identifying events as causes or effects of other events, to the end that the sequence of events, called an event chain, can be determined. They point out that knowing the actual time of a particular event is the easiest way to place the event in its correct position within the larger event chain. MAC times of files involved in an incident provide a wealth of information that contributes to the understanding of cause and effect relationships among events. While the accessed time does not prove that an object played a particular role in an event, the modified and changed/created times definitively show that a file object is the effect of some previous event (Carrier and Spafford 2004).

Chow et al. confirm that analyzing the MAC times of data retrieved from a digital crime scene is "a crucial process that carries significant value in the event reconstruction phase" (Chow et al. 2007). They stress that there is a strong correlation between the construction of a digital timeline and established methods of analyzing evidence in traditional investigations. Though a key focus of their process for MAC times analysis is to establish a particular user's role in an incident, they caution that file timestamps may be changed as the result of previewing a file via a tool such as Windows Explorer or

"batch operations" such as automated virus and malware scanning tools. Thus, identifying the last access time of a file does not necessarily prove that a particular user actually accessed or opened it (Chow et al. 2007).

Since often more than one computer may be involved in an incident, Kiernan and Terzi categorize data regarding network traffic, network alarms, and external logging systems as additional sources of time data that can be useful in establishing event sequences (Kiernan and Terzi 2008). Furthermore, Stevens identifies additional sources of timestamps on a computer, such as temporal information embedded in email headers and application files such as Microsoft Word documents. Discovering the source of a timestamp is often not a trivial task, but it is necessary to determine what clock produced each timestamp. For instance, consider the case of a timestamp obtained from web browsing records. Does the timestamp come from the machine on which the web page was viewed, or does it come from the remote server that supplied the web page? Email headers may include timestamps from both the sending and receiving computer as well as from servers through which the email was routed, each having its own system clock. Files on a single computer can even contain timestamps from various sources, including the system clock on that machine and those of other computers, in the case that files have been created or edited on other machines (Stevens 2005). Identifying the sources of all relevant timestamps thus adds a level of difficulty to a digital forensic investigation.

Stevens defines a process for unifying all of the digital events recorded from multiple sources into a single timeline. Each piece of digital equipment likely has its own clock, so once an investigator has identified all of the machines involved in an incident, she must determine which time data are provided by each clock. Then, because the

stability and predictability of computer clock performance varies widely (as we examine shortly), she needs to develop a model of the behavior of each clock during the time period of the incident. Although this can be challenging, Stevens notes that having many sources of information can both help to corroborate the timeline developed from a single source and increase the chances that an investigator discover circumstances where the timestamp data have been intentionally manipulated (Stevens 2005).

2.3 An Introduction to Computer Timekeeping

Having established the importance of knowing the correct system time on a computer for the sake of digital forensic investigation, we next consider the worldwide standard for civil time. This is followed by an examination of the relevant aspects of computer timekeeping, including factors that influence the accuracy of computer clocks. This discussion highlights reasons why forensic investigators cannot assume that the machines they analyze maintain synchronization with standard time, thus motivating our work in providing a useful tool for correlating time on multiple computers.

2.3.1 The Standard: Coordinated Universal Time

Coordinated Universal Time, or UTC, is the worldwide standard for civil time, and this standard serves as the basis for how system time is measured on computers. UTC is kept by several laboratories across the world, such as the U.S. Naval Observatory. This laboratory keeps track of time using a very precise atomic clock. As defined by the International System of Units in 1967, one second is equivalent to the time it takes for 9,192,631,770 transitions to occur between two energy levels in the ground state of the cesium 133 atom (Taylor and Mohr, 2000). UTC is accurate to about one nanosecond (a

billionth of a second) per day. The time kept by atomic clocks is distributed via Global Positioning System (GPS) satellites and radio stations such as WWV and WWVH (USNO, 2007). Ideally, all computer clocks would maintain synchronization with UTC; if they did, this would greatly simplify the forensic investigator's task of developing an accurate timeline of events after a systems failure, an intrusion, or other incident.

2.3.2 How Computer Clocks Work

Computers have the ability to maintain time while they are switched off via a battery powered Real Time Clock (RTC), which may or may not be synchronized with civil time. This is an independent chip on the computer's motherboard; as it is frequently accessed via the BIOS, it is sometimes called the BIOS clock. Once the computer boots, the operating system determines how to interpret the RTC. The operating system may maintain a software clock, frequently referred to as the system clock, which is initialized from the RTC at startup and in many cases (especially in Unix systems and Windows 2000 and newer systems) is updated via interrupts from the RTC timer (Stevens 2005, Schatz et al. 2006).

The primary components of a clock are an oscillator and a counter. The oscillator's purpose is to produce a consistent frequency, and the counter counts the oscillator's pulses and renders them in a common time unit. Counters are generally considered very reliable in that they can consistently convert the oscillator's pulses into time units with nearly 100% accuracy. The only difficulty with regard to a counter is to set it to the correct "zero-point" so that it is in agreement with other clocks. Some form of standard time – typically UTC for computer clocks – is used as a foundation for this agreement (Symmetricom 2003).

2.3.3 Limitations on the Accuracy of Computer Clocks

Though a clock's accuracy is dependent on the extent to which its oscillator behaves in a stable and predictable manner, this stability varies widely based on the type of oscillator employed in the clock. Highly accurate oscillators include the Earth's rotation (used for astronomical time) and cesium and rubidium energy transitions (used in atomic clocks). The most common oscillators used in computer clocks, however, are inexpensive piezoelectric quartz crystals. They tend to be far less accurate due to the fact that they are not nearly as stable and predictable as the atomic clocks described above. While these crystals are designed to vibrate at a frequency of 32,768 Hz, several factors – including the crystal's size, cut, and orientation – typically cause the crystals to oscillate faster or slower than this frequency. This condition of running faster or slower than the intended rate is called clock skew. The actual frequency of oscillation is also dramatically affected by the crystal's temperature, and to a lesser degree by other environmental factors such as magnetic fields and mechanical vibrations. Due to these variations, clocks with crystal oscillators can drift away from standard time (that is, move faster or slower than standard time) by up to several seconds per day, and this drift can become quite significant over time (Symmetricom2003).

With the goal of building a clock model that relates time stamps on a computer to actual time, Stevens categorizes the issues that affect clock behavior into four major categories: time zone, time zone variations, clock drift, and finally clock error and adjustment. While the time zone represents a fixed offset from standard time (up to twelve hours before and after UTC), daylight savings time and other adjustments within a time zone can cause the actual local time to diverge from standard time by some

additional part of an hour. The rate at which a clock inherently drifts away from standard time is likely unique to the particular clock. Clock error – the condition of the clock being set to the wrong time – can be introduced by a number of causes, including synchronization to an imprecise clock, an accumulation of error due to clock skew, and a user who either accidentally or purposely sets the clock to the wrong time. Stevens points out that users who wish to manipulate the time stamps on files may not only simply change the system time, but also use hex editors and disk partition editing programs (Stevens 2005). Therefore, the correlation between a computer clock and civil time is an important factor for forensic investigators.

Due to the fact that maintaining synchronization with standard time is highly desirable for a variety of reasons, and because computer clocks are inherently limited in their ability to keep highly accurate time, many professional system administrators make use of tools to keep their system clocks synchronized. The most widely used method for synchronizing computer clocks running Unix systems (and related operating systems) with standard time is Network Time Protocol (NTP) (Schatz et al. 2006). We examine this protocol in the next section.

2.4 System Clock Synchronization via NTP

NTP was developed with the goal of maintaining a redundant pool of highly accurate, trusted time sources closely synchronized to standard time, and then distributing this time to hosts across the Internet. The distribution of time follows a hierarchical arrangement, with the top level being the servers that are directly connected to sources of standard time such as atomic, radio, or GPS clocks (Mills 1992). These are called

"stratum 1" time servers; higher strata numbers indicate the level in the hierarchy with which a particular host maintains synchronization. Stratum 2 hosts utilize stratum 1 servers to synchronize themselves; stratum 3 hosts utilize stratum 2 servers, and so on. According to the NTP.org website, as of January 2009, there are 228 stratum 1 and 314 stratum 2 servers functioning across the Internet.

The NTP network consists of computers that can be classified into three categories: primary servers, secondary servers, and client machines. Primary servers are the ones that maintain a direct connection to a trusted time source (i.e., the stratum 1 servers). Secondary servers (i.e., stratum 2 and higher) act both as clients to the primary servers and as distributors of NTP time to their own clients. Any host running NTP that becomes synchronized with an NTP server can itself become an NTP server for peers (NTP hosts at the same stratum) or higher level hosts (i.e., hosts at strata further away from stratum 1). Client machines are merely consumers, but not providers, of the NTP service. Hosts running NTP can expect to achieve synchronization to a trusted time source to within 1 to 50 milliseconds (Mills 2006).

There are three protocol variants that can be used to achieve synchronization. In the client/server mode, a client initiates the process by sending NTP messages to a server and uses the data within the responses to adjust its system clock to conform with UTC. In broadcast client / broadcast server mode, a server initiates the process by broadcasting synchronization messages to its clients. In peer mode, each machine acts as a client of the other; thus, each both pushes synchronization to and pulls synchronization from the other (Mills 2006).

In order to better grasp the process of acquiring and maintaining synchronization with a source of standard time, it is helpful to analyze the data that is transmitted between servers and clients. As shown in **Figure 1,** the NTP packet header consists of a minimum

0	2	5	8	16	24	31
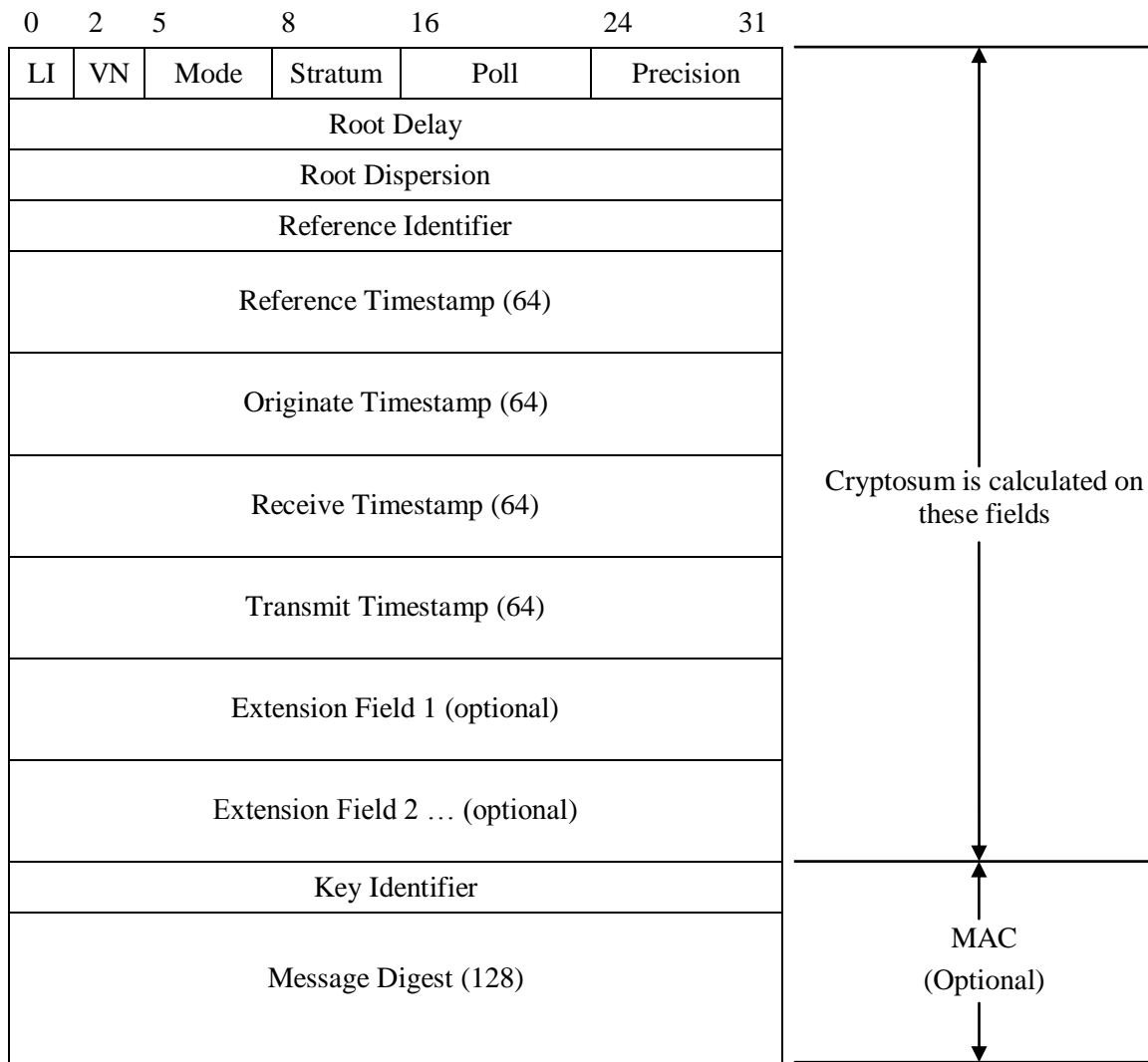
Figure 1: NTP Packet Header Format

of 48 octets (or twelve 32-bit words), primarily consisting of 4 octet (32-bit) and 8 octet (64-bit) timestamps. The NTP packet header immediately follows the UDP and IP headers. All data within the header are interpreted as integer values. The first word of an

NTP packet consists of 6 different fields and starts with a 2-bit Leap Indicator, the value

of which can indicate any of the following conditions:

0	No warning (i.e., a normal NTP message).
1	The last minute of the day will contain 61 seconds.
2	The last minute of the day will contain 59 seconds.
3	Alarm condition: the system clock has never been synchronized.

The 3-bit NTP version number, the current being 4, follows this field. The 3-bit mode

field comes next, representing the mode of operation, including symmetric active /

passive, client, server, and broadcast. Following the mode is the 8-bit stratum field

representing the level (between 0 and 255) the server occupies in the hierarchy: 0

represents an unspecified or invalid stratum, while 1 indicates a primary server (Mills

2006).

Next comes the 8-bit poll value, which indicates the span of time (in $\log_2$ seconds)

that will elapse before the next exchange of synchronization messages. This value falls

between 16 seconds and 36 hours. The fourth octet contains a signed integer that

characterizes the precision of the system clock in $\log_2$ seconds, and it is calculated by

timing a series of measurements of this clock.

The remaining 11 (or more) words in the packet header are all timestamps of one

type or another. NTP utilizes two different timestamp formats in packet headers during

the synchronization process, shown in **Figure 2** below. The 32-bit Short Format is used

in measuring round trip time and computing the error ranges in time measurements. It

contains a 16-bit value representing the number of seconds and another 16-bit value

representing the fraction of a second. The 64-bit timestamp format is an integer value

representing the amount of time that has elapsed since the prime epoch (midnight on

January 1, 1970). The first 32 bits hold the number of seconds (up to 136 years), and the

last 32 bits represent the fraction of a second, with a resolution of 232 picoseconds (one

picosecond being 10^{-12} seconds, or one trillionth of a second). All NTP timestamp values

are network byte ordered in big-endian format (Mills 2006).

