View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by AFTI Scholar (Air Force Institute of Technology)

Air Force Institute of Technology

AFIT Scholar

Theses and Dissertations

Student Graduate Works

3-14-2014

# Analysis of Effects of Sensor Multithreading to Generate Local System Event Timelines

Daniel M. Gallagher

Follow this and additional works at: https://scholar.afit.edu/etd

## Recommended Citation

Gallagher, Daniel M., "Analysis of Effects of Sensor Multithreading to Generate Local System Event Timelines" (2014). *Theses and Dissertations*. 601.
https://scholar.afit.edu/etd/601

This Thesis is brought to you for free and open access by the Student Graduate Works at AFIT Scholar. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of AFIT Scholar. For more information, please contact richard.mansfield@afit.edu.

**ANALYSIS OF EFFECTS OF SENSOR MULTITHREADING TO GENERATE LOCAL SYSTEM EVENT TIMELINES**

THESIS

Daniel M. Gallagher, Captain, USAF

AFIT-ENG-14-M-31

**DEPARTMENT OF THE AIR FORCE**
**AIR UNIVERSITY**

***AIR FORCE INSTITUTE OF TECHNOLOGY***

**Wright-Patterson Air Force Base, Ohio**

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

The views expressed in this thesis are those of the author and do not reflect the official policy or position of the United States Air Force, the Department of Defense, or the United States Government.

This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-14-M-31

# ANALYSIS OF EFFECTS OF SENSOR MULTITHREADING TO GENERATE LOCAL SYSTEM EVENT TIMELINES

THESIS

Presented to the Faculty

Department of Electrical and Computer Engineering

Graduate School of Engineering and Management

Air Force Institute of Technology

Air University

Air Education and Training Command

in Partial Fulfillment of the Requirements for the

Degree of Master of Science in Cyberspace Operations

Daniel M. Gallagher, B.S.C.S.I.A.

Captain, USAF

March 2014

DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE;
DISTRIBUTION UNLIMITED

AFIT-ENG-14-M-31

ANALYSIS OF EFFECTS OF SENSOR MULTITHREADING TO GENERATE LOCAL SYSTEM EVENT TIMELINES

Daniel M. Gallagher, B.S.C.S.I.A.
Captain, USAF

Approved:

| //signed// | 20 Feb 2014 |
|---|---|
| Maj Thomas E. Dube, PhD (Chairman) | Date |
| //signed// | 20 Feb 2014 |
| Barry E. Mullins, PhD (Member) | Date |
| //signed// | 20 Feb 2014 |
| Timothy H. Lacey, PhD (Member) | Date |

AFIT-ENG-14-M-31


## Abstract

In practice, organizations with their own information technology infrastructure normally log or otherwise monitor network information at boundary routers and similar network devices that are log-capable. However, not all organizations opt to log local system information, such as an employee's organization-owned workstation activity. This research explores one approach to logging pertinent local system information using multithreading and free software designed for such logging purposes as well as utilities that come with the Microsoft Windows® 7 Operating System.

Research focuses on file downloads on the local system and combines the aforementioned pieces of software into an event logging suite. The event logging suite consists of four different sensors and utilizes multithreading in an attempt to effectively capture as many pertinent events as possible, with the ultimate goal of capturing 100% of the events in chronological order of actual occurrence. Specifically, the event logging suite increases the number of processes and thus threads that two of the four sensors, Windows® `NETSTAT` and `tasklist` utilities respectively, in the suite execute in order to determine the optimal settings for the two sensors. To add some realism to the experiments, this research implements three different system loads to simulate user activity on the system while a scripted file-download scenario executes and the logging suite actively captures events.

Ultimately, the performance accuracies of the `NETSTAT` and `tasklist` sensors across numerous tests show that while the sensors can capture above 85% of the expected number of events, neither are capable of consistently achieving this accuracy, even under a low system load.

# Acknowledgments

First, I would like to thank my parents for providing me with their support and words of motivation when this task seemed insurmountable. I would also like to thank my academic/thesis advisor, Major Thomas Dube, for guiding me throughout this endeavor. Additionally, I would like to thank Dr. Timothy Lacey and Dr. Barry Mullins for generously being a part of my research committee. Finally, I would like to thank all of my friends and fellow classmates for listening to me on "those days when it seemed that nothing worked properly."

Daniel M. Gallagher

## Table of Contents


Page

Abstract . . . . . . . . . . iv

Acknowledgments . . . . . . . . . . v

Table of Contents . . . . . . . . . . vi

List of Figures . . . . . . . . . . ix

List of Tables . . . . . . . . . . xi

I. Introduction . . . . . . . . . . 1

1.1 Background . . . . . . . . . . 1
1.2 Research Goals and Objectives . . . . . . . . . . 2
1.3 Assumptions and Limitations . . . . . . . . . . 2
1.4 Implications . . . . . . . . . . 2
1.5 Preview . . . . . . . . . . 3

II. Literature Review . . . . . . . . . . 4

2.1 Chapter Overview . . . . . . . . . . 4
2.2 Logs . . . . . . . . . . 4
2.2.1 Simplified Categorization of Log Types . . . . . . . . . . 4
2.2.2 Importance of Logs . . . . . . . . . . 6
2.2.3 Log Collection, Centralized Logging . . . . . . . . . . 8
2.2.4 Log Analysis . . . . . . . . . . 9
2.2.5 Log Analysis Challenges . . . . . . . . . . 10
2.3 Windows® Thread Scheduling . . . . . . . . . . 11
2.3.1 Overview . . . . . . . . . . 12
2.3.2 Thread States and Priority Levels . . . . . . . . . . 12
2.3.3 Dispatcher Database . . . . . . . . . . 14
2.3.4 Quantums . . . . . . . . . . 15
2.3.5 Thread Priority Boosts . . . . . . . . . . 16
2.3.6 Context Switching . . . . . . . . . . 17
2.3.7 Other Notable Scheduling Concepts . . . . . . . . . . 18
2.4 Summary . . . . . . . . . . 19

Page

III. Methodology . . . . . 21

3.1 Chapter Overview . . . . . 21
3.2 Goals . . . . . 21
3.3 Approach . . . . . 21
3.4 System Boundaries . . . . . 25
3.5 System Services . . . . . 26
3.6 Workload . . . . . 27
3.7 Performance Metrics . . . . . 27
3.8 System Parameters . . . . . 28
3.9 Factors . . . . . 29
3.10 Evaluation Technique . . . . . 29
3.11 Experimental Design . . . . . 31
3.12 Summary . . . . . 33

IV. Analysis and Results . . . . . 34

4.1 Chapter Overview . . . . . 34
4.2 Experimental Results . . . . . 34
4.2.1 `NETSTAT` Sensor . . . . . 34
4.2.1.1 `NETSTAT` Low Load Results . . . . . 34
4.2.1.2 `NETSTAT` Medium Load Results . . . . . 38
4.2.1.3 `NETSTAT` High Load Results . . . . . 42
4.2.2 `tasklist` Sensor . . . . . 46
4.2.2.1 `tasklist` Low Load Results . . . . . 47
4.2.2.2 `tasklist` Medium Load Results . . . . . 51
4.2.2.3 `tasklist` High Load Results . . . . . 55
4.3 Summary . . . . . 60

V. Conclusions and Recommendations . . . . . 61

5.1 Chapter Overview . . . . . 61
5.2 Conclusions of Research . . . . . 61
5.3 Research Contributions . . . . . 62
5.4 Recommendations for Future Research . . . . . 62

Appendix A: Miscellaneous Figures . . . . . 64

Appendix B: Complete Performance Summaries . . . . . 65

Bibliography . . . . . 95

Page

Vita . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 99

## List of Figures


Figure — Page

2.1 Logging and analysis architecture Huang et al. created [17] . . . . . . . . . . . 9

2.2 Mapping of Windows® Kernel Priorities to the Windows® API as shown in [37] 13

2.3 Simplified version of thread states and transitions as shown in [37] . . . . . . . 15

3.1 Example of sensor entries . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

3.2 System Under Test (SUT) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

3.3 Separating linked events into separate text files . . . . . . . . . . . . . . . . . 30

3.4 Example of determining ordering accuracy . . . . . . . . . . . . . . . . . . . . 32

4.1 Mean percentages of events captured while increasing maximum number of `NETSTAT` processes by 1 (low load) . . . . . . . . . . . . . . . . . . . . . . . . 37

4.2 Mean ordering frequencies while increasing maximum number of `NETSTAT` processes by 1 (low load) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 39

4.3 Mean percentages of events captured while increasing maximum number of `NETSTAT` processes by 1 (medium load) . . . . . . . . . . . . . . . . . . . . . . 41

4.4 Mean ordering frequencies while increasing maximum number of `NETSTAT` processes by 1 (medium load) . . . . . . . . . . . . . . . . . . . . . . . . . . 43