0	15	16	31
Seconds		Fraction	

NTP Short Format

0	31	32	63
Timestamp		Fraction	

NTP Timestamp Format
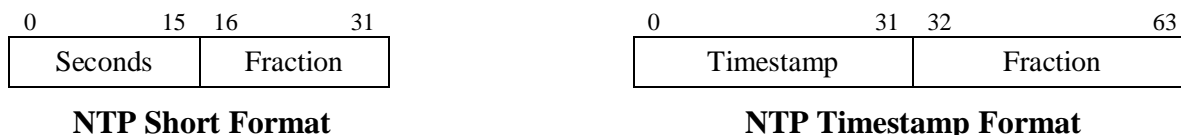
Figure 2: NTP Short and Timestamp Formats

The second word of the packet contains the root delay – a short format timestamp

representing the measurement of the round trip time between the client and server. Next

comes the root dispersion, which is another short format timestamp, this one representing

the maximum possible error range in the measurement. The fourth word contains a

reference identifier, particular to a server or reference clock. Stratum 1 servers are

assigned a unique 4-character ASCII string (left-justified and zero-padded) as their

reference identifier. Some examples include GOES (Geosynchronous Orbit Environment

Satellite), GPS (Global Position System), and PPS (generic pulse-per-second). When the

stratum field contains a zero (invalid or unspecified), the reference identifier consists of a

4-character string called a "kiss code" utilized in debugging and monitoring procedures

(Mills 2006).

Following the reference identifier are four NTP timestamp format fields: the

reference, originate, receive, and transmit timestamps. The time when the client's system

clock was last updated is placed in the reference timestamp field. The originate

timestamp is struck when the packet bound for the NTP server leaves the client machine.

The server strikes the receive timestamp when the client's packet arrives and the transmit timestamp when it sends the response to the client. A fifth timestamp is struck by the client upon the arrival of the server's response. Though this timestamp is not part of the packet header, it becomes part of the packet buffer data structure which is processed by the client (Mills 2006).

The optional extension fields, if included in the packet header, are utilized by the Autokey security protocol (Mills 2006), which is beyond the scope of this thesis. If these fields are used, then a 32-bit key identifier (which designates a secret 128-bit MD5 key) and 128-bit MD5 message digest must follow. The message digest is calculated on all the required fields and optional extension fields in the packet header (Mills 2006).

The NTP protocol can be broken down into five distinct processes: poll, peer, system, clock discipline, and clock adjust process. The poll process governs the transmission of messages from a client to an NTP server or other source of standard time, including the frequency at which this contact is initiated. The peer process involves receiving responses (either from a peer, a server at lower level stratum, or directly from a reference clock) and then interpreting this data. As NTP was developed, a great deal of consideration was given to the fact that some clocks on the network might not keep very accurate time, but advertise that they do. The terms *truechimer* and *falseticker* apply respectively to clocks that can be trusted and those that cannot. A large portion of the NTP specification deals with the system process: algorithms that are employed to ensure that a host becomes synchronized with a truechimer. The clock discipline process controls both the time and frequency of the system clock on a client machine, and the

clock adjust process helps to maintain a consistent frequency by generating a computed

offset from the reference time once per second (Mills 2006).

The flow of these five processes can be summarized in this manner. A client host

running this protocol initiates contact with an NTP server on a schedule determined by its

poll interval, which is some span of time (2^t seconds) ranging from 16 seconds (2^4) to 36

hours (2^{17} seconds). The host strikes the reference and originate timestamps, and then

sends an NTP message to one or more NTP servers. The server responds with a message

that provides the client with the correct offset – that is, the amount the local clock should

be adjusted in order to become synchronized with the standard time provided by the

server. The offset is calculated from the receive and transmit timestamps (struck by the

server) and the root delay and root dispersion. The client adjusts its system clock via

system calls such as (Unix) `settime()` or `adjtime()`. The message may also provide

data that the client may use to choose the best source from multiple time servers. A

series of messages are used so that the server can calculate the round trip delay and send

its message so that it will arrive at the client at a specific time. Mills notes that a host can

initially become synchronized with a trusted source very quickly, but many careful

measurements are required over an extended period of time in order to determine the rate

at which the local clock drifts from standard time. This is done so that the NTP daemon

running on the local machine can calculate its poll interval – that is, schedule when it

needs to contact the trusted time server in order to maintain synchronization to the

millisecond (Mills 1992).

2.5 System Clock Synchronization via SNTP

Another popular method by which a host can be synchronized to a time server is the Simplified Network Time Protocol (SNTP). This is, in fact, a simplified version of NTP. The basics of the protocols work the same, the packet formats are the same, and SNTP algorithms to calculate the client time, clock offset, and round trip delay work just as they do in NTP. The primary difference is that SNTP clients typically synchronize with only one trusted source rather than consulting multiple time servers in order to determine the best source. The second major difference is that SNTP clients, according to the SNTP specification, are not intended to serve as reference sources for additional clients (Mills 2006).

Starting with Windows 2000, Microsoft systems have had the capability of synchronizing with a trusted time source via SNTP. For example, Windows XP systems are set by default to synchronize with the server `time.windows.com` once a week. Because SNTP does not employ NTP's clock discipline algorithms, and because the synchronization occurs only once per week, the system clocks on Windows systems can be expected to drift further away from civil time than UNIX hosts utilizing NTP (Schatz et al. 2006).

2.6 Studies on Synchronization and the Measurement of System Time

We have analyzed how computer clocks work, the characteristics which make them susceptible to deviate from standard time, and the primary tools that many systems administrators employ to achieve synchronization with standard time. In this section, we examine several studies on computer clock synchronization, including hosts that do and

do not employ NTP or SNTP. These studies reveal that employing a synchronization protocol neither guarantees that a host keeps accurate system time nor makes it easy for an investigator to fully understand clock behavior on a remote host. Our analysis further emphasizes the need for practical and accurate tools for correlating a remote host's system clock to standard time.

2.6.1 An Early Survey of the Accuracy of the NTP Network

In a study of the accuracy of the NTP network in 1999, Minar commented, "As more distributed systems are built across the Internet, the quality of the Internet's time synchronization is becoming more significant" (Minar 1999). Unfortunately, his study revealed a "surprising number of bad timekeepers" among the stratum 1 clocks he surveyed. As discussed above, these stratum 1 clocks serve as the reference for the entire NTP network.

Minar estimated that (at the time of his survey) the NTP network consisted of over 175,000 hosts. He identified 907 servers operating as stratum 1 clocks, but he was shocked to discover that only 254 (28%) were keeping accurate time. 638 (70.3%) of these machines were configured to use the local system clock (not a trusted source of standard time) as their reference clock, and that 391 (43.1%) of these stratum 1 servers deviated from standard time by more than 10 seconds. One, in fact, was over 6 years off. Through this survey, he discovered that the Red Hat Linux version of the NTP software had been distributed with the local system clock configured as stratum 0, hence the source of so many machines referencing the local clock as opposed to a trusted source of standard time (Minar 1999). Although this software error has been since corrected, the

study does call into question the accuracy of system clocks that are supposed to be synchronized with standard time.

2.6.2 Observing Clock Skew While Measuring Packet Transit Times

In a study aimed at carefully measuring packet transit times on a network, Paxson discovered that clock skew was a frequent problem. Even when two hosts on a network were tightly synchronized, the differing rates of skew of their system clocks caused it to appear that the network delays experienced by the packets were shrinking and growing though the actual delay remained fairly stable. In fact, when the hosts were synchronized using NTP, the packet transit times sometimes appeared to be very inconsistent. Paxson concluded that the local clock adjustments (made when the NTP server would tell each host to apply a certain offset to bring it back into synchronization), not varying network delays, were the source of the inconsistencies. Thus, we can reason that even employing NTP cannot guarantee that a host exhibits accurate and predictable clock behavior over an extended time (Paxson 1998).

2.6.3 Time Synchronization on Various Operating Systems

Kohono et al. tested the installations of many popular operating systems and concluded that most are configured by default either to synchronize with a trusted time server infrequently or not to do so at all. Windows XP Professional systems do contact Microsoft's NTP server when they boot up, but they maintain synchronization with this server only once a week subsequently. While Red Hat 9.0 Linux systems allow the user to specify an NTP server, they are not configured to use NTP by default. Under the "typical user" configurations of OpenBSD 3.5, FreeBSD 5.2.1, and Debian 3.0 Linux

installations, the ntpd service may not even be enabled by the user (Kohono et al. 2005).

We can deduce that, even though protocols for synchronizing with an accurate time

source exist, a large number of hosts on the Internet likely do not perform consistent or

frequent synchronization with a trusted time source.

2.6.4 Time Synchronization Across a Network

Schatz et al. studied how me might use a "commonly logged corroborative

source" to determine the behavior of another host's system clock. They studied a small

business network consisting of a Windows 2000 server (the domain controller), several

Windows 2000 and XP workstations, and a Linux machine serving as a firewall between

the network and the Internet. The firewall was running NTP, but the domain controller

performed no synchronization with a reliable time source, and thus continued to drift

further from civil time (from around 8 to around 10 seconds) throughout the course of the

experiment. The Windows workstations were configured to perform synchronization

with the domain controller via SNTP. Over the course of a month, the authors frequently

sampled the system time on each host and compared it to the firewall's interpretation of

civil time (as this machine maintained synchronization with a stratum 2 NTP server).

They found that, in most cases, the Windows machines on the network maintained fairly

close synchronization with the domain controller (via SNTP); thus, each machine drifted

away from civil time at about the same rate as the unsynchronized domain controller.

One conclusion of their work is that system clocks tend to drift from civil time at a linear

rate. Due to the several anomalies they encountered, however, they also concluded that it

is very difficult to make authoritative statements about the behavior of system clocks

within a Windows domain (Schatz et al. 2006).

In the above study, the authors revealed several factors that influenced computer clock accuracy, even those that are supposed to be synchronized with a trusted time source. They found that the RTC (BIOS clock) on many of the Windows systems they surveyed was not set correctly. Often, the time zone was set not to local time, but to the default installation time zone. They claim that SNTP is only capable of maintaining synchronization to "within 2 seconds in a particular site and 20 seconds within a distributed enterprise." Furthermore, they state that unless hosts running NTP and SNTP are configured to use cryptographic authentication, they are vulnerable to attacks based on these protocols. (The latest SNTP specification does recommend employing cryptographic authentication (Mills 2006).) Finally, they caution that "software errors in the implementation of software clocks or the timestamp serialization algorithm have the potential for adversely affecting timekeeping accuracy" (Schatz et al. 2006).

2.7 Measuring System Time on Hosts Across the Internet

We have seen that, though protocols such as NTP and SNTP are widely available, it cannot be assumed that a particular host (say a web or email server) on the Internet maintains synchronization with standard time. If this server is the source of timestamped data found on a computer involved in an incident, a forensic investigator needs to be able to correlate the system time on this server with standard time. In this section, we examine methods of measuring the system time on remote hosts.

2.7.1 Sources of Internet Timestamps

Zander and Murdoch have developed several techniques for estimating the clock skew of hosts across a network or the Internet. Their work includes developing methods

of eliminating errors in measurements from sources such as "network jitter", which is the variability in packet transit times across a network, due to factors such as unpredictable and asymmetric paths through the network, or collisions during periods of high traffic. Their research demonstrates that quantization error (the difference between real time and a computer clock's approximation of real time) has the greatest effect on measurements of time on another host. Thus, the frequency of the clock that generates the timestamps is an important factor in the accuracy of the measurement (Zander and Murdoch 2008).

They cite four sources of timestamps from such hosts: ICMP, TCP, and HTTP packet headers, as well as TCP sequence numbers. They conclude that TCP sequence numbers, which are generated by summing a 1MHz clock and a cryptographic function, work well for approximating a target's clock skew only for a short duration, as the function is rekeyed every five minutes. ICMP timestamps may be measured for any given duration; however, they are less accurate than TCP sequence numbers (their frequency is only 1kHz). Furthermore, they (along with other ICMP traffic) are often blocked by firewalls, and they may introduce an element of inaccuracy in that the system clock of a host running NTP may be adjusted by that protocol in between the time that a timestamp request arrives and its response is generated. The frequency of TCP timestamps is dependent upon the operating system (if the OS supports them), and it ranges from 1 Hz to 1 kHz. Zander and Murdoch consider utilizing TCP timestamps the most effective method for measuring time on a wide range of remote hosts, even though they cannot be used in conditions such as the Tor anonymisation network (a major focus of their recent study), as this network does not provide an end-to-end TCP connection

between hosts. HTTP timestamps are generated by all web servers, but have a frequency of only 1 Hz (Zander and Murdoch 2008).

As quantization noise has the most detrimental effect on the accuracy of time measurements on remote hosts, Zander and Murdoch developed a technique for reducing this factor by synchronizing the timestamp measurements with the tick of the system clock on the target host. They present an algorithm for determining the target's clock frequency and adjusting the probe interval (the amount of time before the next packet is sent to the target) such that each timestamp from the target comes virtually right after the target's clock tick, and thus contains the lowest possible quantization error. Their study demonstrates that this type of synchronized sampling is possible using each of the aforementioned timestamp sources, and that it achieves a significant reduction of quantization error over random (non-synchronized) measurements (Zander and Murdoch 2008).

2.7.2 A Large Scale Study of Time Synchronization Across the Internet

Apart from Minar's analysis of the NTP network, each of the studies discussed above has focused on a relatively small number of hosts. In this section, we explore a survey of the clock behavior of a large numbers of computers across the Internet using multiple methods of time measurement.

Buchholz and Tjaden conducted a large-scale study of the degree to which over 8,000 servers on the Internet maintained synchronization with standard time over a six month period. Goals of this study included gathering data on what percentage of hosts connected to the Internet are synchronized to standard time, collecting data useful to assembling a description (or model of the behavior) of a computer's system clock, and

exploring methods of measuring the system time on remote hosts. As various other authors have discussed, they reiterate the necessity of being able to understand the clock behavior of a remote computer, since the forensic investigation of even a single host is likely to yield timestamps introduced by external sources. A full understanding of the clock behavior of these external hosts can be very helpful in either establishing or confirming a timeline of events on the local computer (Buchholz and Tjaden 2007).

The authors used the DMOZ Top Listed Domains website as the source for choosing servers across the Internet whose clocks they might sample. Using DNS to resolve the 8,329 domain names they gathered, they compiled a list of 8,410 unique IP addresses. They wrote a program called web-time to collect HTTP timestamp data from the servers, 90% of which responded regularly with a valid timestamp. Using this tool, they discovered that around 74% of the servers were synchronized to within 10 seconds of standard time (UTC). Of the remaining servers, around 41% were between 10 seconds and 24 hours ahead of standard time, while 59% ran slower than standard time, being behind by an average of 21 days. Discarding the 2 clocks reading farthest in the past (which were off by a century), the average was about 3 ½ hours behind standard time (Buchholz and Tjaden 2007).

Interested in comparing these results with another method of measuring time, they surveyed the same 8,410 hosts with *clockdiff*, using each of the three options available with this program. Because of the considerable time consumed by *clockdiff*'s evaluation of each target, the initial survey of the same hosts took several days. 4,413 of these computers responded to at least one of the options, and thus only these hosts were included in the daily analysis, each round taking about 18 hours to complete. The results

from the *clockdiff* experiment also showed that 74% of the hosts kept reasonably accurate time to within 10 seconds of UTC. Due to the limitations of the IP and ICMP timestamp fields, and possibly due to inaccurate results reported by *clockdiff*, the other 26% of hosts showed time differences averaging 10 minutes behind and 12 minutes ahead of standard time, with the greatest differences being only 11 hours behind and nearly 12 hours ahead (Buchholz and Tjaden 2007).

Comparing the performance of the two tools, the study showed that web-time and *clockdiff* generally yielded results that were consistent with one another. The delta between the two measurements was within 10 seconds for 95% of the hosts surveyed (92% of the deltas were less than 1 second). Thus, for 5% of the hosts (187 of the 3,714 which responded to both methods), the disagreement between web-time and *clockdiff* was greater than 10 seconds. It is this discrepancy, along with our subsequent examination of *clockdiff*'s source code, which has stimulated our interest in testing the accuracy of *clockdiff*. One of the conclusions reached by Buchholz and Tjaden is that "additional tools are needed for measuring time on a remote system over the Internet … Additional methods of sampling a remote clock may be able to perform better measurement or at least give us additional evidence about the time on a remote system when performing a forensic analysis" (Buchholz and Tjaden 2007). In Chapter III, we show why *clockdiff* is not entirely adequate as a tool for assessing the system time on a remote host and discuss ways in which its underlying method of time measurement can be improved.

III. Problem Definition and Solution

We have pointed out that multiple computers may be involved in an intrusion, accident, or other incident. When conducting a forensic investigation of an incident, an investigator may discover digital evidence, including timestamps, from a variety of sources on a network, even across the Internet – this would certainly be true in cases such as email headers or HTTP data from a web server. In order to use these timestamp data to help establish a timeline of events, the investigator will need to fully understand the correlation between the digital timestamps from the various sources and a standard time such as UTC. It is likely that he will have to make multiple comparisons between the system time on these remote hosts and a standard time source in order to build a model of clock behavior for each remote clock that can contribute to the accuracy of the timeline he is attempting to establish.

One method he might use is to synchronize a computer with a trusted time server via NTP, and then use a tool such as *clockdiff* to sample the system time on each computer he is interested in. We have seen from the study conducted by Buchholz and Tjaden that, when *clockdiff* and *web-time* (i.e., parsing the HTTP headers) are used together to determine the system time on a remote host, these two methods do not always produce consistent output. We believe that, in many of these instances of inconsistency, *clockdiff* is actually reporting inaccurate results. Our discussion of this problem reveals that *clockdiff's* process is needlessly time-consuming and only accurate when the two hosts are roughly synchronized. We then present *clockvar*, a tool that we developed to measure time on remote hosts with greater speed and without data adjustments that are

opaque to users, and our own version of *web-time*, which we used to corroborate our findings when *clockdiff* and *clockvar* disagreed on the time difference between two hosts.

3.1 Problem Definition: *Clockdiff's* Slowness and Possible Inaccuracy

Because *clockdiff* uses many packets in an attempt to calculate the network delays in each direction, it takes a considerable time to produce a result. We determined by experiment that *clockdiff* takes an average of around 11 seconds to process one target host. In addition to being unnecessarily slow, we offer evidence that *clockdiff* does not always generate a result that is in accord with the raw timestamp data it receives. In order to demonstrate why we lack confidence in *clockdiff's* output, we examine how it works in detail, including the Internet protocols on which it is founded and the methodology employed to calculate the time difference between the local and target hosts. We conclude the problem definition with the reasons for which we suspect *clockdiff* may generate inaccurate results.

3.1.1 Internet Timestamps

Before examining the inner workings of *clockdiff* in detail, we discuss the message formats utilized by the program. The default mode uses ICMP timestamp requests/replies, and additional methods employ ICMP echo requests with IP timestamp options set in the IP headers.

ICMP timestamp requests and replies have the same format. The first field in an ICMP datagram is an 8-bit field for the ICMP type. Type 13 represents a timestamp request, and type 14 a timestamp reply. This is followed by an 8-bit code, and 16-bit fields for the identifier and sequence number. The actual timestamp data are contained in

the three subsequent 32-bit fields. The first, the Originate Timestamp, is the time the message was last handled by the sender (the host requesting a timestamp) prior to sending it. This information is returned to the sender in the same field when the reply message is sent. The remaining two fields, the Receive and Transmit Timestamps, represent timestamps from the request recipient. The former is the time the recipient first touched the message, and the latter is the time it was last handled prior to sending the reply to the requester (Postel, 1981b).

Each 32-bit timestamp represents the number of milliseconds since midnight UTC. If a host cannot provide a timestamp in milliseconds or with respect to midnight UTC, then it may insert any number into this field, as long as it sets the high order bit of the timestamp to indicate that it contains a non-standard value (Postel, 1981b). Since there are 86,400,000 milliseconds in a day ($24 \cdot 60 \cdot 60 \cdot 1000$), this is the highest value a timestamp should theoretically contain. This number can be represented by a 24-bit number, which means that the timestamp field has space to store the number of milliseconds in 24.8 days (2^{31} milliseconds). However, since a timestamp is the number of milliseconds since midnight UTC, it is not clear how to interpret a value representing a number larger than 86,400,000.

ICMP echo / echo reply datagrams have a similar structure to the ICMP timestamp request/reply datagrams. The header contains the same five fields described above (type, code, checksum, identifier, and sequence number); however, the code is 8 for an echo message and 0 for an echo reply. Instead of fields for timestamps, the data portion of this message contains whatever data the sender wants to be returned (Postel, 1981b).

IP timestamp options follow the standard 20-octet IP header. The first option field, the option type (1 octet), must be set to 68 for IP timestamps. The second octet is the total length of the options (in octets), including the type, length, pointer, overflow/flag, and timestamps; the maximum value is 40. The next field (1 octet) contains a pointer (number of octets) to the space where the next timestamp begins. The next field (4 bits) represents the number of IP modules that were unable to record timestamps on account of a lack of space. The last field (4 bits) prior to the beginning of the timestamps is a flag which signals how the timestamps and the corresponding IP addresses of the hosts that register them (each as 32-bit values) are to be recorded. A value of 0 indicates that only timestamps are to be recorded. A value of 1 indicates that each host which registers a timestamp should also record its IP address; the IP address is recorded in the first four octets of a pair, and the timestamp in the second four octets. With this option, there are room for up to four pairs of IP addresses and timestamps (the fifth "pair" is used by the host originating the timestamp request). A flag value of 3 indicates that only the IP addresses (within the options portion of the IP header) specified by the originating host may record timestamps. If the packet is routed through other hosts whose addresses are not specified, they forward the packet to the next hop without registering a timestamp (Postel, 1981a).

3.1.2 Packet Formats Used by *Clockdiff*

Clockdiff, which is based on code from the BSD timed daemon and compiled by Dr. Alexey Kuznetsov, is part of the Linux iputils package. It is used to determine the difference in time between the local host and one other remote host. *Clockdiff* is invoked

from the command line by entering: `clockdiff [-o] [-o1] <destination>`, with
`<destination>`, being a fully-qualified URL or IP address.

 Clockdiff supports two different methods for obtaining a timestamp from the
target host. The default option sends a series of fifty ICMP timestamp requests to the
target host. The other method (invoked with `-o` or `-o1` arguments) involves sending fifty
ICMP echo requests with the IP timestamp option selected in the IP header (RFC 791).

 The ICMP timestamp message used by *clockdiff* is 20 octets in length and is
constructed as shown in **Figure 3**.

type = 0x0D (13)	code = 0x00	checksum
identifier = *clockdiff's* process ID		sequence number
local host's originate timestamp (32 bits, network byte order)		
0x00000000 (all zeroes; space for target host's receive timestamp)		
0x00000000 (all zeroes; space for target host's transmit timestamp)		

Figure 3: *Clockdiff*'s ICMP Timestamp Packet

The first of two IP options uses four-term specified hop addresses, while the other uses
three-term specified hop addresses. The IP options portion of the IP header for the four-
term specified hop addresses is 36 octets in length and looks like this:

code = 0x44 (68)	length = 0x24(36)	pointer = 0x0D (13)	oflw/flag = 0x03
local host's address (32 bits, network byte order)			
local host's originate timestamp (32 bits, network byte order)			
target host's address (32 bits, network byte order)			
0x00000000 (all zeroes; space for target host's timestamp)			
target host's address (32 bits, network byte order)			
0x00000000 (all zeroes; space for target host's timestamp)			
local host's address (32 bits, network byte order)			
0x00000000 (all zeroes; space for local host's receive timestamp)			

Figure 4: *Clockdiff*'s 4-Term Specified IP Options Packet

The IP options portion of the IP header for the three-term specified hop addresses is 28 octets in length and looks like this:

code = 0x44 (68)	length = 0x1C(28)	pointer = 0x0D (13)	oflw/flag = 0x03
local host's address (32 bits, network byte order)			
local host's originate timestamp (32 bits, network byte order)			
target host's address (32 bits, network byte order)			
0x00000000 (all zeroes; space for target host's timestamp)			
local host's address (32 bits, network byte order)			
0x00000000 (all zeroes; space for local host's receive timestamp)			

Figure 5: *Clockdiff*'s 3-Term Specified IP Options Packet

3.1.3 How *Clockdiff* Obtains the Time Difference Between Two Computers

When *clockdiff* is invoked against a target host, it sends the target a series of ICMP timestamp request messages (or ICMP echo messages with IP timestamp options set) and then measures the delta, or time difference, by parsing the responses. The following pseudo code illustrates the process *clockdiff* uses when sending either an ICMP timestamp request or echo request with IP timestamp option:

```
for (1 to 50)
begin loop
        Originate timestamp = gettimeofday() % 86,400,000
        send timestamp request to target host
        receive reply
        recvtime = gettimeofday() % 86,400,000
        rtt = recvtime – Originate timestamp
        if (Receive timestamp & 0x80000000 != 0)
                exit loop; report non-standard timestamp format
        end if

        delta1 = Originate timestamp – Receive timestamp
        if (delta1 == min(delta1)
                store this delta1
        else
                discard delta1
        end if

        delta2 = recvtime – Transmit timestamp
        if (delta2 == min(delta2)
                store this delta2
        else
                discard delta2
        end if

        if (delta1 < -43,200,000)
                delta1 += 86,400,000
        else if (delta1 > 43,199,999
                delta1 -= 86,400,000
        end if

        if (delta2 < -43,200,000)
                delta2 += 86,400,000
        else if (delta2 > 43,199,999
                delta2 -= 86,400,000
        end if

        measure_delta = (delta1 – delta2) / 2
end loop
output: host name, average rtt, minimum rtt, measure_delta, ctime()
```

The details of the process *clockdiff* executes are laid out in the 16-step process below. The first 15 of 16 steps are performed fifty times (as a series of fifty packets is sent to the target host), and the last step produces the output seen by the user:

1. The current system time on the local host is obtained using `gettimeofday()`.

2. The result (a timeval struct containing the number of seconds and microseconds since the epoch) is converted to the number of milliseconds since the epoch.

3. This number is divided modulo 86,400,000 milliseconds (24 hours), resulting in the number of milliseconds since midnight UTC (in essence, a timestamp from the local host).

4. This timestamp is placed in the field for the "Originate" timestamp in the outgoing ICMP message, and the packet is sent to the target host.

5. When the response from the target host is received, *clockdiff* again obtains the current system time on the local host using `gettimeofday()`.

6. The result is also divided modulo 24 hours and is stored as the variable `recvtime`.

7. The data in the Originate field (the local host's system time just prior to sending the request) are stored as the variable `sendtime`.

8. The round trip time (rtt) is calculated by subtracting `sendtime` from `recvtime`. As this occurs fifty times, the program stores the shortest round trip time.

9. The data in the "Receive" timestamp field are stored in the variable `histime`.

10. The data in the "Transmit" timestamp field are stored in the variable `histime1`.

11. If the high order bit in `histime` is set, then processing halts, and *clockdiff* reports that the target's timestamp has been sent in a non-standard format.

12. `delta1` is calculated by subtracting `sendtime` from `histime`. This result is the difference in system times on the two hosts, measured by subtracting the local host's time from the target host's time at the point when the target host received the timestamp request. Each time through the loop, the new `delta1` is compared to the previously stored smallest `delta1`. If the new delta1 is smaller, this value is stored; otherwise, the value is discarded.

13. `delta2` is calculated by subtracting `histime` from `recvtime`. This result is the difference in times measured by subtracting the target host's time from the local host's time at the point when the local host received the timestamp reply. As in step 12, the smallest `delta2` value is stored.

14. An adjustment is made under certain circumstances (we pass over this step for the moment).

15. The difference in system time of the two hosts, `measure_delta`, is calculated thus: `measure_delta = (delta1 - delta2) / 2`.

16. The output contains the host name, average round trip time, smallest round trip time, the delta (difference in system time of the two hosts), and the current system time on the local host, which is calculated by a call to `ctime()`.

Consider this toy example to illustrate the process. The local host determines that it is exactly 4 milliseconds past midnight UTC and sends a timestamp request to a target host at that time. The target host receives this request 2 ms later, but determines that it is exactly 7 ms past midnight UTC (thus the target host is exactly 1 ms behind the local host). The target sends this number back in its reply as the receive timestamp. 2 ms

later, the target host receives the reply and determines that its system time is now 8 ms

past midnight UTC. According to the algorithm above, the following are true:

```
sendtime  = 4          recvtime      = 8

histime   = 7          delta1        = 3

delta2    = 1          measure_delta = (3-1)/2 = 1
```

In this example, *clockdiff* correctly reports a delta of 1; that is, the target host is exactly 1

ms behind the local host. Consider, however, what would happen if everything in the

previous example remained the same except that it takes 6 ms for the trip back from the

target to local host. Now these would be the values determined by *clockdiff*:

```
sendtime  = 4          recvtime      = 12

histime   = 7          delta1        = 3

delta2    = 5          measure_delta = (3-5)/2 = -1
```

Now *clockdiff* reports that the same target is 1 ms **ahead** of the local host. Herein lies a

fundamental difficulty in determining the difference in system times between two hosts

on the Internet (or possibly even on the same network): the path a packet takes traveling

from a remote host back to the local host is not necessarily the reverse of the path that a

similar packet takes going from the local host to the target. Not only are network paths

asymmetrical, but also the travel time from host to host is neither consistent nor

predictable from one packet to the next.

Clockdiff attempts to correct for this inconsistency by sending a series of fifty

timestamp requests to the target host. For each iteration of the above processing

algorithm, it determines and stores the smallest travel times between local and target

hosts (in both directions) and ultimately uses these values in step 15. This increases the likelihood that *clockdiff* will report a more accurate result than if only one timestamp request were sent.

3.1.4 Situations in Which *Clockdiff* Generates Questionable Results

While all of this makes sense and appears correct, a fundamental flaw of *clockdiff* rises from the assumption that the two hosts are synchronized to within twelve hours of one another. The following procedure takes place in step 14 of the processing algorithm above:

> If `delta1` (or `delta2`) is less than -43,200,000 (half the number of milliseconds in a day, or about twelve hours), then the value 86,400,000 is arbitrarily added to this number. Likewise, if `delta1` (or `delta2`) is greater than 43,199,999, then the value 86,400,000 is arbitrarily subtracted from this number.