4.5 Mean percentages of events captured while increasing maximum number of `NETSTAT` processes by 1 (high load) . . . . . . . . . . . . . . . . . . . . . . . 46

4.6 Mean ordering frequencies while increasing maximum number of `NETSTAT` processes by 1 (high load) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 47

4.7 Mean percentages of events captured while increasing maximum number of `tasklist` processes by 1 (low load) . . . . . . . . . . . . . . . . . . . . . . . . 50

4.8 Mean ordering frequencies while increasing maximum number of `tasklist` processes by 1 (low load) . . . . . . . . . . . . . . . . . . . . . . . . . . . . 52

Figure Page


4.9 Mean percentages of events captured while increasing maximum number of tasklist processes by 1 (medium load) . . . . . 54

4.10 Mean ordering frequencies while increasing maximum number of tasklist processes by 1 (medium load) . . . . . 56

4.11 Mean percentages of events captured while increasing maximum number of tasklist processes by 1 (high load) . . . . . 58

4.12 Mean ordering frequencies while increasing maximum number of tasklist processes by 1 (high load) . . . . . 59

A.1 Breakdown of the tcpdump command used within the experiment . . . . . 64

B.1 Mean percentages of events captured for all sensors when increasing NETSTAT by 1 (low load) . . . . . 65

B.2 Mean percentages of events captured for all sensors when increasing NETSTAT by 1 (medium load) . . . . . 70

B.3 Mean percentages of events captured for all sensors when increasing NETSTAT by 1 (high load) . . . . . 75

B.4 Mean percentages of events captured for all sensors when increasing tasklist by 1 (low load) . . . . . 80

B.5 Mean percentages of events captured for all sensors when increasing tasklist by 1 (medium load) . . . . . 85

B.6 Mean percentages of events captured for all sensors when increasing tasklist by 1 (high load) . . . . . 90

## List of Tables


Table — Page

4.1 Performance summary of NETSTAT capture accuracy under a low load . . . . . 35

4.2 Performance summary of NETSTAT capture accuracy under a medium load . . . 40

4.3 Performance summary of NETSTAT capture accuracy under a high load . . . . . 45

4.4 Performance summary of tasklist under a low load . . . . . . . . . . . . . . 48

4.5 Performance summary of tasklist under a medium load . . . . . . . . . . . 53

4.6 Performance summary of tasklist capture accuracy under a high load . . . . 57

B.1 Complete performance summary of NETSTAT under a low load (1 of 4) . . . . . 66

B.2 Complete performance summary of NETSTAT under a low load (2 of 4) . . . . . 67

B.3 Complete performance summary of NETSTAT under a low load (3 of 4) . . . . . 68

B.4 Complete performance summary of NETSTAT under a low load (4 of 4) . . . . . 69

B.5 Complete performance summary of NETSTAT under a medium load (1 of 4) . . 71

B.6 Complete performance summary of NETSTAT under a medium load (2 of 4) . . 72

B.7 Complete performance summary of NETSTAT under a medium load (3 of 4) . . 73

B.8 Complete performance summary of NETSTAT under a medium load (4 of 4) . . 74

B.9 Complete performance summary of NETSTAT under a high load (1 of 4) . . . . 76

B.10 Complete performance summary of NETSTAT under a high load (2 of 4) . . . . 77

B.11 Complete performance summary of NETSTAT under a high load (3 of 4) . . . . 78

B.12 Complete performance summary of NETSTAT under a high load (4 of 4) . . . . 79

B.13 Complete performance summary of tasklist under a low load (1 of 4) . . . . 81

B.14 Complete performance summary of tasklist under a low load (2 of 4) . . . . 82

B.15 Complete performance summary of tasklist under a low load (3 of 4) . . . . 83

B.16 Complete performance summary of tasklist under a low load (4 of 4) . . . . 84

B.17 Complete performance summary of tasklist under a medium load (1 of 4) . 86

Table Page

B.18 Complete performance summary of `tasklist` under a medium load (2 of 4) . 87

B.19 Complete performance summary of `tasklist` under a medium load (3 of 4) . 88

B.20 Complete performance summary of `tasklist` under a medium load (4 of 4) . 89

B.21 Complete performance summary of `tasklist` under a high load (1 of 4) . . . 91

B.22 Complete performance summary of `tasklist` under a high load (2 of 4) . . . 92

B.23 Complete performance summary of `tasklist` under a high load (3 of 4) . . . 93

B.24 Complete performance summary of `tasklist` under a high load (4 of 4) . . . 94

# ANALYSIS OF EFFECTS OF SENSOR MULTITHREADING TO GENERATE LOCAL SYSTEM EVENT TIMELINES

## I. Introduction

### 1.1 Background

Logs are a means of record-keeping. People use logs in many parts of their everyday life, such as in the form of diaries or journals. Another log example takes place in the field of crime scene investigation, specifically referring to the chain of custody log [1–3]. These are two examples of seemingly basic, yet arguably important logs. More often than not, logs in the information technology (IT) field serve a similar yet more technical purpose, depending on the type of log.

In practice, organizations with their own information technology infrastructure normally log or otherwise monitor network information at boundary routers and similar network devices that are log-capable. However, not all organizations may choose to log local system information. Several reasons may exist as to why not all organizations opt to log local system information. Likewise, several reasons may exist as to why not all organizations opt to log local system information for the purpose of logging events pertaining to specific file downloads. One such reason may be that the organization is simply not concerned with logging the aforementioned type of information. However, another reason may be that a replicable method for logging data pertaining to specific file downloads does not exist. In fact, this thesis literature review did not locate any works that used sensor multithreading (as this research does) for the purpose of logging events pertaining to specific file downloads.

## 1.2 Research Goals and Objectives

The goal of this research is to investigate and subsequently identify a method that utilizes multithreading for producing a timeline consisting of locally-produced and collected network and system-related events pertaining to file downloads. Specifically, this research looks at how differences in thread allocation amongst the sensors, as well as imposed system and network-traffic workloads, affect the accuracy of the resulting timeline. The contributions of this research are:

- an exploration into the effects of sensor multithreading,
- an application and subsequent statistical analysis of sensor multithreading for the purpose of logging events pertaining to specific file downloads, and
- finding the optimal maximum thread count for the `NETSTAT` and `tasklist` sensors within the logging suite for the scenario and values tested.

## 1.3 Assumptions and Limitations

This research only focuses on the end user downloading files to a predetermined, and thus known, directory. Also known beforehand are the downloaded files, along with the internet protocol (IP) addresses of the web servers hosting the downloaded files. The sensors within this experiment, which are in the form of Python v3.3.2 [4] scripts in addition to other utilities discussed later in this thesis, utilize the information contained within the first two sentences of this section verbatim, such as the exact IP addresses of the web servers. The hard-coding of the previously highlighted information within the Python scripts limits the application of this logging suite to this particular file-download scenario.

## 1.4 Implications

The implications of this research are significant. The application of the event logging suite utilized 100% of the central processing unit (CPU) for the duration of the time that

it was logging local system events pertinent to the experiment. The statistical analysis indicates that the `NETSTAT` sensor has the capability to be accurate up to ~100% under low system loads. The statistical analysis also indicates that the `tasklist` sensor has the capability to be accurate up to 89.3% under low system loads. However, under medium and high system loads, both sensor's overall accuracies plummet by up to 88.6% in comparison to their overall accuracies under low system loads. Unfortunately, even under low system loads, both sensors' capture accuracies are inconsistent and thus rule out use of this method in a real world setting.

## 1.5 Preview

The *Literature Review* presents pertinent literature regarding log capturing and storing techniques. Additionally, the *Literature Review* also reviews some current log analysis techniques and log analysis challenges. The *Methodology* chapter presents the approach to research experiments and also covers all pertinent details pertaining to the experiments. Chapter IV, the *Analysis and Results*, presents the results of the research experiments. It also provides an interpretation and analysis of the experiment's results. Finally, the *Conclusion* summarizes the main points of thesis. Additionally, the *Appendices* contains other miscellaneous figures and the remaining performance summaries omitted from the *Analysis and Results* chapter.

# II. Literature Review

## 2.1 Chapter Overview

This chapter summarizes pertinent literature regarding logs and log analysis and briefly covers several live capture methods. The first section introduces logs in an IT context and briefly describes the different types of logs. Section 2.2.2 highlights the importance of logs when it comes to, but is not limited to, attribution and computer forensics investigations. Successive sections describe centralized logging, two current log analysis techniques, and current challenges pertaining to log analysis. Finally, the chapter concludes with a summarization of the main points of this literature review.

## 2.2 Logs

The creator of the log can make the contents of the log as basic, detailed, or technically in-depth as needed. This level of customization is no exception regarding logs within IT assets, whether the host operating systems (OS), hardware and software firewalls, network hardware, or any type of server, generates the log [5]. For the purposes of their research on digital crime investigation, Gupta and Meena [5] specify four types of logs, specifically: firewalls, network devices, web servers, and a category for other miscellaneous logs. The following section simplifies and re-categorizes Gupta and Meena's log categorization and briefly describes each item in order to provide the reader with a general overview of the different types of logs.