The author's comments in *clockdiff*'s source code explain the reason for this adjustment: "Handles wrap-around to avoid that around midnight small time differences appear enormous. However, the two machine's clocks must be within 12 hours from each other." While well-intentioned, we do not believe that it is correct to make this adjustment in all circumstances. It would seem correct in the following case: Host A (the local host) is tightly synchronized to NTP time, while Host B (the target) is running precisely 10 seconds behind NTP time. If Host A runs *clockdiff* against Host B at, say 5 seconds after midnight UTC on Tuesday, then the Originate timestamp from this machine would state its time as 00:00:05 UTC, while the Transmit timestamp from Host B would state 23:59:55 (and however many milliseconds it took to receive and process the request from

Host A) UTC (as Host B thinks it is still Monday). Without any adjustment, *clockdiff* would report a delta of +23:59:50 (or 86,390,000 milliseconds), which would not reflect the true difference between the host. Instead, when *clockdiff* processes this delta, it subtracts 86,400,000 milliseconds, reporting a delta of -00:00:10, which is the correct difference in this case.

While one could use the above situation to support *clockdiff*'s inclusion of this step in its algorithm, we believe it is impossible to assert its general correctness. As stated in the source code, this adjustment is appropriate "around midnight." But how close to midnight is "around midnight?" Ten seconds? Ten minutes? Ten hours? Any cutoff that we or others might propose for when to perform or not perform this delta adjustment would be completely arbitrary. As shown by previous studies of how well thousands of servers across the Internet maintain synchronization with a standard time such as UTC (Buchholz and Tjaden 2007), there is no valid basis for the assumption that all computers are synchronized to within 12 hours of standard time. Many hosts have been shown to be off from UTC by much farther than twelve hours. Because it performs this adjustment every time it encounters this time difference, *clockdiff* forces two hosts whose system times actually do differ by greater than twelve hours to appear more closely synchronized than they actually are. In these cases, *clockdiff* reports inaccurate results, as the actual timestamp values are improperly manipulated.

To make matters worse, *clockdiff* does not report to the user that the number has been changed. *Clockdiff's* "man" page does contain the statement: "*clockdiff* shows difference in time modulo 24 days." Perplexed by this statement, even after carefully analyzing the source code, we asked *clockdiff's* author to provide an explanation. In a

personal communication with the author (October 2008), he stated that this was a

mistake, and the intended statement was "*clockdiff* shows difference in time modulo 24

hours", meaning that the results from a host that is off by more than 24 hours are not to

be trusted. In fact, as shown above, *clockdiff* cannot correctly handle a situation in which

the system time on two machines differs by more than twelve hours, nor was it ever

intended to handle this situation. Unfortunately, the output from running the program

against a target that is closely synchronized with the local host and one that differs by

greater than twelve hours may look extremely similar. Because the user is never notified

when an arbitrarily manipulated result is displayed, at least one other means of measuring

the system time on a remote host is necessary to corroborate the result reported by

clockdiff.

3.2 Improving System Time Measurement in Speed and Fidelity to the Raw Data

Our solution to this problem centered on developing a program that utilizes the

same internet protocols for obtaining a timestamp from the target host, but that does so

much faster and is not limited by the need for the two hosts to be synchronized to within

12 hours. Therefore, we wrote a program called *clockvar* that also uses ICMP and IP

timestamp messages to determine the differences in system time between two computers.

Also, since the *clockdiff's* output consists only of the calculated results (and thus we

can't know when the time difference has been manipulated), we wanted to provide the

user with the raw timestamps in binary format and a human-readable representation of

the raw timestamps in addition to the time difference calculated by the program.

Clockdiff can only be invoked against one target at a time. In order to make

clockvar more useful for a large-scale study like the one described above, we built in the

option of reading a list of targets from a file. In this section, we describe how our program produces its results.

3.2.1 How *Clockvar* Obtains the Time Difference Between Two Computers

This pseudo code algorithm demonstrates the way in which *clockvar* measures the system time on remote hosts:

```
socket timeout value = value read from config file
target_start = gettimeofday()
local_time = Originate timestamp = target_start % 86,400,000

send timestamp request to target host
if (reply received)
        target_finish = gettimeofday() % 86,400,000
        target_time = Transmit timestamp
        if (target_time & 0x80000000 != 0)
                exit loop; report non-standard timestamp format
        end if

        rtt = target_start – target_finish
        delta  = target_time – local_time – (0.5 * rtt)

        output: host name, IP address, timestamp option used, actual
        local and target timestamps received, rtt, delta
else
        when socket timeout expires, declare host down
end if
```

This process is elaborated in the following 8 steps:

1. When the raw socket is created, the socket timeout value (read from the `config` file) is set using `setsockopt`; this prevents the program from waiting for a reply in an infinite loop. If the timeout expires, *clockvar* reports that this host is not responding and moves on to the next target.

2. The current system time on the local host is obtained using `gettimeofday()` and stored in the timeval struct `target_start`. This struct is converted to milliseconds since the epoch and divided modulo 24 hours. This value, representing the local host's system time in number of milliseconds since midnight UTC, is stored in the variable `local_time`.

3. *Clockvar* sends a timestamp request (or echo request with IP timestamp options set) to the target host, with the value of `local_time` as its originate timestamp.

4. When *clockvar* receives the timestamp reply, it stores the value as `target_time` and gets the local host's system time once more, storing it as `target_finish`.

5. The round-trip-time to the target is calculated by subtracting the timeval struct `target_start` from `target_finish`.

6. The estimated delta between the two system clocks is calculated by subtracting `local_time` from `target_time` and then subtracting ½ of the round-trip time.

7. The round-trip-time also represents the maximum error range for the time measurement. This is the worst-case scenario for the amount by which *clockvar*'s estimated delta could diverge from reality. That is, we are assuming that if it takes zero milliseconds for the local host to transmit the timestamp request to the target host, then it would take the full number of milliseconds in `rtt` for the program to receive the timestamp reply from the target host, and thus the accuracy of the reported delta could be off by up to that number of milliseconds. Certainly, in reality, it is not possible for the transmission of the timestamp request to be instantaneous; thus, we can have a high degree of confidence that the reported

`rtt` does in fact represent an upward bounds on the actual error caused by network delay.

8. The default behavior is for *clockvar* to display the following data:

   1. number of the target host (out of total number of targets)

   2. the target host's name and IP address,

   3. the timestamp option used

   4. the actual timestamp from the local and target hosts, formatted as hh:mm:ss.000 (hours, minutes, seconds, and milliseconds since midnight UTC)

   5. the round-trip time for obtaining the timestamp from the target host

   6. the estimated delta, calculated by the formula:

   $$\texttt{delta} = \texttt{target\_time} - 0.5(\texttt{rtt}) - \texttt{local\_time}$$

We also added an option so that *clockvar* could be invoked to run in a "*clockdiff* mode"; that is, when it encountered a system time difference of greater than 12 hours, it would perform the same adjustment that *clockdiff* does. We foresaw that it would be useful to collect data from target hosts in both modes for comparison with the *clockdiff* results.

3.3 Confirming the results: how *Web-time* Works

Establishing the degree to which internet hosts are synchronized to standard time was not one of the goals of our study. However, in order to provide a third method of time measurement for the inevitable situations when *clockdiff* and *clockvar* would disagree, we wrote our own version of *web-time* to run against each target host along with the other two programs.

According to the standard set by RFC 2616, servers must use the following format when including date information in HTTP/1.1 header fields transmitted to clients:

Ddd, dd Mmm yyyy hh:mm:ss GMT

where Ddd represents the three-letter abbreviation for the weekday, dd represents the two-digit date, Mmm represents the three-letter abbreviation for the month, yyyy is a four-digit year, and hh:mm:ss represents the 6-digit hour, minute, and second. Although servers are not required to transmit a timestamp, if they do, the time must be according to GMT, and hence the "GMT" must be included with the timestamp (Fielding, 1999).

We wrote a simple program that, given an IP address, transmits the following request to the server: `"GET /index.html HTTP/1.1 \r\n\r\n"`. Since our program does not first establish a TCP connection with the remote host, the server typically sends back a "Bad request" message, most often including a date/time stamp. *Web-time* parses the response, looking for the line which includes the word "Date", and then reads the rest of the line. An example of *web-time's* output is: "`Thu, 01 Jan 2009 06:40:59 GMT`".

IV. Experiments and Results

We conducted a large-scale experiment in order to test the speed and accuracy of *clockvar*. For this experiment, we had three primary goals:

1. Drastically reduce the amount of time needed (versus using *clockdiff*) to determine the system times on a large number of hosts.

2. Show that the difference in system times can be measured with reasonable accuracy using just one timestamp request (versus 50 messages, as *clockdiff* uses). It was our desire to see the difference between the results reported by *clockdiff* and *clockvar* to be under 50 milliseconds for the vast majority of hosts surveyed.

3. Demonstrate that, when the system times on two hosts differ by more than 12 hours, *clockvar* generates output that is more consistent with the raw timestamp data that it receives than *clockdiff*'s output.

In the following sections, we outline how we conducted our experiments and analyzed the data, and then discuss the significance of the results we obtained.

4.1 Experiment Setup

So that we could test *clockvar* and *clockdiff* against a substantial number of targets, we began with the same list of 8,410 hosts used in Buchholz's and Tjaden's experiment. In order to eliminate those hosts that simply would not respond to ICMP and IP timestamp requests, we ran both *clockvar* and *clockdiff* once against all 8,410 hosts (August 2008). Our analysis of the outcome was not surprising: many of the hosts did not respond to our timestamp requests (far fewer, in fact, than in the earlier Buchholz /

Tjaden experiment); however, the ones that replied to *clockdiff* were the same ones that replied to *clockvar*. A total of 2,389 hosts responded to at least one of the options at that time. This list of 2,389 hosts was used in our daily measurements.

We installed *clockvar* and *clockdiff* on a server running Red Hat Enterprise Linux (version 2.6.9-67.EL) which was synchronized with standard time via NTP. We wrote a simple shell script that reads our target list, and for each target invokes first *clockvar*, then *clockvar* running in *clockdiff* mode, then *web-time*, and finally *clockdiff*. All of the output from each program is directed to a single output file for later study.

We composed a program that parses the script output (for a single day) and places the results of each of the programs into a single, tab-separated line containing (along with some raw data) the host name, the time differences generated by *clockvar*, *clockvar* (in *clockdiff* mode), and *clockdiff*, and the date/time string produced by *web-time*. The output of this program can be opened with any spreadsheet program for further analysis. Although we rarely saw a measurement difference between *clockvar* and *clockdiff* of greater than one second, we decided to classify any difference of at least 10 seconds as an outlier. For the overwhelming majority of hosts, the difference in the results reported by the two programs was 10 milliseconds or less, but there were a few outliers each day. We hypothesize about the causes of the outliers in a subsequent section.

One of the functions of this program was to sort the measurement differences into a number of "bins" in order to generate a histogram and cumulative histogram of the results. For the charts and tables in Chapter IV, we define the rule for placing a value into a bin as follows:

bin label	values in the bin
0	difference between *clockvar* and *clockdiff*'s measurement is 0
10	0 < measurement difference ≤ 10 milliseconds
20	10 < measurement difference ≤ 20 milliseconds
…	
more	300 < measurement difference ≤ 10,000 milliseconds (10 seconds)
outlier	10,000 milliseconds < measurement difference

Initial experiments revealed that, for most of the target hosts, the difference between the measurements was exceedingly small, and that 98% of the differences were less than 300 milliseconds. Thus, for the sake of limiting the number of bins, we placed any value between 301 and 10,000 milliseconds into the "more" bin, and any value greater than 10,000 into the "outlier" category.

4.2 Highlights of the Results

We ran our experiment from November 3, 2008 to February 21, 2009. Due to system crashes, we are missing some data from a few of these days. However, we collected complete data for a total of 105 days, including 172,259 measurements of the system time on remote hosts using *clockvar*, *clockdiff*, and web-time. On average, 1,656 hosts responded to timestamp requests each day. The lowest number of hosts responding was 899 on January 12 (the server crashed after just over one half of the hosts had been measured), and the highest number of hosts responding was 1,794 on November 13. Although we surveyed 2,389 total hosts each day (and these were the hosts from which we initially received responses), the numbers responding each day of the experiment

never approached this number for two reasons. First, the initial measurement was taken using a server with a direct connection to the Internet, and it was capable of processing both ICMP and IP timestamp requests and replies. The remainder of the experiment was performed using a server behind a NAT, and we discovered that neither *clockdiff* nor *clockvar* is capable of receiving timestamp replies to requests transmitted from behind a NAT. Thus, this excludes the hosts that had initially responded to either of the IP timestamp request options, but not to ICMP timestamp requests. Secondly, the 2,389 include all hosts for which *clockvar* did not experience a server timeout. That is, these hosts transmitted a response, but it may not have been a valid timestamp. For example, the non-standard response bit may have been set, or the timestamp may have turned out to be a number greater than 24 hours. These responses were discarded as invalid by both *clockdiff* and *clockvar*.

Ideally, we would see the delta reported by *clockvar* exactly match the delta reported by *clockdiff* for all hosts, except for those where the delta is greater than 12 hours (as we know the programs deal with the deltas differently in this case). In reality, though, we expected to see some difference in these numbers due to the fact that *clockdiff* executes a more complicated algorithm for estimating the network delay.

When *clockvar* and *clockdiff* measure the system times on two hosts, they report the delta between them in milliseconds. As stated above, we had hoped to find that the difference between the deltas would be less than 50 milliseconds for the majority of targets surveyed. In fact, we found that the difference between the measurements of *clockvar* and *clockdiff* turned out to be only 10 milliseconds or less for 95.17% of the

hosts measured over the course of the experiment. We now discuss some of the trends we discovered.

4.2.1 The Precision of *Clockvar* and *Clockdiff*

Our experiment revealed that it is possible to measure the system time difference between two computers using the same internet protocols as *clockdiff* with a very high degree of accuracy, but with far greater speed. We show the advantages gained in speed in section 4.4. Here, we would like to focus on how consistently close *clockvar*'s estimated deltas (using one measurement) are to *clockdiff*'s, which utilizes a series of 50 measurements before displaying a result. **Table 1** shows a complete summary of the averages of each daily round of measurements during the entire experiment. The first column is a list of the bins (as described above), with one additional row, which we are calling "*clockdiff* outlier" and define below). The second is an average number of hosts falling into each bin per day, rounded to a whole number. The third is a sum of all the hosts in each bin over the entire 105-day experiment. The last two columns are the percentage and cumulative percentage of the total number of hosts responding in the experiment.