### *2.2.1 Simplified Categorization of Log Types.*

***Application Logs.*** During the application development phase, these logs serve as an aid to application developers. These application logs provide the developer(s) with vital crash report information helping them to narrow the scope of the origin of the issue [6–9]. Post-release, individual applications that run on the OS may generate and store their own

log files on the local hard drive, which are accessible by the end-user [7, 9]. However, released applications do not always generate logs for the end-user to see. For example, Mozilla Firefox [10] makes use of application crash reports by giving the end-user the option of whether or not they want the crash report sent to Mozilla [8] for review and subsequent mitigation. In this way, the end-user never actually sees the logs generated by Mozilla Firefox. These logs are useful during the application (software) development phase and also post-release.

***Hardware and Network Logs.*** This category encompasses both hardware device logs as well as network logs as they often track the same types of information. Two systems that generate these types of logs are Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS). IDS and IPS can exist in either hardware or software form, but one of their purposes is to monitor network traffic to detect or prevent intrusions [11–13]. Other various network infrastructure components include firewalls, hubs and repeaters, routers, and switches. These components may have readily available log files, while others may not.

***OS Logs.*** The OS logs a number of different events, ranging from system boots and system shutdowns to invalid log-on attempts [14]. The information collected varies depending on the administrator preferences. The OS logs each event and assigns each logged event its own log entry. For example, when a user logs into the system, this action creates a single log entry with the information specific to that particular event. Microsoft Windows® provides the user with a tool called the Microsoft Management Console [15]. Within the Management Console are various administrative snap-ins, one of which is the Event Viewer, which graphically categorizes and lists all of the logged events that Windows® captures [16]. The data collected, specific formatting, ultimate presentation of the data to the end-user, and a number of other traits vary across different OSes. For

example, while Windows® provides the user with a graphical representation of its event logs, most flavors of Linux provide the user with text documents.

#### *2.2.2 Importance of Logs.*

A number of peer-reviewed works on logs highlights the importance of logs [17, 18]. The two aforementioned works both reference the same 2009 Data Breach Investigations Report (DBIR) that was first put together and published by Verizon Communications' Research, Investigations, Solutions, and Knowledge, (RISK) Team [19] in 2008. According to the RISK team's website, the team gathers, interprets, and provides feedback on security incidents. The RISK Team publicly releases an annual DBIR. Each report since 2008 references log data one or more times.

- The 2008 DBIR [20] highlights the monitoring of event logs as being one of three items of importance. The second item of importance pertains to "Anti-Forensics", which Section 2.2.5 of this literature review discusses in greater detail. The third instance lists the monitoring of event logs as one of the main recommendations of the report. The report also states that of the data breaches report on, the logs contained evidence of events leading up to 82% of those data breaches. This means that preventing 82% of the data breaches reported on were preventable.

- Both [17, 18] reference the 2009 DBIR [21], which contains one notable instance pertaining to logs. The 2009 report states that of the data breaches reported on, the logs contained evidence of events leading up to 66% of those data breaches.

- The 2010 DBIR [22] contains four notable instances pertaining to logs. The first instance is a repeat of monitoring and mining of event logs as being one of seven items of importance. The second instance mentions that the RISK Team observed companies that were pre-occupied with deploying system patches rather than reviewing log files. The third instance is a response by the RISK Team to the

individuals that were subject to data breaches. Trying to locate malicious activity amongst the sheer number of log entries is a daunting task. However, the RISK Team states that instead of looking for needles in haystacks, look solely for haystacks as the haystacks may contain several red flags indicating a data breach. For example, the RISK Team noted that the number of log entries increased by 500% following a data breach. Additionally, the RISK Team noted that SQL injections and various other attacks leave longer log entries that the average or normal log entry. These two examples are what the RISK Team considers "haystacks" as opposed to "needles in a haystack". Lastly, the 2010 report states that of the data breaches report on, the logs contained evidence of events leading up to 86% of those data breaches. Much like the 2009 report, in this instance, the opportunity existed (at one time or another) to prevent 86% of the data breaches discussed.

- The 2011 DBIR [23] mentions the monitoring and mining of event logs as being one of eight items (up from seven in the 2010 report) of importance regarding data breach mitigation.

- The 2012 DBIR [24] reiterates the "haystack" terminology as previously mentioned in the 2010 DBIR. In this DBIR, the RISK Team states that anomalous behavior can be in the form of drastic increases in log data seen. Conversely, the RISK Team also states that anomalous behavior can be in the form of drastic decreases or entire gaps in log data seen.

Additionally, log files, among other forms of digital evidence, are of significant use in digital forensic investigations [25–28]. The integration of computers and mobile devices into everyday life means that they are a means of producing pieces of evidence in criminal investigations. The main use of digital evidence comes in the form of establishing a timeline, which is inarguably one of the most important parts of reconstructing a crime.

One such tool that creates a timeline based on log files is `log2timeline`. Section 2.2.4 highlights `log2timeline` in more detail.

### *2.2.3 Log Collection, Centralized Logging.*

A number of different works [17, 29, 30] make mention of centralized logging as one logging technique. Centralized logging utilizes a logging server that is separate from the actual device(s) doing the live logging. While the actual devices accomplishing the live logging may continue to store their logs in whatever manner the administrator of the devices establishes, these devices ultimately send the logs directly to a centralized logging server. Encrypting the server and the connection adds an additional layer of security.

This technique thwarts attackers' attempts to destroy log evidence after successfully breaking into a device [29]. The alteration of logs by attackers is one such challenge pertaining to log analysis. Section 2.2.5 describes the challenges of logging in greater detail. Again, depending on how the administrator configures their devices, the attacker may find that either the system contains logs that they can edit or the attacker may find the system does not contain any logs. The logs stored on this centralized logging server hold the un-altered versions of the logs and the administrator then has the option and the ability to determine whether or not the logs kept on the local device contain any alterations.

Figure 2.1 shows a system for centralized logging and subsequent analysis of logs, which Huang et al. proposed [17]. Huang et al. use the Shanghai Education and Research Network (SHERNET) as their test subject. SHERNET is a Metropolitan Area Network (MAN) and a conglomerate of all universities located within Shanghai and consists of different node sites for each university. Each node side consists of a number of network devices in order to facilitate the demands of so many users. In order to transmit all of the network device logs, Huang et al. make use of the `syslog` protocol [31] on their centralized logging server. All network devices in SHERNET transmit their activity logs to this centralized logging server.

The centralized logging server is responsible for sorting through all of the raw logs and parsing out all of the important information so that the individuals in charge of reviewing the logs will only need to sift through what Huang et al. consider to be the important log entries. A MySQL database stores logs sent by the centralized logging server, which Huang et al. later use for statistical analysis. In addition to sending logs to a MySQL database, the centralized logging server also passes the log entries to what Huang et al. refer to as a "real-time alarm" module. Huang et al. pre-program this module with a number of different rules, which are similar to firewall rules. If the real-time alarm receives a log entry that violates one of the pre-programmed rules, then the individuals that are in charge of sifting through logs receive notifications of the rule violation and can take appropriate action.

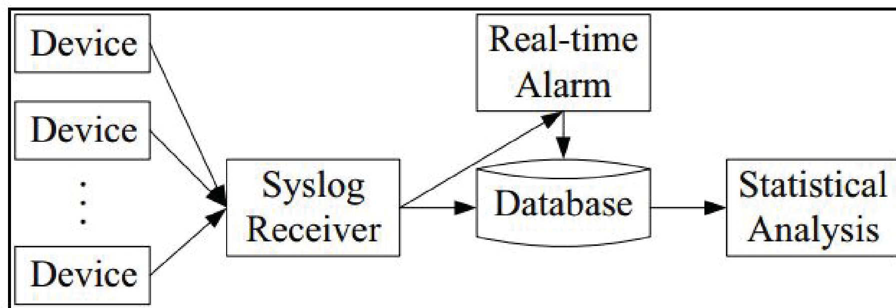


Figure 2.1: Logging and analysis architecture Huang et al. created [17]

### *2.2.4 Log Analysis.*

***Event Correlation***. The idea of linking like or seemingly unlike events together in order to generate one single past event is not a new concept. Myers et al. describe event correlation as a subset of log analysis [18]. Accordingly, enterprise-level networks are using event correlation more frequently than in the past as Myers et al. note. Myers et al. argue that separation of the log collection and event correlation processes is cheaper than centralized log analysis. They also argue that centralized log analysis is "difficult" because the centralized database may have trouble interpreting differences in transmitted

log messages because not all logs have the same syntax. However, Myers et al. also highlight some disadvantages pertaining to the use of event correlation as a log analysis technique. One such disadvantage pertains to the distribution of event correlation systems. Unlike in a centralized log system, event correlation systems do not have a central location, which means that applying changes, such as an update, occur in multiple locations instead of one location. Regardless, individual preference among other factors determine whether an entity uses centralized log analysis or event correlation.