Time diff. msecs	Avg. daily hosts	Total hosts	% of total	Cumulative %
0	548	57,037	33.11%	33.11%
10	1028	106,908	62.06%	95.17%
20	28	2,899	1.68%	96.86%
30	10	1,001	0.58%	97.44%
40	5	531	0.31%	97.75%
50	3	305	0.18%	97.92%
60	1	118	0.07%	97.99%
70	1	109	0.06%	98.05%
80	1	70	0.04%	98.10%
90	1	62	0.04%	98.13%
100	0	37	0.02%	98.15%
110	0	38	0.02%	98.17%
120	0	29	0.02%	98.19%
130	0	35	0.02%	98.21%
140	0	18	0.01%	98.22%
150	0	17	0.01%	98.23%
160	0	24	0.01%	98.25%
170	0	7	0.00%	98.25%
180	0	13	0.01%	98.26%
190	0	6	0.00%	98.26%
200	0	5	0.00%	98.26%
210	0	9	0.01%	98.27%
220	0	4	0.00%	98.27%
230	0	8	0.00%	98.28%
240	0	3	0.00%	98.28%
250	0	11	0.01%	98.28%
260	0	4	0.00%	98.29%
270	0	10	0.01%	98.29%
280	0	2	0.00%	98.29%
290	0	4	0.00%	98.30%
300	0	3	0.00%	98.30%
301 – 10,000	20	2,069	1.20%	99.50%
"*clockdiff*" outlier	3	356	0.21%	99.71%
outlier (> 10,000)	5	507	0.29%	100.00%
Total	1,656	172,259	100.00%	100.00%

Table 1: Summary of the Results of the *clockdiff* / *clockvar* Comparison

4.2.2 Consistency of the Results

From day to day, the distribution of the measurement differences across the "bins" did not vary much. **Figure 6** shows a breakdown of this data into just 5 categories. The largest of these (95.17%) contains those instances where the measurements taken by *clockvar* and *clockdiff* differ by 10 or fewer milliseconds. The second largest group (3.12%) consists of differences of between 10 and 300 milliseconds. The next group (1.20%) contains those measurements that differed by between 301 and 10,000 milliseconds. Outliers made up 0.29% of the measurements, and the smallest group (0.21%) consists of a special category of outlier, which we are calling "*clockdiff* outlier". These are all hosts whose system times differed from NTP time by more than 12 hours. Thus, the deltas reported by *clockdiff* and *clockvar* differed greatly, as *clockdiff* added or subtracted 86,400 seconds (24 hours) before displaying the delta. For each of these hosts, however, the differences between the *clockdiff* and *clockvar* running in *clockdiff* mode (i.e., making the same delta adjustment *clockdiff* makes) were very small. Although this category contains the fewest hosts, the calculations we made (as well as our inspection of the raw packets transmitted) prove that, for these 356 measurements, *clockdiff* did indeed report a delta that did not correlate to the raw timestamps it received from the target hosts.
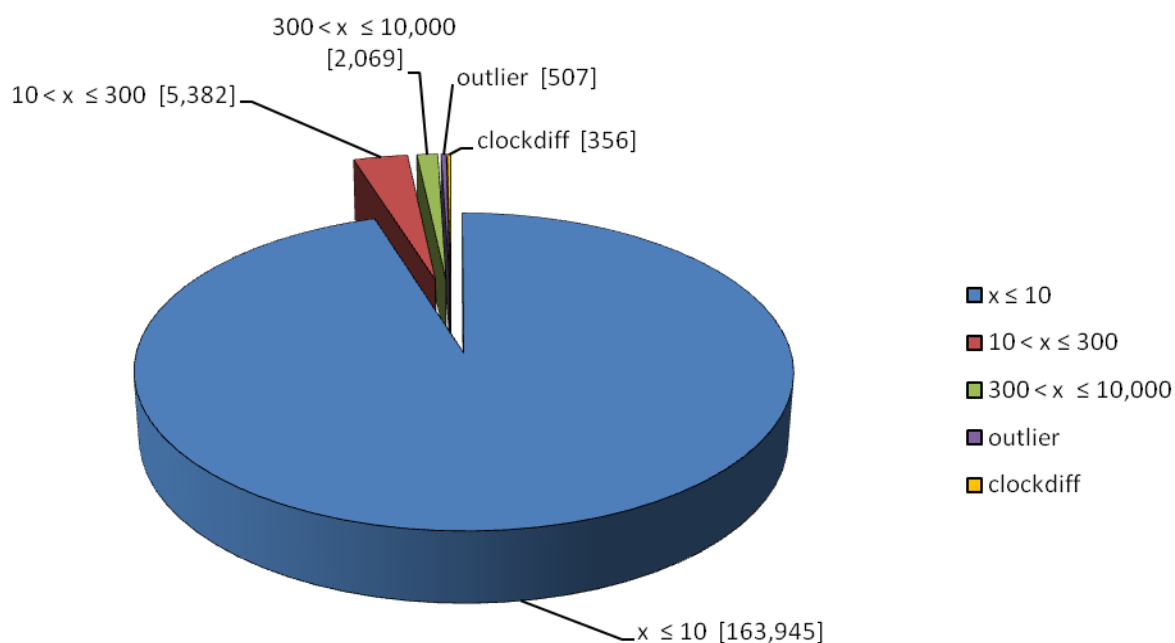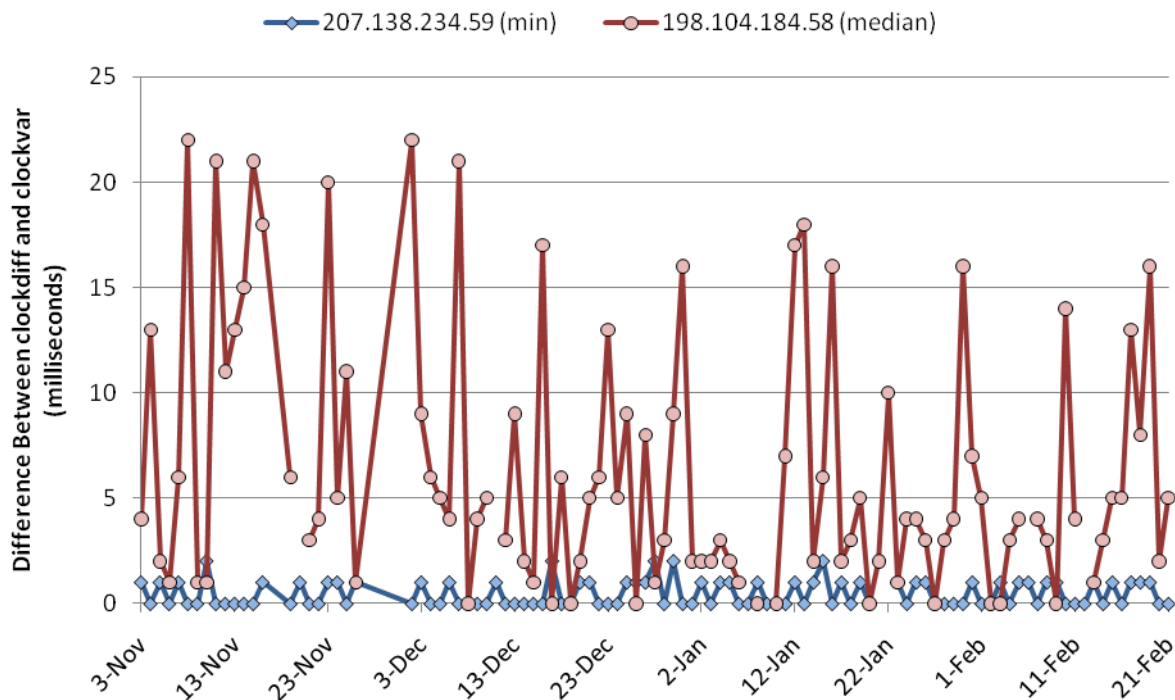
**Figure 6: Average Differences Between *Clockdiff* and *Clockvar* Measurements
Entire Experiment: 172,259 measurements**

4.2.3 Consistency in the Measurement of Individual Hosts

In addition to seeing a consistent distribution across the bins, we hoped to see a

great deal of consistency in the difference between *clockvar*'s and *clockdiff*'s

measurement of each individual host across the entire experiment. For the majority of

hosts, these differences were consistent when we measured them over the 105 days of the

experiment. The server with the lowest average difference (0.47 milliseconds) between

clockvar's and *clockdiff*'s measurements was 207.138.234.59. On many days, the

difference between the measurement from each program was exactly 0, with the highest

difference being only 2 milliseconds. The median value for average differences was 6.41

milliseconds, and this came from the server at 198.104.184.58. Differences for this target

ranged from 0 to 22 milliseconds, and we did not receive responses to our timestamp requests on 6 days of the experiment. A graph of the differences between the measurements of *clockvar* and *clockdiff* for these two servers can be seen in **Figure 7**.

Figure 7: Minimum and Median Differences Between *Clockdiff* and *Clockvar*

Figure 8 represents the other end of the spectrum. Of the hosts that did not qualify by our criteria as "outliers" (a difference of over 10 seconds), the server at 168.83.72.5 had the largest average difference between the measurements made by *clockvar* and *clockdiff*, 1194.90 milliseconds, or just over one second. For this host, the smallest difference was a mere 2 milliseconds, but the largest was 6.058 seconds. We received replies from this server to all but 5 of our timestamp requests. Although we are not able to confirm this hypothesis, it is possible that multiple physical machines (with unsynchronized clocks) answer timestamp requests to this IP address. That would

provide a reasonable explanation for such a large difference among measurements taken each day within milliseconds of each other and using the same Internet protocols. **Figure 8** shows the differences in the measurements for this host on each day of the experiment.
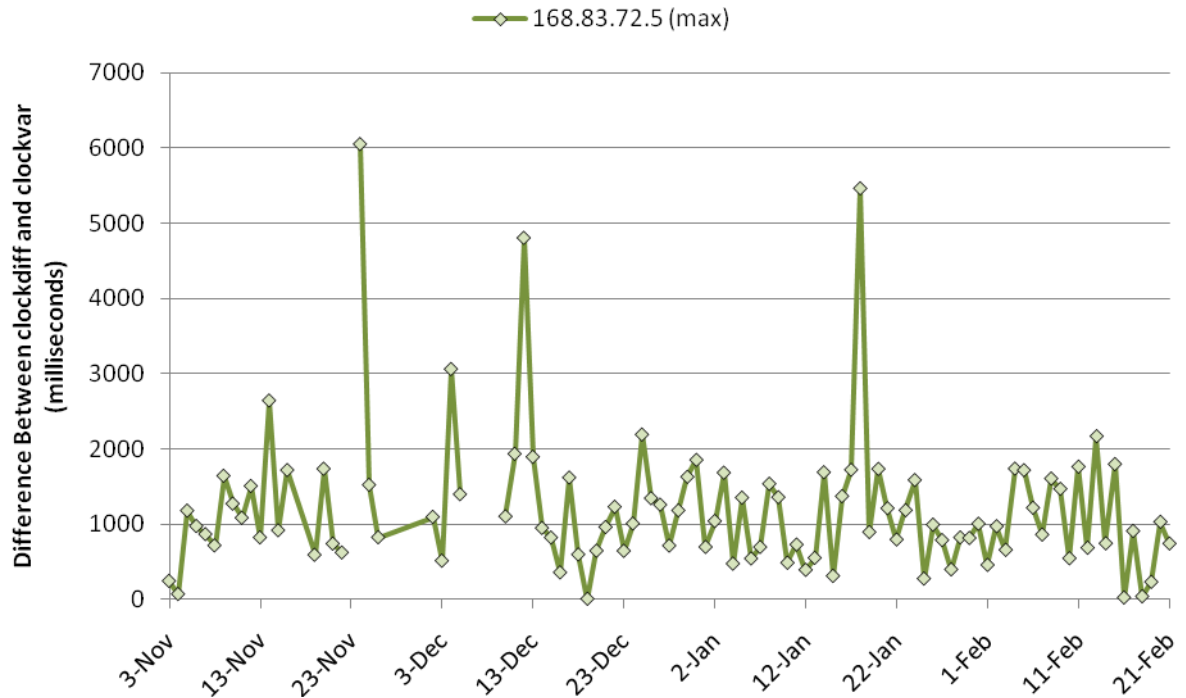

Figure 8: Maximum Differences Between *Clockdiff* and *Clockvar* (non-outlier)

4.3 Outliers

Each day of the experiment, there were measurements that qualified as outliers; that is, the differences between *clockvar*'s measurement and *clockdiff*'s was greater than 10 seconds. Our initial evaluation of the data yielded the conclusion that outliers made up a total of 863 out of 172,259 measurements in the experiment (0.501%). However, further analysis revealed that in 356 of these cases, the measurement was being classified as an outlier due to the fact that *clockdiff* made a 24-hour adjustment in the delta, while *clockvar* reported the delta based solely on the raw data it received. We now classify

these 356 (0.207% of the total) measurements as "*clockdiff* outliers" and the remaining

507 (0.294%) as regular outliers. **Figures 9** and **10** show the total number of both kinds

of outliers that occurred each day of the experiment.
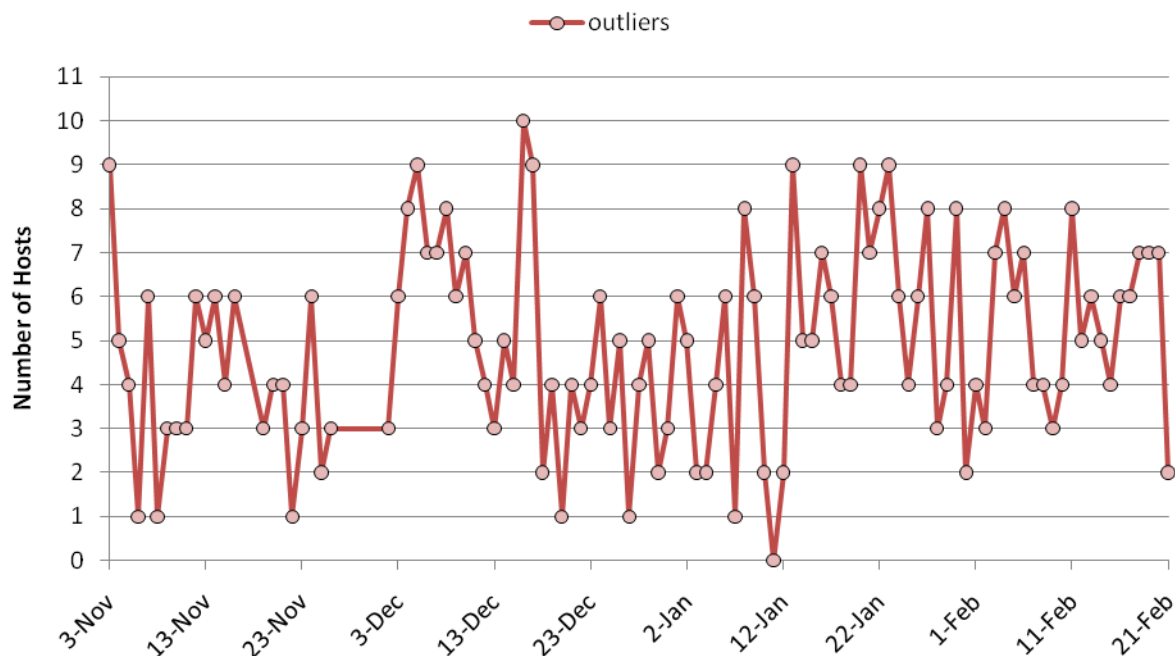
Figure 9: "Regular" Outliers Per Day