***Establishing a Timeline***. As previously mentioned, nearly all IT device types have the ability to generate activity logs. These activity logs aid in criminal cases, either helping to prove guilt or innocence by aiding in establishing the chronological timeline of a crime. For example, as referenced in [32], the defense team in the Casey Anthony murder trial used Mozilla Firefox browser cache evidence in an attempt to show that their client (Casey Anthony) did not spend a significant amount of time browsing content relating to chloroform, which was of significant importance in this case. Note that cache entries contain timestamps that show specific item access times. The importance of this case is that it demonstrates the application of digital evidence in court cases as a means of establishing not only a timeline, but also attempting to aid the judge and jury in making a decision on whether the defendant is guilty or innocent.

#### 2.2.5 Log Analysis Challenges.

***Anti-Forensics***. Anti-forensics is the destruction or otherwise modification of data or any form of record, such as system logs, in an attempt to conceal a malicious event. The motive behind destroying this data is an attempt by the perpetrator(s) to make putting together what happened, in addition to who many have committed the malicious act, difficult for an investigative team. An example of anti-forensics is changing the *date modified* field of a particular file to a date prior to the occurrence of the malicious event. A tool called `Timestomp`, created by James Foster and Vincent Liu [33], makes this

particular anti-forensic tactic possible. `Timestomp` allows the user to either modify or completely erase the timestamps pertaining to targeted files. A number of peer-reviewed works pertaining to anti-forensics [34–36] reference `Timestomp`. In fact, [35] states that `Timestomp`, "renders time stamping recovered by forensic tools unreliable in court." In essence, this means `Timestomp` tampers with potential evidence and thus *none of the evidence is of use* in court.

***Quantity over quality***. The sheer amount of log entries to sift through on a daily basis is one of the main challenges of log analysis. The amount of logs entries generated depends on the pre-defined setup. In the previously referenced annual data breach reports (DBIR) that Verizon's RISK Team published [20–24], one of the main complaints by responsible individuals pertains to too many log entries to review. However, as the reports point out, properly reviewing the logs prevents a considerable percentage of data breaches. The previous statement leads to the next log analysis challenge, *lack of understanding*.

***Lack of understanding***. Knowing what specifically to look for within log data is another challenge of log analysis. The previous statement directly relates to the commonly-referenced "needle(s) in a haystack" metaphor [22, 24]. Essentially, the individuals reviewing the logs are looking for single log entries that are blatantly malicious, but the malicious log entries may not appear suspicious. Another issue pertaining to lack of understanding is that the reviewer may be looking for single malicious activities when there may be an anomalous increase in the number of log entries over the course of a month instead of appearing as one single malicious entry.

### 2.3 Windows® Thread Scheduling

The logging suite introduced in this research spawns thousands of threads within each test. This section of the literature review summarizes how Windows® 7 schedules threads. Additionally, this section describes how the OS manages processes and threads, which are relevant to this research.

### *2.3.1 Overview.*

Windows® 7 utilizes a preemptive scheduling system based on thread priorities [37]. At least one of the highest-priority threads that is ready to run, executes at any given time. However, *processor affinity* decides which processor a thread runs on. Processor affinity may actually limit a given thread to run on one particular processor. The process that spawns the thread decides the processors on which the thread runs.

The thread runs for a predetermined amount of time referred to as a *quantum*. However, if a higher priority thread becomes ready to run, the higher priority thread then preempts the lower priority, currently running thread, cutting the lower priority thread's quantum short. A *context switch* occurs when a new thread becomes ready to run, such as in the previous instance. A context switch includes saving the state of the previously running thread and subsequently loading the new thread's state and then executing the new thread. If no higher priority thread is ready to run, then the thread finishes its quantum and then the next highest priority thread begins executing for its time quantum. This procedure continues as long as a runnable thread exists, or until Windows® shuts down. Subsections 2.3.2 through 2.3.7 delve into greater detail, expanding on this overview.

### *2.3.2 Thread States and Priority Levels.*

Windows® 7 utilizes 32 thread priority levels in order to establish when each thread runs. Two separate categories share these 32 priority levels. Thread priority levels 0 through 15 make up the *variable* category, while thread priority levels 16 through 31 make up the *real-time* category. However, Windows® reserves level 0 for the *zero page thread*. According to [38], the zero-page thread is a "system thread responsible for zeroing any free pages when there are no other threads that need to run."

Furthermore, Windows® 7 assigns thread priority levels from both the Windows® API and the Windows® kernel. Essentially, a process within Windows® has a priority class, which ranges from highest to lowest priority: Real-time, High, Above Normal, Normal,

Below Normal, and Idle. Windows® then assigns *relative priority levels* to each individual thread within that process, which range from highest to lowest priority: Time-critical, Highest, Above-normal, Normal, Below-normal, Lowest, and Idle. Windows® also assigns numerical ranks to both sets of priorities in order for the kernel to ultimately calculate when any given thread should run. Figure 2.2 shows the result of this procedure. The following two sentences describe a hypothetical scenario that serves as an example to explain Figure 2.2. On a uniprocessor system, if a thread with a priority class of *Normal* and a relative priority of *Normal* is running and a thread with an *Above Normal* priority class and a *Normal* relative priority becomes ready to run before the currently running thread's quantum finishes, then Windows® performs a context switch as the *Above Normal* thread preempts the *Normal* thread. The numerical priority ranking for the *Above Normal* thread is 9 whereas the priority ranking for the *Normal* thread is only 8, as seen in Figure 2.2.

| Priority Class Relative Priority | Realtime | High | Above Normal | Normal | Below Normal | Idle |
|---|---|---|---|---|---|---|
| Time Critical (+ SATURATION) | 31 | 15 | 15 | 15 | 15 | 15 |
| Highest (+2) | 26 | 15 | 12 | 10 | 8 | 6 |
| Above Normal (+1) | 25 | 14 | 11 | 9 | 7 | 5 |
| Normal (0) | 24 | 13 | 10 | 8 | 6 | 4 |
| Below Normal (-1) | 23 | 12 | 9 | 7 | 5 | 3 |
| Lowest (-2) | 22 | 11 | 8 | 6 | 4 | 2 |
| Idle (- SATURATION) | 16 | 1 | 1 | 1 | 1 | 1 |

Figure 2.2: Mapping of Windows® Kernel Priorities to the Windows® API as shown in [37]

Windows® also has the ability to boost thread priority levels [37]. This possibility exists because a single thread actually carries two priority levels, one inherited from its parent process and also a *current priority*. Windows® makes actual scheduling decisions upon a thread's *current priority*. Windows® may boost the priority of a thread based on its

*current priority*. However, threads with a *real-time* relative priority are exempt from this priority boosting.

Lastly, a thread does not always run to completion uninterrupted by another thread of equal or higher priority [37]. Additionally, threads have different *execution states*. There are eight different thread execution states: Ready, Deferred Ready, Standby, Running, Waiting, Transition, Terminated, and Initialized. A *ready* execution state signifies a thread that is awaiting execution. A *deferred ready* execution state signifies a thread that is not currently running, but is next to run on a specific processor. Russinovich et al. describe the deferred ready state as existing so that, "the kernel can minimize the amount of time the per-processor lock on the scheduling database is held." A *standby* execution state signifies a thread that is not currently running, but is next to run on a specific processor. A *running* execution state is when a thread is actually running. A *waiting* execution state describes a thread that is waiting for its next time quantum. Additionally, it is worth noting that after a thread finishes waiting, it either runs immediately or moves back to the *ready* execution state. A *transition* execution state signifies a thread whose kernel stack is, "paged out of memory," but the thread is still *ready* for execution. A *transition* execution state signifies a thread that is ready for execution but its "kernel stack is paged out of memory" [37]. Finally, during the creation of a thread, that thread is in an *initialized* execution state. Figure 2.3 shows the thread states as well as the transitions between those states, beginning with thread initialization, "Init (0)."

### *2.3.3 Dispatcher Database.*

The *dispatcher database* supervises threads awaiting execution and threads currently executing. The actual database is a collection of kernel data structures. Each processor maintains its own thread *ready* queue, summary, and *deferred ready* queue. Russinovich et al. state that each processor maintains the previously mentioned items to allow each processor to check which thread runs next without locking down the, "systemwide ready

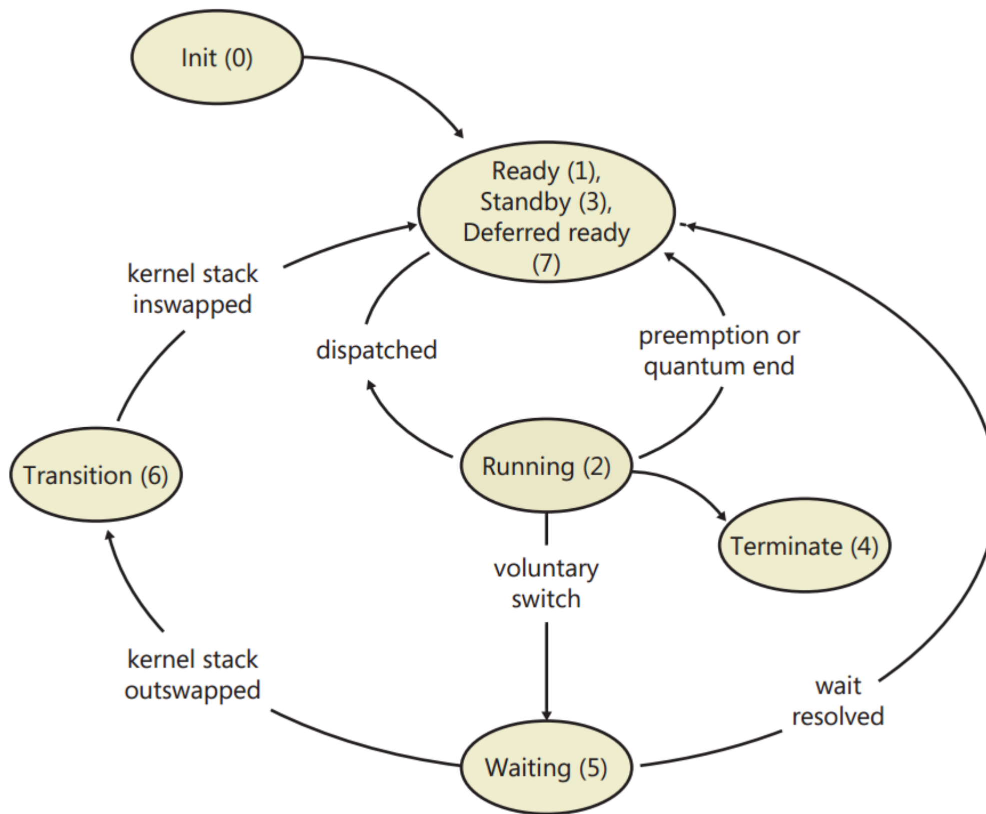


Figure 2.3: Simplified version of thread states and transitions as shown in [37]

queues" [37]. Windows® places each ready thread into the ready queue slot that is equivalent to their thread priority ranking, 0 through 31 (relative to priorities listed in Figure 2.2). Windows® then performs a single bit scan (bit by bit) on each processor's ready queue in order to locate the highest set bit (0 through 31), which will then be the next thread to execute on that processor. Russinovich et al. note that Windows® performing a single bit scan is the reason why some works refer to the Windows® scheduling algorithm as an O(1), or, "constant time, algorithm."

#### *2.3.4 Quantums.*

A quantum is the period of time Windows® allots to a thread for its execution [37]. After a thread uses its entire quantum, Windows® checks for runnable/ready threads with

the same priority ranking as the previously completed thread. If Windows® does not find an equally ranked thread that is ready to run, the previously completed thread runs for an additional quantum. Recall that if a thread with a higher priority becomes ready to run while a lower priority thread runs, Windows® preempts the lower priority thread regardless of whether the thread completes its quantum. A quantum on Windows® 7 equates to two clock intervals. The duration of the clock interval depends on the local hardware platform (e.g. 32 or 64-bit). In most cases, this clock interval is approximately 15 milliseconds. Additionally, new threads launched inherit a *quantum target* from their parent process. Once a thread runs for its quantum target, a context switch occurs and the next equal-priority thread runs for its quantum target, unless preempted by a higher priority thread.

#### *2.3.5 Thread Priority Boosts.*

Recall that a thread carries two priority levels, one inherited from its parent process and a *current priority* [37]. Under certain circumstances, Windows® will boost the thread's current priority. Russinovich et al. note that generally Windows® boosts a thread's current priority in order to reduce latency and to avoid, "inversion and starvation scenarios." The following non-inclusive list describes the different types of thread priority boosts.

- Unwait Boosts: In the event that a thread is in the *ready* state and that thread is waiting on specific and perhaps volatile information from a certain object, then Windows® applies a boost to the thread that is currently in the *ready* state. This occurs so that the thread has a chance to retrieve the volatile information, since the volatile information may change before the thread gets a chance to run.

- Post-Input/Output (I/O) Completion Boosts: If a thread requires I/O, but another thread was utilizing the I/O, then Windows® applies a boost to the thread that was waiting for the I/O so that it has a better chance to run next. Note that thread previously mentioned enters the *waiting* state until it has a chance to run.

- Awoken Graphical User Interface (GUI) Threads: If a thread has a window, such as a command prompt, the thread receives a temporary boost upon awakening in order to process such operations as text output to the window.

- CPU Starvation Boosts: Occasionally, a low-priority thread may be in the ready queue for several seconds without getting to run because higher-priority threads outrank the low-priority thread. Windows® implements a mechanism that scans the thread ready queue once per second, checking to see if any threads that have not run for four or more seconds exist. In the event that the search identifies a thread that has not run for four seconds, Windows® boosts that thread's priority to 15. The previous statement implies that the thread essentially equals or outranks all other threads in the ready queue, with the exception of threads with a *relative priority* of *realtime* (refer back to the thread priority mapping in Figure 2.2). Once the boosted thread runs and completes its quantum, it returns to the ready queue at its original priority level. However, this mechanism does not scan every single thread in the ready queue. The mechanism only scans 16 ready threads, but remembers the last thread scanned and continues from where it previously finished during the next scan.

#### ***2.3.6 Context Switching.***

Generally speaking, a context switch is like placing a bookmark inside of the last page read in a book to remember where one left off in order to start on another task [37]. Context switching involves saving the context of the previously running thread, placing the thread back into the ready queue, locating the next highest priority thread that is ready to run, and then loading that thread's context in order to run it [39]. An instruction pointer, kernel stack pointer, and a pointer to the address space in which the thread runs, are three items that a context switch saves and reloads [37]. The most common reasons for context switching are: a time slice elapsing, a higher priority thread becoming ready to run, and a running

thread that needs to wait [39]. Context switches occur under three different circumstances, as highlighted by Russinovich et al.

- Voluntary Switches: A voluntary switch occurs when the currently running thread gives up the processor that is in use because it is awaiting another resource that has not yet become available. Once the thread yields the processor, its state changes to *waiting* and the next highest priority *ready* thread executes its quantum.
- Thread Preemption: Thread preemption occurs when a higher priority thread becomes ready to run. If the thread that becomes ready to run is of higher priority than the thread currently running, then the lower-priority thread gives up the processor to the higher-priority thread. The lower priority thread does not go to the end of the ready queue, instead Windows® places it at the beginning of the ready queue. The time quantum elapses the next time the lower-priority thread runs.
- Quantum End: A quantum end is when a thread finishes its quantum and the next highest priority thread that is ready to run executes. Unlike thread preemption, the thread that finishes its quantum moves to the end of the ready queue instead of the beginning. The thread might run for another quantum if no higher-priority thread that becomes ready to run or an equal-priority thread that is in the ready queue.

#### 2.3.7 Other Notable Scheduling Concepts.

***Idle Threads.*** Each CPU has its own idle thread. Windows® launches a CPU's idle thread when no *runnable* thread exists. It is worth noting that the idle process possess all idle threads. This idle process appears in Windows® Task Manager as, "System Idle Process," which always has a process ID (PID) of zero.

***No Idle Processors Available.*** The *Core Parking engine* is a feature in Windows® that assists, at least in this case, with locating an idle processor. If the Core Parking engine cannot successfully locate an idle processor, then Windows® compares the ideal processor

number of the thread that is ready to run with the thread currently running on that thread's ideal processor. If the thread that is ready to run has a higher priority than the currently running thread, then the thread that is ready to run preempts the currently running thread and executes. Recall that in this case, Windows® places the preempted thread at the start (or beginning) of that processor's ready queue so that the preempted thread can finish its quantum as soon as the successor thread finishes its respective quantum. If the thread that is ready to run has an equal or lesser priority than the thread currently running on its ideal processor, then Windows® places the thread that is of equal or lesser priority at the end of that processor's ready queue in the appropriate ranking slot.

***Ideal and Last Processor.*** Every thread that executes stores the CPU numbers of its ideal processor, the processor on which the thread last ran, and the processor that the thread, "will be, or is already, running on" [37]. As long as the threads within a process accomplish an equal amount of work, the first runnable thread of that process has the numerical value for its ideal processor already set and each subsequent thread that belongs to that process has an ideal processor number of one greater than the prior runnable thread. This assignment technique continues while runnable threads exist belonging to that process and will obviously wrap back around to the first CPU once a thread meets the last (highest numbered) CPU.