4.3.1 Extreme Outliers: Hosts Differing from NTP Time by More Than 12 Hours

Examining *clockdiff*'s code convinced us that this program does not report a delta

that reflects the timestamps it receives when the system time on the host on which it is

running differs from that of the target host by over 12 hours. When we began to

investigate why *clockvar* and *clockdiff* reported such widely varying measurements for

the system time on these hosts, we discovered that the hosts whose system clocks

diverged from NTP time by more than 12 hours were always outliers. In each case, the

delta reported by *clockvar* was much larger than the one reported by *clockdiff*. We

concluded that *clockvar* was reporting the actual difference, while *clockdiff* was manipulating the timestamp response due to its assumption that the clocks must be synchronized to within 12 hours.
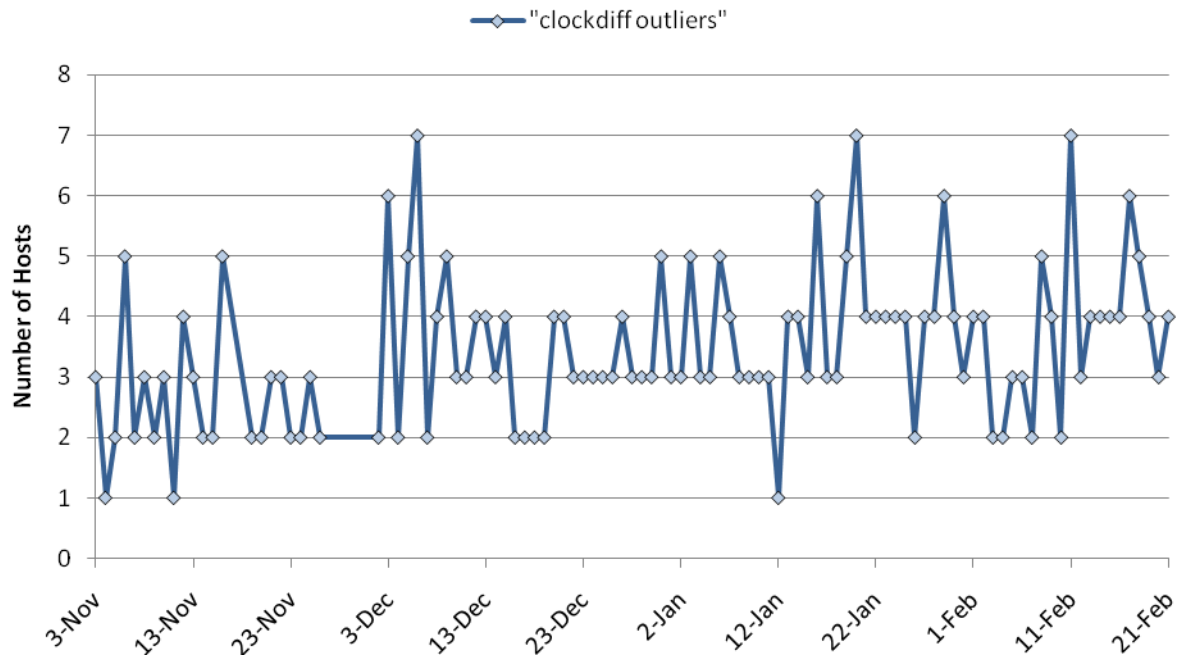
Figure 10: "*Clockdiff*" Outliers Per Day

In order to prove that this was the case, we performed additional experiments on these hosts using a laptop running Windows Vista. This machine uses a VMware workstation to run a Fedora Core 4 installation of Linux 2.6.11-1.1369. While running our shell script described in section 4.1 on the virtual machine, we captured the raw network packets using Wireshark, which was running in the Windows environment. As an example, we show the results of a series of measurements against host 69.57.128.4 on March 3, 2009.

Clockvar reported a local time of 01:55:54.423 UTC (the local time on our machine was 8:55 pm Eastern Standard Time, which is 5 hours behind GMT). *Clockvar* reported the target timestamp of 20:57:57.098 and a round-trip time to the target as 122 milliseconds. *Clockvar* calculates the delta by subtracting the two raw timestamp numbers (75,477,098 – 6,954,423) and then subtracting ½ of the RTT (61). This yields a difference of 68,522,614 milliseconds, or 19 hours, 2 minutes, 2 seconds, and 614 milliseconds (+19:02:02.614). As part of our experiment, *clockvar* ran a second time in "*clockdiff* mode", that is, configured to adjust the delta by 24 hours as *clockdiff* does. Just milliseconds later, the program reported the target timestamp of 20:57:57.208, but a delta of -17,877,394 milliseconds, or (-4:57:57.394). When we examined the raw data in the packet through Wireshark, we confirmed that the program received both a Receive and Transmit timestamp of (network byte ordered) 0x047fb0d8, and this corresponds to 20:57:57.208, which the program reported.

Around a half of a second later, *clockdiff* received its first of 50 responses for this host. Again, using Wireshark to view this packet, we observed that the raw data *clockdiff* received as a timestamp was 0x047fb2b0, or 20:57:57.680 after midnight UTC. The time on the local host was 1:55:55.034, so *clockdiff* should have reported a delta of (+19:02:02.582). If it had, the difference between *clockvar*'s and *clockdiff*'s deltas would have been only 32 milliseconds. However, *clockdiff* reported the delta as -17,877,418, or (-4:57:57.418). Thus, the difference between *clockvar*'s and *clockdiff*'s deltas is 24 hours and 32 milliseconds; however, the difference between *clockdiff*'s and *clockvar*'s (in *clockdiff* mode) deltas is only 24 milliseconds. As we have such a close agreement between *clockdiff* and *clockvar* in *clockdiff* mode, we can conclude with

confidence that there is such a large disagreement between *clockdiff* and *clockvar* (in normal mode) because *clockdiff* is adjusting the delta prior to reporting its results.

Interestingly, the timestamp string we received from this host via web-time was "Tue, 3 Mar 2009 20:57:57 GMT" which corroborates the delta and timestamp reported by *clockvar* exactly. Furthermore, this gives even less credence to the result reported by *clockdiff*.

We encountered an average of 3.42 measurements exhibiting this behavior (a huge difference between the *clockdiff* and *clockvar* deltas, but a very small difference between *clockdiff* and *clockvar* in *clockdiff* mode) in each daily run of the experiment.
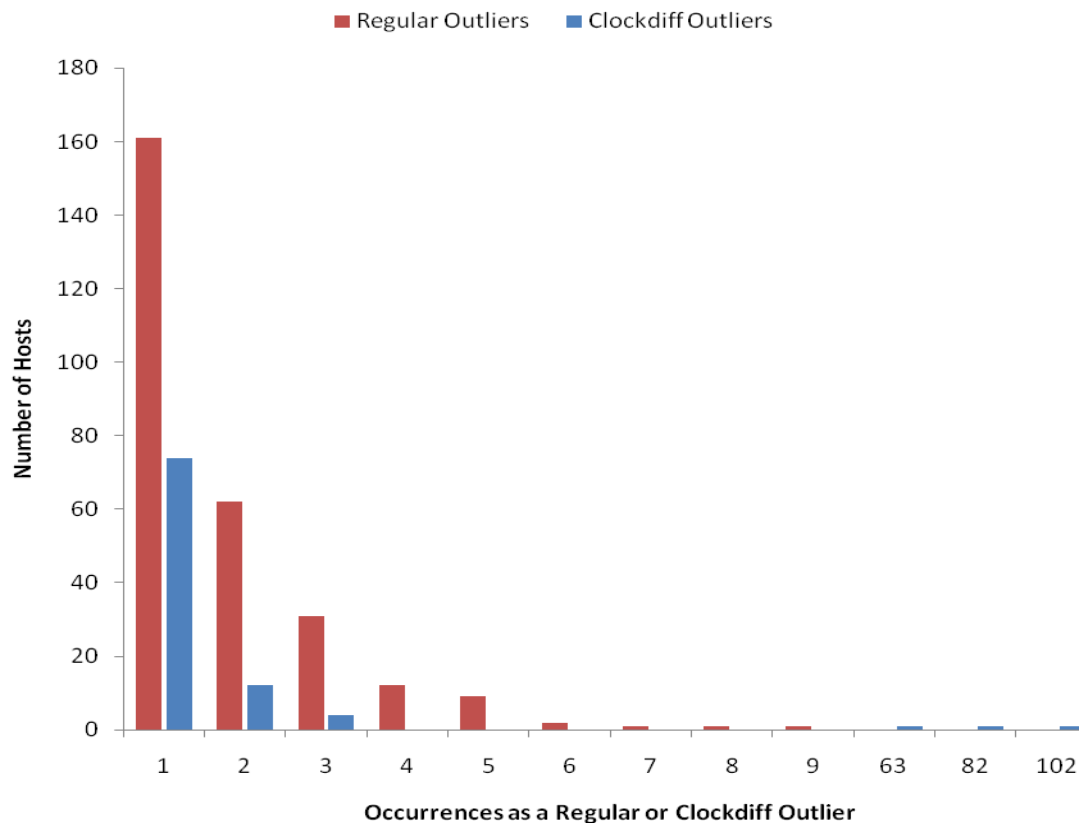
Figure 11: Number of Times an Individual Host was a Regular or "*Clockdiff*" Outlier

The 356 total occurrences of "*clockdiff* outliers" corresponded to only 94 distinct hosts. The servers at 72.9.249.194, 206.176.210.45, and 69.10.136.151 fell into this category 63, 81, and 102 times respectively. The other 91 hosts were *clockdiff* outliers only 1, 2, or 3 times over the course of the experiment. **Figure 11** shows this distribution.

4.3.2 Other Outliers

Figure 11 also shows that 280 distinct hosts fell into the outlier category (a difference of over 10 seconds between the deltas reported by *clockdiff* and *clockvar*) at least once. The maximum number of times that a particular host was an outlier was 7 times. The 507 instances of outliers accounted for 0.29% of the total 172,259 measurements. While this already represents a small percentage of the total, a further 497 of these occurrences can be explained – and eliminated with more careful coding.

When we produced the first version of *clockvar*, we knew that we would be comparing its results against *clockdiff*'s against a large list of targets in a single run. When we wrote the shell script that invoked both programs against each host in the target list, we were aware of the possibility that one program could begin to measure the time on a new target before the other program finished measuring the time on the previous target. For instance, we saw in the packet capture files that the shell script occasionally caused *clockvar* to launch its timestamp request to, say, host 4 before *clockdiff* had received its last response from host 3. In order to deal with this possibility, we initially took *clockdiff*'s lead.

The Linux kernel passes ICMP packets (such as the ones used by *clockdiff* and *clockvar*) to all open raw sockets. *Clockdiff* uses part of its own process identifier (ID) to determine whether it is the intended destination of each packet that the kernel sends it.

When a Linux process is created, the system assigns the process an integer identifier, called the Process ID. In order to insure that a particular *clockdiff* process "recognizes" the timestamp requests that it sends out, *clockdiff* places part of its process ID into the ICMP Identifier (ICMP ID) field of all outgoing packets, and then checks for this number in the ICMP ID field of all incoming packets, rejecting any that do not have the correct identifier. The ICMP ID field is only 16 bits long, but the process ID is a 32-bit value. Thus, *clockdiff* determines its ICMP ID by performing a logical AND between its process ID and 0xffff.

We employed the same strategy in *clockvar*; however, due to a slight coding error in the version that ran throughout the experiment, only the last 8 bits of the ICMP ID were tested. Thus, in the rare cases that each of the following conditions occurred, *clockvar* processed one of the packets intended for *clockdiff*, and reported incorrect results for the target host:

1. The last 8 bits of both *clockdiff*'s and *clockvar*'s process ID were identical.

2. The shell script caused *clockvar* to initiate measurement of one target host before *clockdiff* had finished measuring the previous host. That is, *clockvar* has sent a timestamp request to host 2 while *clockdiff* is awaiting a reply from host 1.

3. The timestamp reply from host 1 (intended for *clockdiff*) arrives prior to the reply from host 2 (intended for *clockvar*).

In this case, both *clockdiff* and *clockvar* would process the timestamp reply received from host 1 (*clockvar* then closes the open socket after receiving what it believes to be a response to its request, and thus it never receives the reply from host 2). However,

clockvar would report this as the timestamp reply received from host 2, and thus its calculated delta would not match that of *clockdiff*'s.

The following sequence of packets sent and received to two target hosts illustrates this condition. In Frame 5106 of the packet capture from an entire run of our comparison shell script, *clockvar* sends a timestamp request to 213.238.33.194; its ICMP ID is 0x8946. In Frame 5107, we captured the 50th response from 213.218.116.170 (the previous target), intended for *clockdiff*, but processed by both programs. This is due to the fact that *clockvar* had an open raw socket at this point awaiting a response from 213.238.33.194, and *clockdiff*'s ICMP ID is 0x8546 (the last 8 bits match *clockvar*'s) . Here, the target's transmit timestamp is 0x024c840b, or 10:42:48.971. *Clockvar* computed the round trip time as 3 milliseconds, and calculates the delta using the raw timestamps as 38568971 - 26130389 (10:42:48.971 - 07:15:30.389) – (½ * 3) , or 12438581 milliseconds (3:27:18.581).

Frame 5108 is the second timestamp request from *clockvar* to 213.238.33.194. A second timestamp request is sent when *clockvar* runs in "*clockdiff* mode"; the delta calculated from this reply would undergo the same adjustment *clockdiff* uses if its delta were greater than 12 hours. Frame 5109 contains the reply from 213.38.33.194 to *clockvar*, which the program did not process, as it believed Frame 5107 was this reply. Had it processed this response, it would have calculated the delta between the two hosts as 42506772 - 26130389 (11:48:26.772 - 07:15:30.389) – (½ * 3), or 16376382 milliseconds (4:32:56.382 UTC). If so, then the difference between *clockdiff* and *clockvar* deltas for this host would be only 174 milliseconds, as opposed to the 3937974

milliseconds (1:5:37.974) reported by the program we wrote to process the shell script output.

Frame 5110 contains the second timestamp reply from 213.238.33.194 to *clockvar*. This time, *clockvar* is not confused by a packet intended for *clockdiff*, and it correctly reports the target timestamp as 11:48:26.772. This time, the shell script processing program reports a reasonably small difference between the deltas calculated by *clockvar* and *clockdiff*.

4.4 Performance

Since one of the primary goals for our work is to determine the system time on remote hosts drastically faster than using *clockdiff*, we carefully evaluated how much time it took for each program to perform its measurements. In order to focus on the differences due to the measurement algorithms (primarily, *clockvar*'s one packet method versus *clockdiff*'s use of fifty packets), our initial comparisons were made running *clockvar* as a single-threaded application, since *clockdiff* does not have multi-threading capability. We analyze the performance gains from multi-threading following the single-threaded comparison.

We used a shell script to measure *clockdiff*'s performance against a large number of targets (since the program can only measure one target per invocation). We discovered that it takes *clockdiff* an average of 12,039 milliseconds (12.039 seconds) to return a result for a target that is responding to the timestamp requests. When the target host is not responding, it takes this program an average of 10,861milliseconds (10.861 seconds) to announce that the host is down.