## 2.4 Summary

This chapter provides a brief introduction to log files, log file processing, and log file analysis techniques. The use of logs not only every day life in different types of organizations, but also because logs are commonplace in court cases, evidences the fact that logs are important. Specifically, logs provide a way or a means to establish a timeline of events. This chapter also points out the challenges that come with logs, one being the difficult task of having to sift through the vast amount of log entries in an attempt to find meaningful information. Additionally, this chapter provides an overview of

Windows® thread scheduling, to include information on thread priority boosts and levels, quanta, and context switching.

# III. Methodology

## 3.1 Chapter Overview

This chapter details the research experiments of this thesis. It begins by introducing the research goals and approach taken to achieve these goals. Next, this chapter describes the system under test and measures of system failure or success in addition to the workload of the system under test. Successive sections define the performance metrics, parameters, and experimental factors. The chapter concludes by outlining the evaluation technique and the experimental design.

## 3.2 Goals

This research provides a statistical analysis of the effects of sensor multithreading for the purpose of logging local system events pertaining to specific file downloads. A suite of sensors, described later in this chapter, generate a timeline consisting of locally-produced and collected network and system-related events. This research observes and subsequently analyzes how differences in thread allocation amongst the sensors, as well as imposed system and network-traffic workloads, affect the accuracy of the resulting timeline. This research specifically focuses on two sensors within the suite of sensors. As such, an additional goal of this research is to find the optimal maximum thread count for the two aforementioned sensors for the values tested.

## 3.3 Approach

The experimental environment is an up to date Windows® 7 virtual machine (VM) with all of the updates applied that Microsoft identifies as *important*. The update setting within Windows® is set to *never check for updates*, once all important updates finish. This prevents Windows® from communicating with update servers and downloading updates automatically, which may affect test results. The research requires the installation of *Wget*

[40] and Python. The research also includes the pertinent Python scripts and directories (discussed later) in the experimental environment. Enabling Windows® auditing allows each test within the experiment to focus specifically on the creation of files within the download directory. Finally, the research includes a snapshot of this virtual machine, which the research reverts to prior to running each test during the experiment.

In order to achieve the aforementioned goal, this research makes use of the following utilities.

- The `NETSTAT` command, when used with the parameters `a`, `n`, and `p` (with the option "`tcp`"), displays all active Transmission Control Protocol (TCP) connections (without attempting to resolve host names) and ports that the computer is listening on [41]. Resolving host names while using the `NETSTAT` command slows down the entire process as experienced in initial investigative testing.
- The `tasklist` command displays all active processes along with their PIDs running on the local system [42].
- The enabling of native Windows® auditing for the test download directory will not only produce entries for the log timeline, but also verifies that the files within each experiment actually download.
- Lastly, `tcpdump` [43] captures packet data from the network. For the purposes of this experiment, `tcpdump` only captures the Hypertext Transfer Protocol (HTTP) GET requests sent from the local machine during the tests. Figure A.1 in the Appendix provides a breakdown of the different parts of the `tcpdump` command.

These four items (`NETSTAT`, `tasklist`, Windows® audit entries, and `tcpdump`) log their respective data and ultimately combine into one log file, which represents a chronological timeline of when the logged events occur on the local system for that particular test.

A series of Python scripts automate the tests within the experiment. The Python scripts collect the aforementioned data from the four sensors on the Virtual Machine (VM) as well as specifying the maximum number of threads to assign the sensors. Additionally, the scripts execute a file-download scenario in order to check the accuracy of the sensors within each test. The scripted file-download scenario consists of downloading five unique files from five separate file servers 50 times each for a total of 250 files downloaded per test. This research utilizes *Wget* to download the files, creating a separate *Wget* executable for each of the five files in order to differentiate and determine ordering accuracy after each test completes and the logging suite outputs the final consolidated timeline. The experiment only increases the maximum number of `NETSTAT` and `tasklist` processes, as preliminary tests show that only one `tcpdump` process is necessary and that Windows® logs the specified actions in the predetermined download directory regardless of the tests. This research mainly focuses on the multithreading of the `NETSTAT` and `tasklist` sensors and thus did not increase the number of tcpdump threads. Note that one tcpdump instance captures nearly 100% of the HTTP GET requests during every test even with only one instance running. The native `NETSTAT` and `tasklist` commands on Windows® do not include timestamps. The Python scripts also append microsecond resolution timestamps to the beginning of every captured event.

The experiment utilizes three different system and network-traffic workloads (*low*, *medium*, and *high*). The aforementioned workloads simulate a user that is actively using the machine while the tests execute, thereby potentially affecting the results. Three different workloads necessitate three rounds of testing; Section 3.9 explains the workloads in greater detail. A single test within the experiment executes as follows.

1. Set the workload (beginning with *low*)

2. Set the maximum process allocation for `NETSTAT` and `tasklist`

3. Start capturing `tcpdump` data followed by `NETSTAT` and `tasklist` data

4. Start the scripted file-download scenario,

5. Once the scripted file-download scenario completes, stop capturing `tcpdump`, `NETSTAT`, and `tasklist` data

6. Pull the pertinent Windows® audit entries, in this case, those events with Event ID number 4663 ("an attempt was made to access an object") from the Windows® audit log. One event with ID number 4663 accompanies every file downloaded in the scenario, for a total of 250 events

7. Python scripts consolidate the captured data into one chronologically-sorted log file, which results in the timeline for this particular test

8. Save the resulting files to an external source, revert to snapshot, and repeat the test accordingly

Each test permutation, a maximum of one `NETSTAT` and one `tasklist` process constitutes one permutation, repeats 50 times in order to effectively calculate the mean accuracies for each of the four sensors within the experiment. The resulting data also provides the mean accuracy of the test as a whole, the mean of all sensors combined. Then, modifying the permutation by, for instance, increasing `NETSTAT` to have a maximum of five processes running and again conducting 50 tests. Repeating this process once more by increasing the maximum number of processes for `NETSTAT` to ten provides a framework from which to work. Then, the mean accuracies identify a trend for the data that `NETSTAT` captures after completing three series of tests for the three permutations. This trend may be a downward, upward, or level slope, or a combination of two. Conducting tests of the permutations between these results will enable the identification of `NETSTAT` set to a maximum of five processes as actually being the most accurate.

Lastly, comparing the generated timelines from each test to a baseline timeline facilitates the ability to calculate the mean accuracies of each test. One thousand entries or

"events" derived from the scripted file-download scenario constitutes the baseline timeline. Capturing one log entry per sensor per file downloaded effectively tests each sensor's accuracy. Two hundred and fifty of these 1,000 events are `tasklist` entries, 250 are `NETSTAT` entries, 250 are `tcpdump` entries, and the final 250 entries are Windows® audit log entries. Figure 3.1 shows examples of what these sensors' entries look like in the resulting timeline from each test.

**Tasklist Entry** = 00:03:53.683586 wget1.exe 3652 Console 1 4,368 K
**Netstat Entry** = 00:03:53.714786 TCP 192.168.46.128:52073 10.1.0.46:80 ESTABLISHED
**tcpdump Entry** = 00:03:53.933186 IP 192.168.46.128.52073 > 10.1.0.46.80: Flags [P.], ack 491788371, win 64240, length 131
**Windows Audit Log Entry** = 00:03:53.995587 Event ID 4663 ('An attempt was made to access an object'): 'C:\Users\dgallagher\Desktop\Components\wget\Downloads\2\356px-Seal_of_the_US_Air_Force.png'

Figure 3.1: Example of sensor entries

### 3.4 System Boundaries

The system under test (SUT), shown in Figure 3.2, includes:

1. a series of Python scripts that make up a component under test (CUT) that

   - execute scripted file downloads which generate network traffic and security events that make up the resulting timeline,
   - automate the event capturing process by executing `tcpdump`, as well as specifying the maximum number of processes for the `NETSTAT` and `tasklist` sensors and subsequently executing them, and
   - extract the pertinent Windows® audit log file ("`Security.evtx`") entries; and

2. a series of Python scripts that consolidate the logs produced by the four sensors into the resulting timeline, which is one text file per test.

The input of the SUT is a scripted file-download scenario that downloads a total of 250 files per test.

The output of the SUT is the consolidated timeline, from which the research derives the experiment metrics. Additionally, the possibility exists for the appearance of events in the logs that are not a product of the experiment procedure. This research attempts to minimalize this from occurring by turning off automatic updates of software installed on the system, to include Windows® itself. Regardless of this precaution, the research takes this into consideration when filtering the resulting timelines such that the only entries kept are those that apply to the experiment procedure.