Clockvar, on the other hand, has a configurable server timeout value (expressed in seconds and microseconds). This value is set in the `config` file and can be changed at any time without recompiling the program. We initially found that *clockvar* receives a response from all the hosts in this study (that is, if the host is answering timestamp requests) in less than two seconds. We concluded that two seconds would be an appropriate timeout value for all subsequent experiments, and thus it takes the program an average of 2 seconds per host to report that a target is not answering. Typically, *clockvar* displays its results in an average of 104 milliseconds (0.104 seconds) when the target host is answering the timestamp requests. **Table 2** shows the results of our performance experiments.

	Total Time (hh:mm:ss)		Average Time per host (seconds)	
Experiment	*clockvar*	*clockdiff*	*clockvar*	*clockdiff*
Test 1: 8410 hosts	3:20:21	26:08:04	1.429	11.187
Test 2: 2382 responding	0:04:02	7:47:07	0.104	12.039
Test 3: 6082 not responding	3:16:18	18:20:57	1.937	10.861

**Table 2: Performance Times of *Clockdiff* and Single-Threaded *Clockvar***

For Test 1, we used all 8,410 hosts from the initial experiment. 2,328 of the target hosts responded to *clockvar* and *clockdiff*, while 6,082 did not answer ICMP timestamp requests. For Test 2, we used the 2,328 hosts that responded during Test 1. We used the remaining 6,082 targets for Test 3. **Figure 12** shows that the amount of time *clockdiff*

consumes to process targets grows in a linear manner along with the number of hosts.

Figure 13 shows the average time per host utilized by each program. *Clockvar*, on the

other hand, uses much less time to measure targets that are answering timestamp

requests. The largest part of the total time in a *clockvar* run against multiple targets is

spent waiting for the timeout to expire when a host is not responding. Clearly,

completing the test against all 8,410 targets took only 4 minutes longer than the 3 hours

and 16 minutes for the test against the 6,082 hosts not answering timestamp requests.

The 2,328 servers that responded to our timestamp requests were measured in those 4

short minutes.
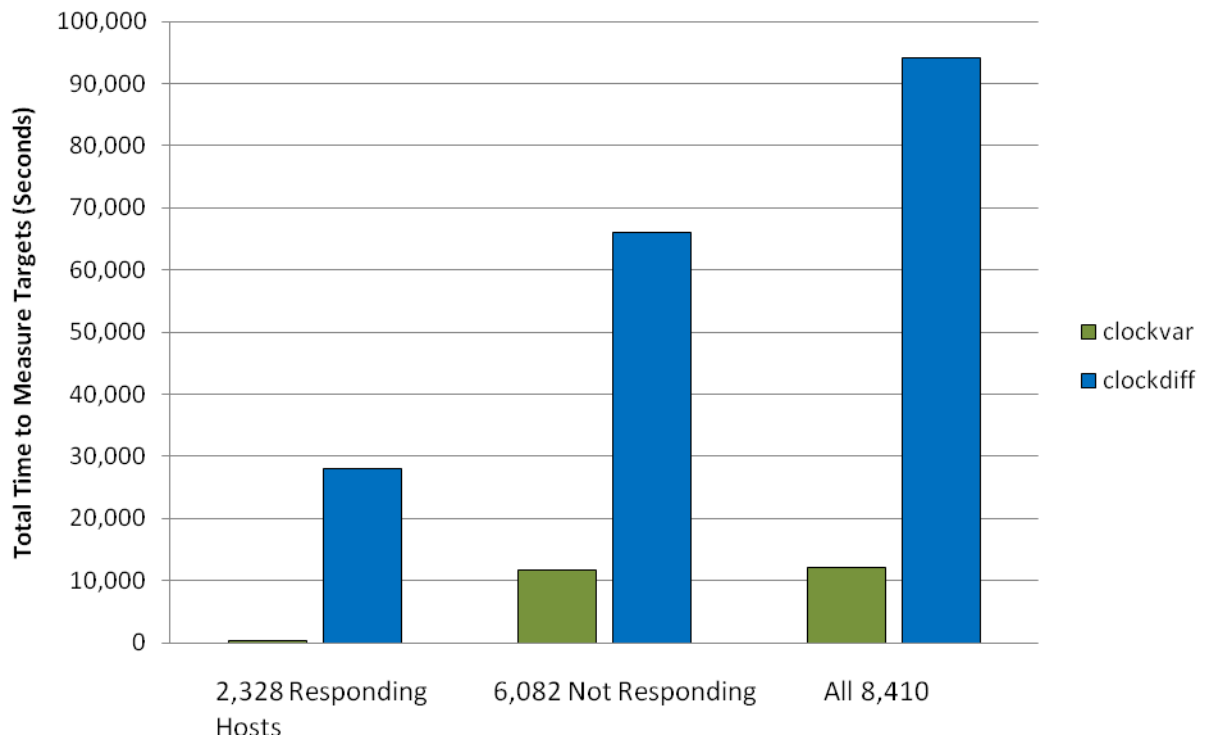
Figure 12: Total Time Consumed by *Clockvar* and *Clockdiff*
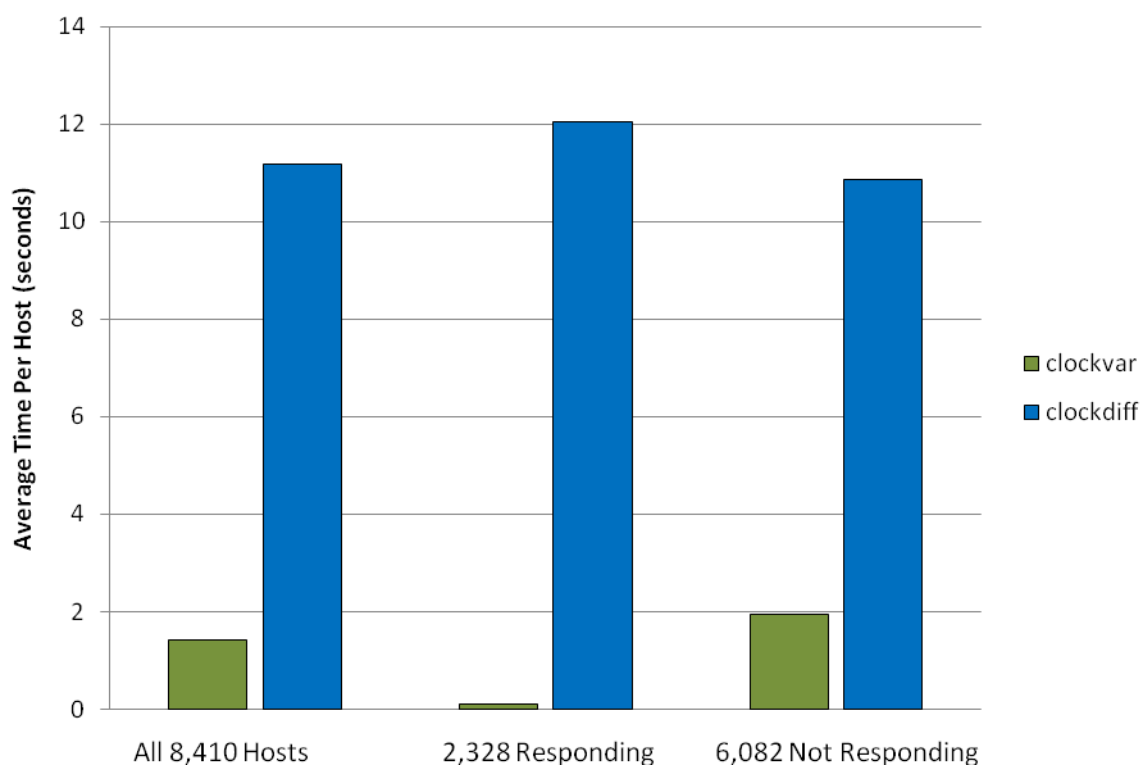
Figure 13: *Clockvar* **and** *Clockdiff* **– Average Processing Times Per Target**

As **Table 2** and **Figure 12** show, the overwhelming majority of *clockvar*'s running time is spent idly, waiting for a timeout to expire when a target host is not answering timestamp requests. Clearly, if we increase the number of targets that can be measured at the same time, then, when the targets are not responding, the program can wait for multiple timeouts at the same time, resulting in greater efficiency.

We accomplished this through multi-threading. When *clockvar* is invoked against a list of targets (contained in a text file), the user can pass the desired number of threads to run via a command line argument. *Clockvar* begins by attempting to create the number of threads requested. If more threads are requested than the machine on which it is running is capable of creating, then it proceeds with the maximum number of threads possible. Once a thread is created, it enters the following loop:

1. If unprocessed targets exist in the input file, read the next target from the file.

2. Attempt to measure the system time on the target host via the method described in Section 3.2.1. If the target does not respond, proceed to the next target after the timeout expires.

3. Write the results to the output file and/or display them on screen, depending on the command line arguments passed to the program.

Once all the targets in the input file have been processed, the threads are joined and summary statistics are displayed and/or written to file.

We performed an additional large-scale experiment to measure the benefits of multithreading. We wrote a shell script that, using the original list of 8410 servers as a target list, invokes *clockvar* with between 25 and 300 threads, in increments of 25. This

| Threads Running	Average Time (Min:Secs)	Average Time (Seconds)	Average Time (Milliseconds)	Time (Msecs) per Host
1	3:20:21	12028	12028000	1429
25	10:44	644	644851	77
50	5:50	350	350018	42
75	4:00	240	240299	29
100	2:59	179	179744	21
125	2:19	139	139683	17
150	1:56	116	116300	14
175	1:44	104	104113	12
200	1:32	92	92592	11
225	1:22	83	83478	10
250	1:16	76	76383	9
275	1:12	72	72717	9
300	1:08	68	68185	8

Table 3: Performance Times of *Clockvar* Running Various Numbers of Threads

script ran for one month, and then we averaged the results, which are displayed in **Table 3**. The columns show: 1) the number of threads that ran concurrently, 2) the average time it took for all 8410 hosts to be measured (in minutes and seconds), 3) and 4) the average time of the run in seconds and milliseconds, respectively, and 5) the average number of milliseconds required to measure the time on one target host. The first row in the table represents results from our previous experiment using the single-threaded version.
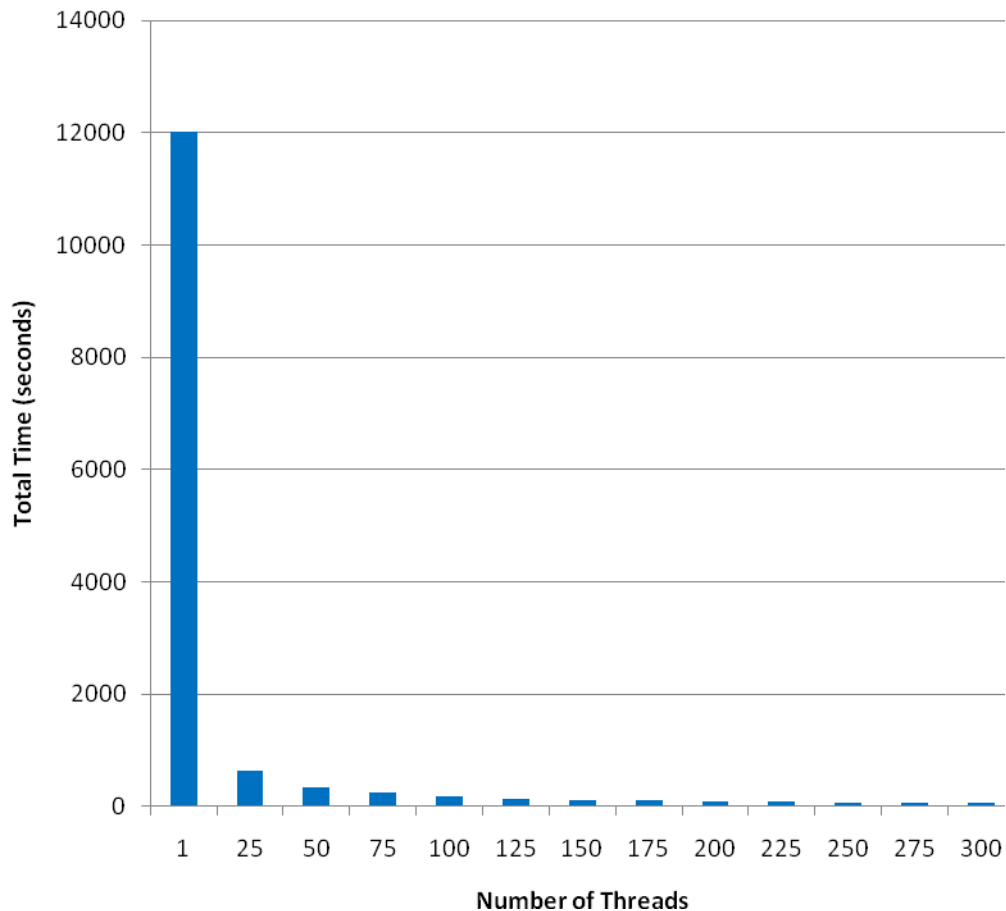
**Figure 14: Total Measurement Times of 8410 Target Hosts**

Observe **Figure 14**, the total time consumed measuring the system time on the 8410 hosts, and **Figure 15,** the total time expended waiting for timeouts to expire for hosts that did not respond. From these figures, it can clearly be seen that increasing the number of threads yields an exponential benefit in performance. *Clockvar* took over three hours to process the 8410 targets running as a single-threaded application (which is, of course, still a significant improvement over *clockdiff*, which took over 26 hours to accomplish this). Using 25 threads dramatically reduces this time to under eleven minutes, and using 300 threads brings down the total time for measuring this large number of hosts to just over one minute.
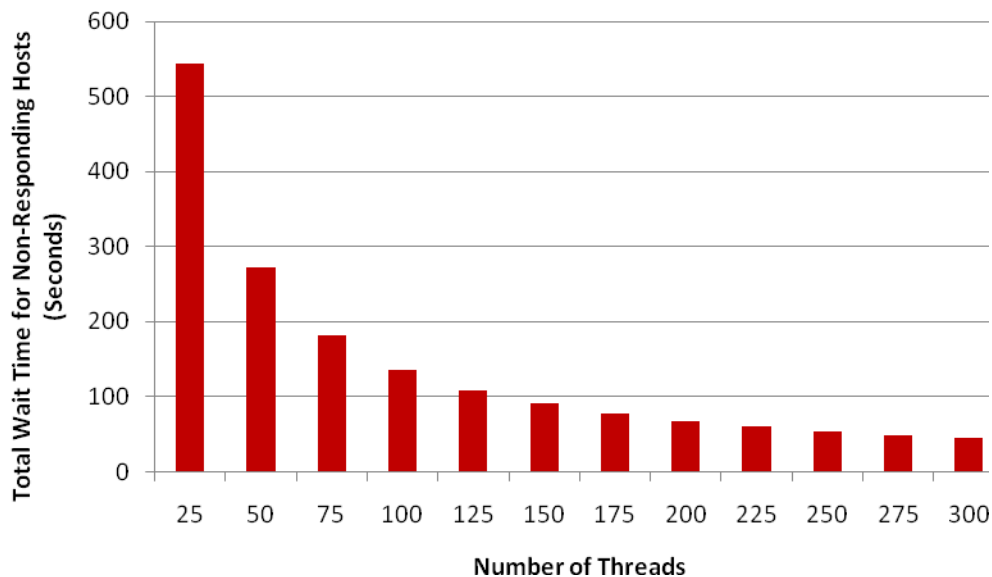
Figure 15: Total Time Consumed Waiting for Non-Responding Host Timeouts