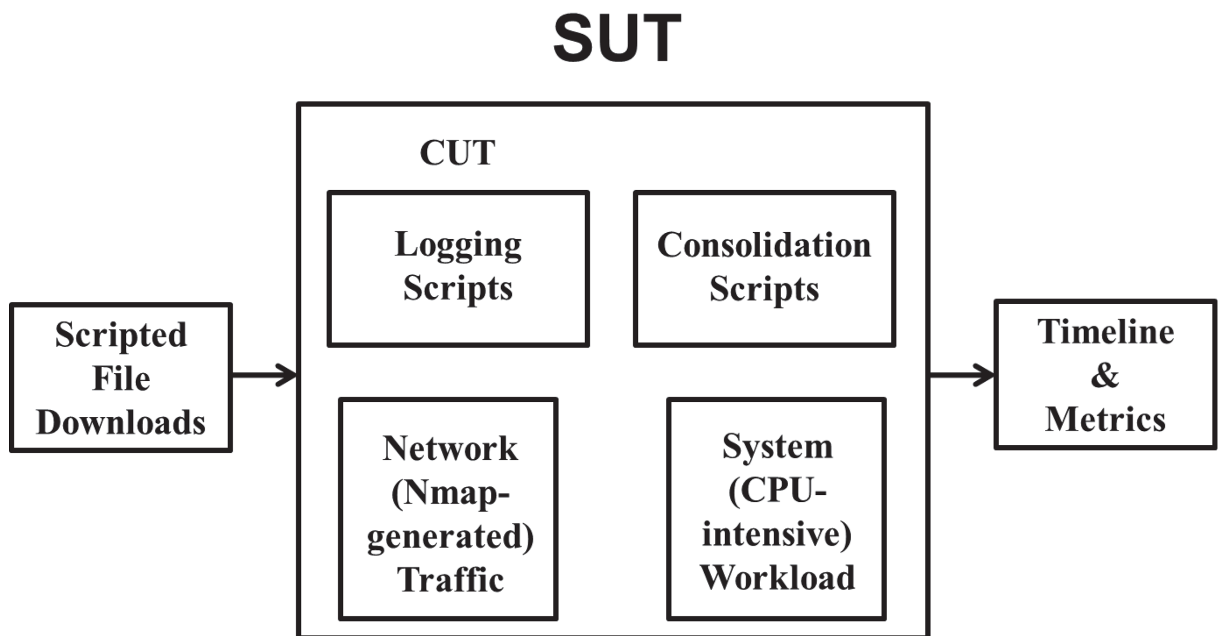


Figure 3.2: System Under Test (SUT)

### 3.5 System Services

The system output gives the end-user the ability to determine the optimal number of `NETSTAT` and `tasklist` threads to use within the procedure by comparing the resulting

timeline to a baseline timeline under the three different network and system workloads. Three possible outcomes of the SUT exist:

- a successful outcome
  - system outputs a timeline that captures every event in correct chronological order; or
- failed outcomes
  1. system outputs a timeline that captures every event, but they are not in the correct chronological order, or
  2. system outputs a timeline that *does not* include every event in the baseline timeline.

### 3.6 Workload

The scripted file-download scenario includes network traffic and writing the downloaded files to disk. In addition to the scripted file-download scenario, the actual logging of the `NETSTAT`, `tasklist`, and `tcpdump` entries, and thus launching of those processes generate another portion of the workload. The system logs `NETSTAT` and `tasklist` entries by collecting the output of each query and saves them to individual text files (one text file per query). Therefore, during one test, hundreds and sometimes thousands of individual text files written to disk. The workload tasks hold true across the low, medium, and high workload factors. Section 3.9 details the workload factors, which also include additional network and CPU-intensive tasks.

### 3.7 Performance Metrics

The accuracy of the produced timelines from the experiments is the primary metric. Specifically, one metric is *if* the timelines captured the events as expected. Each test produces a series of captured events. The research pre-determines that the number of events

per sensor is a maximum of 250 events, for a total of 1,000 events (250 * 4 sensors) per test. The research divides the number of captured events by each sensor by 250, which then produces the accuracy (percentage of events captured) for that sensor for that workload level. The research produces average performance accuracies (percentages) for each sensor, in addition to an average of the four sensors combined, for a total of five metrics (`NETSTAT`, `tasklist`, `tcpdump`, Windows® logs, and total average) per test series. Accordingly, the results also contain standard deviations and confidence intervals for each of the five metrics. Another metric is the ordering of the events in the timelines in comparison to their actual chronological occurrences.[1] Section 3.10 provides details on the particular ordering within each test's resulting timeline.

### 3.8 System Parameters

- The system hardware:
  - CPU: Intel® Core™ i7 CPU, M640 @ 2.80GHz
  - Random access memory (RAM): 4 gigabytes (GB) DDR3-1333MHz
  - Cache: 4 megabytes (MB) L3 cache
  - Hard disk drive (HDD): 60GB (of 500GB) Seagate Momentus 7200RPM, 16MB cache, 3.0Gb/s
- OS: As this experiment executes multiple threads, the OS, Windows® 7, is a parameter. The OS schedules individual tasks and thus the scripts cannot control the scheduling of individual tasks. As such, the host OS scheduler schedules the scripts and subsequent spawned tasks from within the scripts. The experiments that follow utilize two processors with two cores per processor (four processor cores total) for the experiment system.

[1] For example, the four events A, B, C, and D chronologically occurred in this order, but may *appear* in an incorrect order in the resulting timeline as A, C, B, and D.

### 3.9 Factors

As previously highlighted, this research imposes three different system and network-traffic workloads in order to simulate user activity. Simulating user activity while the tests execute aims to potentially affect the results of each test. The following list describes the load factors and corresponding levels used in this research.

***Low Load.*** The processes running within the test itself as well as the Windows®-generated processes running in the background on the OS snapshot (as previously outlined in Section 3.3).

***Medium load.*** In addition to the *low* load processes, custom CPU-intensive and network traffic-intensive Python scripts run during the tests. The CPU-intensive script continuously computes the factorial of a large integer, while the network traffic-intensive script launches an *intense* `Nmap` [44] scan which runs throughout the duration of the test. The `Nmap` scan runs with the timing (`-T`) option set to `5`, being the most aggressive (and thus noisiest) timing. In order to stress the system's sensors, three CPU-intensive and three network traffic-intensive scripts run simultaneously to make up the *medium* load.

***High load.*** The *high* load factor executes twice as many CPU-intensive and network traffic-intensive scripts as the *medium* load factor.

### 3.10 Evaluation Technique

A comparison of the baseline timeline with the resulting timelines of each test serves to evaluate the system. Prior knowledge of the expected outcome enables the ability to check within the resulting timelines of each test. Likewise, the resulting timelines of each test also serve to determine the ordering accuracy. One custom Python script counts the number of entries found, while another custom Python script cycles through each of the resulting timelines determining the ordering accuracy. The script determines the ordering accuracy by first splitting the events within the resulting timeline into five separate text files, one for each of the five files in the scripted download scenario, while keeping their

chronological order intact. Because each unique file within the scripted scenario downloads from its own unique file server using a unique *Wget* executable, each of the five text files contains only those entries that pertain to that unique file. Figure 3.3 provides an example of a potential outcome of this process. In Figure 3.3, the letters A, B, C, and D signify the captured events of the four sensors. In this case, A is the `tasklist` entry, B is the `NETSTAT` entry, C is the `tcpdump` entry, and D is the Windows® audit log entry, which is the desired order of appearance for the events in each test. The first subscript number (one through five) signifies the server. Finally, the second subscript number, 50, signifies the file counter for each particular server. In the example given, all of the servers have just completed their 50th file download.

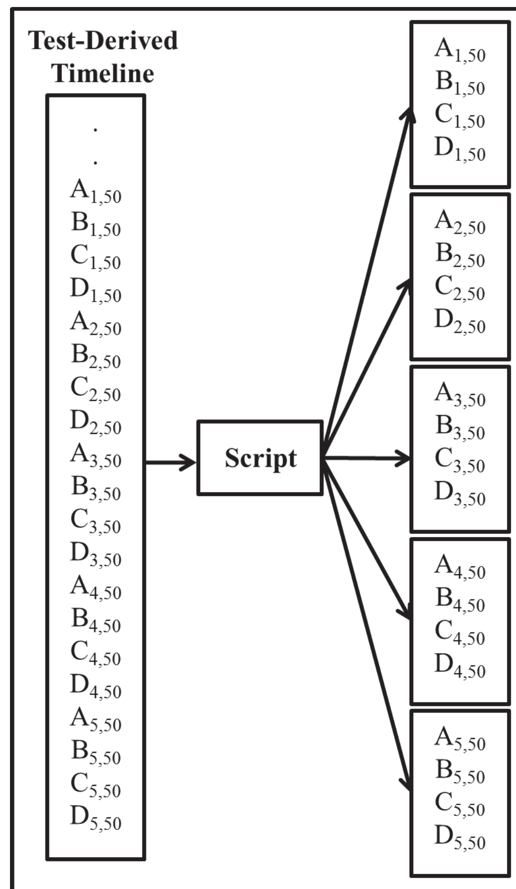


Figure 3.3: Separating linked events into separate text files

Counting the chronological occurrences of the events in each of the five text files determines the ordering accuracy. Specifically, because four sensors are present within each test and they *would* appear in a predictable order, an *ideal* timeline *should* consist of 250 sets of the four sensors events.[2] However, a total of 1,000 entries for every single test is not a feasible expectation as the possibility exists for missing events within the resulting timeline. Instead:

- counting each perfect set of four events ($A_1$, $B_1$, $C_1$, $D_1$), in this case called "quartets", followed by
- sets of three in-order events ($A_1$, $B_1$, $C_1$ or $B_1$, $C_1$, $D_1$), called "triplets", followed by
- sets of two in-order events ($A_1$, $B_1$ or $B_1$, $C_1$ or $C_1$, $D_1$), called "doubles", and finally
- the leftover events called "singles" ($A_1$ or $B_1$ or $C_1$ or $D_1$), serves as a way to determine ordering accuracy.