Figure 16 shows a vary similar trend. While it took an average of 1429 milliseconds to process the 8410 targets with a single thread, this processing time falls dramatically when using the multi-threading capability, taking an average of as low as

only 8 milliseconds per host. Clearly, *clockvar*'s multi-threading offers an enormous

performance advantage over *clockdiff* – an average of 8 milliseconds per host (measuring

300 targets concurrently) versus an average of 11,187 milliseconds per host (measuring

just one target at a time).
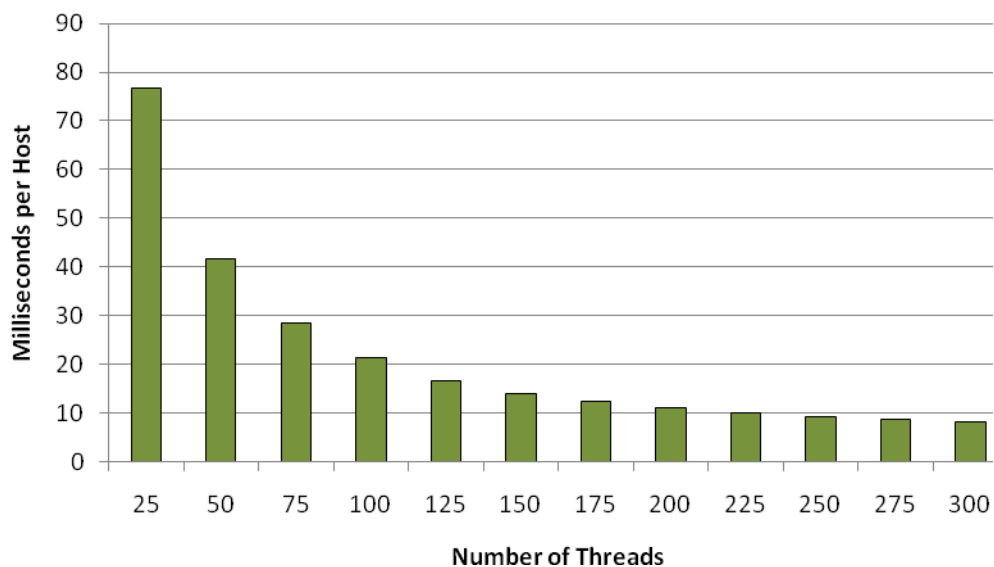
Figure 16: Average Processing Time Per Host

V. Conclusions and Future Work

In this chapter, we present the conclusions that we have drawn from this experiment. Our primary goal was to maintain a high degree of accuracy while drastically reducing the time it takes to measure the system time on remote hosts. A secondary goal was to prove that *clockdiff*'s output cannot always be trusted. Our results demonstrate that we have achieved these goals. We conclude with a brief summary of *clockvar*'s advantages over *clockdiff* and a discussion of several objectives for future work in this area.

5.1 Advantages of *Clockvar* over *Clockdiff*

A number of advantages in using *clockvar* over *clockdiff* are evident from this study. The first and most obvious is speed. Because *clockvar* does not send multiple packets to the target host, it returns results significantly faster than *clockdiff*. *Clockdiff* is preconfigured to use a series of 50 ICMP timestamp requests (or 50 ICMP echo requests with IP timestamp options), and we feel that such a large number is unnecessary. However, should a user wish to run multiple measurements, that can be done with *clockvar*; the number of packets to be sent to each target host can be passed into the program as a command line argument. Also, we have demonstrated that responses from all the servers in our experiment reach our test machine within 2 seconds of sending the request; thus we have concluded that 2 seconds is a reasonable timeout value – although this value can also be modified by the user. Thus *clockvar*, by default, will report after two seconds that a host is not responding, while *clockdiff* takes an average of over 10 seconds to make the same determination. Our experiments show that, on average, in a

single-threaded mode, *clockvar* yields a result 7.8 times faster than *clockdiff*. When the host is not responding to timestamp requests, *clockvar* reports this 5.4 times faster. When the host is answering, *clockvar* produces its output an average of 115.8 times faster than *clockdiff*. Again, this performance advantage is gained merely by running *clockvar* as a single-threaded application. When running against a large number of targets with 300 threads, for example, *clockvar* measures the system time on all of the targets an average of 1389.7 times faster than *clockdiff* does.

The second major advantage is flexibility. Like *clockdiff*, *clockvar* can be run against a single target. Unlike *clockdiff*, however, *clockvar* can read a list of targets from an input file and obtain timestamps from an unlimited number of targets in one run of the program. With *clockdiff*, the user specifies which timestamp option to use for the target (ICMP, IP with 4-term specified route, or IP with 3-term specified route). If the user wishes to try multiple options, this requires multiple runs of the program. With *clockvar*, the user has the ability to try any combination of the options (1, 2, or all) in the same run of the program.

Clockdiff only has the option to send its output to the screen (though, as with any program run on Linux, the output can be redirected). Screen output only is the default behavior for *clockvar* when run against only one target; however, when reading targets from an input file, *clockvar* can be instructed (via command line arguments) whether to display or suppress screen output, and whether to direct its output to files. File options include binary output (for efficient storage and machine processing) and human-readable output, which may be broken down into multiple files, including an error log, all program

output, and lists of targets that 1) responded to a particular option, 2) responded to any option, and 3) did not respond.

Although *clockdiff* uses multiple measurements of round-trip-time in an effort to estimate the network delay and provide an accurate result, it ultimately reports only a best guess of the difference in system time between the local and remote host. It displays the current time on the local host, the best round-trip times, and a delta value representing the time difference between the two hosts, but not the actual time on the target host. *Clockvar* reports a best guess as well, but it also displays the actual timestamp received from the remote host along with the delta and round-trip time. Thus the raw data from *clockvar* is available if the user wants to perform additional calculations. As we demonstrated in Section 4.3, *clockdiff* makes no report to the user when it encounters a timestamp indicating that the target host's system time differs by more than 12 hours from the local host's time and then arbitrarily adjusts the delta, showing a closer synchronization than may be the case in reality. Thus, without another source of measurement providing corroborating evidence, we ultimately cannot trust the accuracy of the delta that *clockdiff* reports.

5.2 Future Work

Our results show that *clockvar* is a promising alternative to using *clockdiff* to measure the system time on a remote host; however, the research in this area is not complete. We have several ideas about possibilities for future work, including other potential uses for this application.

The main goal of our study has been to further digital forensic science by providing a useful tool for measuring the system time on remote computers. Investigators can use *clockvar* to build a clock description of a remote host that is a source of time stamped data on a local machine so that they may form a reasonable hypothesis regarding its past behavior. This may prove essential in confirming or developing a precise timeline of events on a local machine or network.

Other uses are also possible. Buchholz and Tjaden propose that one may want to monitor the system time on each computer within one's own network. *Clockvar* could be used from a dedicated machine to do so quickly and efficiently on a regular basis for two purposes. First, this would guarantee that a highly accurate clock description of each host on the network would be immediately available if ever needed for an investigation. Second, the process could be used to generate an alarm if it perceives that the system clock on a host has been altered. This might play a part in intrusion or misuse detection (Buchholz and Tjaden, 2007).

Kohono et al. developed a technique for "fingerprinting" a computer on the Internet by carefully measuring its clock skew over time. Although the major part of their work focuses on measuring the TCP timestamp options clock (which is implemented within the network hardware, rather than maintained by the operating system), *clockvar* might be used to identify a host's signature by measuring the skew of the system clock. These researchers suggest that this technique could be useful not only in forensic analysis, but also in tracking a particular computer across the Internet, even when these it makes a connection from varying locations with different IP addresses (Kohono et al., 2006). Zander and Murdoch propose that using their technique of

synchronized sampling could enhance this method of remote host identification and tracking (Zander and Murdoch, 2008). *Clockvar* is capable of measuring the system time on a target host with just one packet, but the user may direct the program to utilize any number of packets. Thus, it may be possible to refine *clockvar*'s accuracy by incorporating a synchronized sampling algorithm when a large number of measurements are taken against a target host.

Ultimately, *clockvar* is a useful tool, but it cannot be used in all situations. The protocols for ICMP and IP timestamps messages require that the timestamps be formatted as the number of milliseconds since midnight UTC. Although the 32-bit timestamp field has enough space for a number representing over 24 days, only numbers less than 24 hours make any sense. Thus, for timestamp replies to be valid, the target hosts must maintain some degree of synchronization with standard time. Additionally, the protocols specify that, if timestamps are not being transmitted in a standard format, the high-order bit is to be set. However, even if we receive a timestamp reply with this bit set, we know only that it is "non-standard," but we don't know how we should otherwise interpret the number. Also, not all hosts play by the rules; for example, they may send a non-standard response without flagging it as non-standard.

Furthermore, many professional network administrators configure their servers not to respond to ICMP messages for security reasons. Neither *clockdiff* nor *clockvar* can measure the time on a host if they do not receive timestamp replies. We also found by experiment that the IP timestamp options don't work when the host transmitting the timestamp requests resides behind a NAT device. In this case, *clockvar* and *clockdiff* register the IP address of the transmitting host inside the IP options portion of the IP

header; however, although the kernel on this host places its own address in the source address portion of the regular IP header, this value is overwritten by the NAT host when the packet passes through it on the way to the target host. We hypothesize that the target host drops the IP timestamp request packet when it inspects the header and sees that the address of the host requesting the timestamp (within the IP options) does not match the address of the host from which it received the request (i.e., the NAT host). Thus, further research needs to be done on finding and refining means of measuring the system time on remote hosts using NAT.

Finally, additional comparisons between the results obtained by clockdiff and clockvar should be made. Our experiments demonstrated that clockdiff manipulates the delta when this value is larger than 12 hours; however, we are not able to state with certainty whether this adjustment was right or wrong in each situation. Clockdiff's author incorporated this adjustment to prevent the program from reporting an enormous delta when measurements are taken very close to midnight and the two hosts timestamps reflect the number of milliseconds past UTC on different days. However, we question whether this should be done in all situations when the delta appears to be larger than 12 hours. For instance, say the local host's time is 19 hours past midnight UTC, and it receives a timestamp of 6 hours past midnight UTC from the target host. Is it more likely that the hosts' clocks are set to the same day and the target is 13 hours behind the local host, or should we assume (as clockdiff does) that the target host's clock is 11 hours ahead of the local host (and is set to the next day)? We do not believe that this is a safe assumption.

Additional experiments that may shed more light on this issue should involve adjusting the time of day at which the measurements are taken. For our 105-day experiment, we initiated the shell script at 1 a.m. EST (06:00:00 UTC). On most days, the shell script ran for around 8 and a half hours, so all measurements took place between 06:00:00 UTC and around 14:30:00 UTC. It would be very interesting to see if we discover a varying number of "clockdiff" outliers as we vary the time of measurements from very close to midnight UTC to as far away as possible from midnight UTC.

References

Boyd Chris, Forster Pete. Time and date issues in forensic computing – a case study. Digital Investigation 2004;1(1):18-23.

Buchholz Florian, Tjaden Brett. A Brief Study of Time. Digital Investigation 2007; 4S:31-42.

Carrier B, Spafford E. Defining Event Reconstruction of Digital Crime Scenes. Journal of Forensic Science, November 2004: Vol. 49, No. 6.

Chow K.P., Frank Y.W. Law, Michael Y.K. Kwan, Pierre K.Y. Lai. The Rules of Time on NTFS File System. Second International Workshop on Systematic Approaches to Digital Forensic Engineering, 2007: 71-75.

DMOZ Top Listed Domains, <http://www.domaintools.com/internet-statistics/dmoz-listings.php> [accessed October 2006].

Fielding R, Gettys J, Mogul J, Frystyk H, Masinter L, Leach P, et al. Hypertext Transfer Protocol – HTTP/1.1. Technical Report RFC 2616, Internet Society; June 1999. <ftp://ftp.isi.edu/in-notes/rfc2616.txt>.

Kiernan Jerry, Terzi Evimaria. Constructing Comprehensive Summaries of Large Event Sequences. Proceeding of the 14th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, 2008: 417-425.

Kohno T, Broido A, Claffy KC. Remote Physical Device Fingerprinting. In: IEEE Symposium on Security and Privacy, Oakland, CA, May 2005.

Kuznetsov, Alexey. *Clockdiff*(8). System Manager's Manual: iputils, January 2008.

List of Stratum 1 NTP servers. http://support.ntp.org/bin/view/Servers/StratumOneTimeServers [accessed January 2009]

McAlpin John. U.S. Energy Secretary Says Weeks Needed to Analyze Blackout Data. The Standard Times, <http://www.southcoasttoday.com/daily/08-03/08-28-03/a19wn082.htm> August 2003.

Mills D. Network Time Protocol (version 3): Specification, Implementation and Analysis. Technical Report RFC 1305, Network Working Group; March 1992.

Mills D. RFC 4330 (SNTP) January, 2006. The Internet Society.
           <http://www.ietf.org/rfc/rfc4330.txt>

Minar Nelson. A Survey of the NTP Network,
           <http://www.media.mit.edu/wnelson/research/ntp-survey99/>; December 1999.

Palmer G. A Road Map for Digital Forensic Research. Technical Report DTR-T001-01,
           The MITRE Corporation, August 2001.

Paxson V. On Calibrating Measurements of Packet Transit Times. Conference on
           Measurement and Modeling of Computer Systems, 1998;11–21.

Postel J. Internet Protocol. Technical Report RFC 791. Information Sciences Institute,
           University of Southern California , September 1981. (Postel 1981a)

Postel J. Internet Control Message Protocol. Technical Report RFC 792. Network
           Working Group, September 1981. (Postel 1981b)

Schatz Bradley, Mohay George, Clark Andrew. A Correlation Method for Establishing
           the Provenance of Timestamps in Digital Evidence. In: Proceedings of the 6th
           Annual Digital Forensic Research Workshop, August 2006.

Stevens Malcolm. Unification of Relative Time Frames for Digital Forensics. Digital
           Investigation, 2005;1(3):225–39.

Symmetricom: Timing, Test, and Measurement Division. How Time Finally Caught up
           With the Power Grid, <http://www.symmttm.com/pdf/Gps/wp_PowerGrid.pdf>,
           2004.

Symmetricom: Timing, Test, and Measurement Division. Stochastic Model Estimation
           of Network Time Variance,
           <http://www.symmttm.com/pdf/Network_Timing/wp_Stochastic_Model.pdf>.

Taylor B., Mohr P.. The NIST Reference on Constants, Units, and Uncertainty. Last
           updated October 2000. http://physics.nist.gov/cuu/Units/ [accessed Jan 2009]

United States Naval Observatory: Astronomical Applications Department. What is
           Universal Time?, <http://aa.usno.navy.mil/faq/docs/UT.html>; 2003.

Willasen Svein. Timestamp Evidence Correlation by Model Based Clock Hypothesis
           Testing. e-Forensics, January 2008: 21-27.

Zander S., Murdoch J. An Improved Clock-skew Measurement Technique for Revealing
           Hidden Services. 17th USENIX Security Symposium, San Jose, CA, 2008.