Therefore, each resulting timeline will produce counts of quartets, triplets, doubles, and finally singles, which when combined ($quartets * 4 + triplets * 3 + doubles * 2 + singles$) totals the number of events found in that particular test. Each event within the resulting timeline is in either a quartet, triplet, or double set or is a single and is only counted once. Figure 3.4 provides a simplified visual depiction of this process.

### 3.11 Experimental Design

This experiment adopts a full-factorial design for all tests. The OS, Windows® 7, conducts audit logging. In the case of `tcpdump`, this procedure only allocates one process, which is running continuously throughout each test. The procedure increases the maximum number of `NETSTAT` processes for each test series (after completing 50 individual tests) by

[2] e.g., $A_1$, $B_1$, $C_1$, $D_1$, $A_2$, $B_2$, $C_2$, $D_2$,..., $A_{50}$, $B_{50}$, $C_{50}$, $D_{50}$ for each of the five files in the scripted download scenario

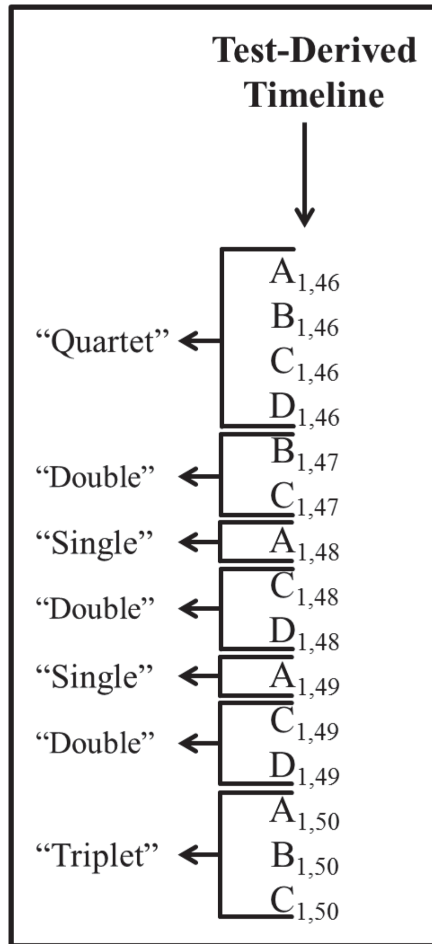


Figure 3.4: Example of determining ordering accuracy

one from one up to ten, while keeping the maximum number of `tasklist` processes at one. For each increase (by one) in the maximum process count for `NETSTAT`, the execution of the experiment conducts the tests under low, medium, and high load factors as previously defined in Section 3.9. The procedure also completes this same process for `tasklist`. As such, the experiment requires 30 series of tests for `NETSTAT` and 27 series of tests for `tasklist`. In this case, a test series is for example, 50 individual tests with a maximum of one process for both `NETSTAT` and `tasklist` under a low load. Based on the previous example given, this is why the experiment only requires 27 series of tests for `tasklist`;

the resulting data is the same under low, medium, and high loads for both sensors when both are set to a maximum of one process. For each of the 57 series within the experiment, the research conducts 50 tests (for a total of $57 * 50 = 2{,}850$ individual tests) in order to produce test mean, variance, standard deviation, and confidence interval metrics. However, for uniformity and ease of reporting results, the research states it conducts 60 series of tests, with the caveat that the results report three series of tests twice.

### 3.12 Summary

This chapter summarizes the methodology of the experiments conducted during this research by first defining the goals and the approach of the research. It defines not only the system under test, but also the workload of the system as well as the measures of system failure or success. Sections 3.7 through 3.9 define the performance metrics, parameters, and experimental factors, respectively. Finally, the last two sections explain the evaluation technique and the experimental design of this research.

# IV. Analysis and Results

## 4.1 Chapter Overview

This chapter details the results of the experiment. Section 4.2 provides a statistical analysis of the experimental results. Furthermore, successive subsections present the results of the experiment beginning with tests that increase the maximum number of `NETSTAT` processes (Subsection 4.2.1) and likewise for `tasklist` processes (Subsection 4.2.2). Each of these two subsections has three additional subsections which provide specific experimental data of `NETSTAT` and `tasklist` under a low, medium, and high load.

## 4.2 Experimental Results

This section provides the results of the experiments. Additionally, the figures and tables that follow present a statistical analysis of the results. This section presents the results in two subsections in order to provide specific details pertaining to `NETSTAT` (Subsection 4.2.1) and `tasklist` (Subsection 4.2.2) data under low, medium, and high loads.

### *4.2.1 `NETSTAT` Sensor.*

This subsection presents three additional subsections. Each subsection presents the results of testing the `NETSTAT` sensor under the three different load factors. Beginning with the low load results, this subsection continues by presenting the results of the medium and high load tests.

#### *4.2.1.1 `NETSTAT` Low Load Results.*

Table 4.1 provides a summary of the experiment when increasing the maximum number of `NETSTAT` processes under a low load. Appendix B contains complete performance summaries (Tables B.1 through B.4) for these tests under a low load which include the data for all sensors within the experiment. As previously mentioned, the

research presents 60 series of tests. Table 4.1 shows the NETSTAT data for 10 of these 60 series of tests.

In Table 4.1, the results show that in order to capture the most NETSTAT-related events under a low load while increasing the maximum number of NETSTAT processes, a maximum of one NETSTAT process suffices. Granted, the data also shows that none of the series is actually able to capture 100% of the NETSTAT-related events every time, otherwise the results would present a series in the table with an average of 100%. However, several individual tests from each series of tests actually capture 100% of the NETSTAT-related events.

Table 4.1: Performance summary of NETSTAT capture accuracy under a low load

| Max NETSTAT Processes | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|
| 1 | 99.83529 | 0.40141 | [99.72513 , 99.94546] |
| 2 | 99.70981 | 0.58284 | [99.54984 , 99.86976] |
| 3 | 99.49804 | 0.71484 | [99.30185 , 99.69423] |
| 4 | 98.88627 | 1.22720 | [98.54947 , 99.22308] |
| 5 | 98.23529 | 1.41574 | [97.84674 , 98.62384] |
| 6 | 98.76078 | 1.33448 | [98.39454 , 99.12703] |
| 7 | 98.00000 | 1.59599 | [97.56198 , 98.43802] |
| 8 | 97.98431 | 1.40618 | [97.59839 , 98.37024] |
| 9 | 97.89804 | 2.22832 | [97.28648 , 98.50960] |
| 10 | 96.73725 | 2.33015 | [96.09775 , 97.37676] |

Figure 4.1 provides a visual depiction of the mean percentages of NETSTAT-related events captured under low load while increasing maximum number of NETSTAT processes by one from one up to ten, but keeping tasklist at a maximum of one process. The figure shows that increasing the maximum number of NETSTAT processes has an overall negative effect on the amount of captured tasklist-related events up to the fifth series (5 along the X-axis). As the maximum number of NETSTAT processes increases and the maximum number of tasklist processes remains constant, the amount of data that the tasklist sensor captures declines. One significant discrepancy in the results, as shown in Figure 4.1, is the increase in events captured by the tasklist sensor, beginning with the sixth test series. One theory for the cause of the aforementioned discrepancy is that because hundreds and perhaps thousands of NETSTAT processes running at this point, several NETSTAT-related threads are waiting on the same resources in order to complete their NETSTAT query, thus taking longer for each NETSTAT process to complete. Thus, several NETSTAT-related threads perform voluntary switches and enter the *waiting* state. Once the resource becomes available, the NETSTAT thread then moves to the ready queue and awaits its next quantum in order to finish. However, enough NETSTAT processes still finish in order to maintain the 96.74% to 99.84% average. Essentially, several NETSTAT-related threads perform voluntary switches opens up CPU time for tasklist threads to finish their queries. Granted, confirmation of this theory is not possible due to lack of OS scheduling visibility and because this research did not primarily focus on how thread scheduling occurs during each test series.

Accordingly, as the amount of data captured by the tasklist sensor declines, the overall amount of captured events for each series of tests also declines, which Figure 4.1 also depicts. The amount of entries the tcpdump and Windows® logs sensors are capturing is unaffected by the increase in NETSTAT processes. Note that neither the tcpdump sensor data nor the Windows® log sensor data is present in Figure 4.1 as the figure becomes

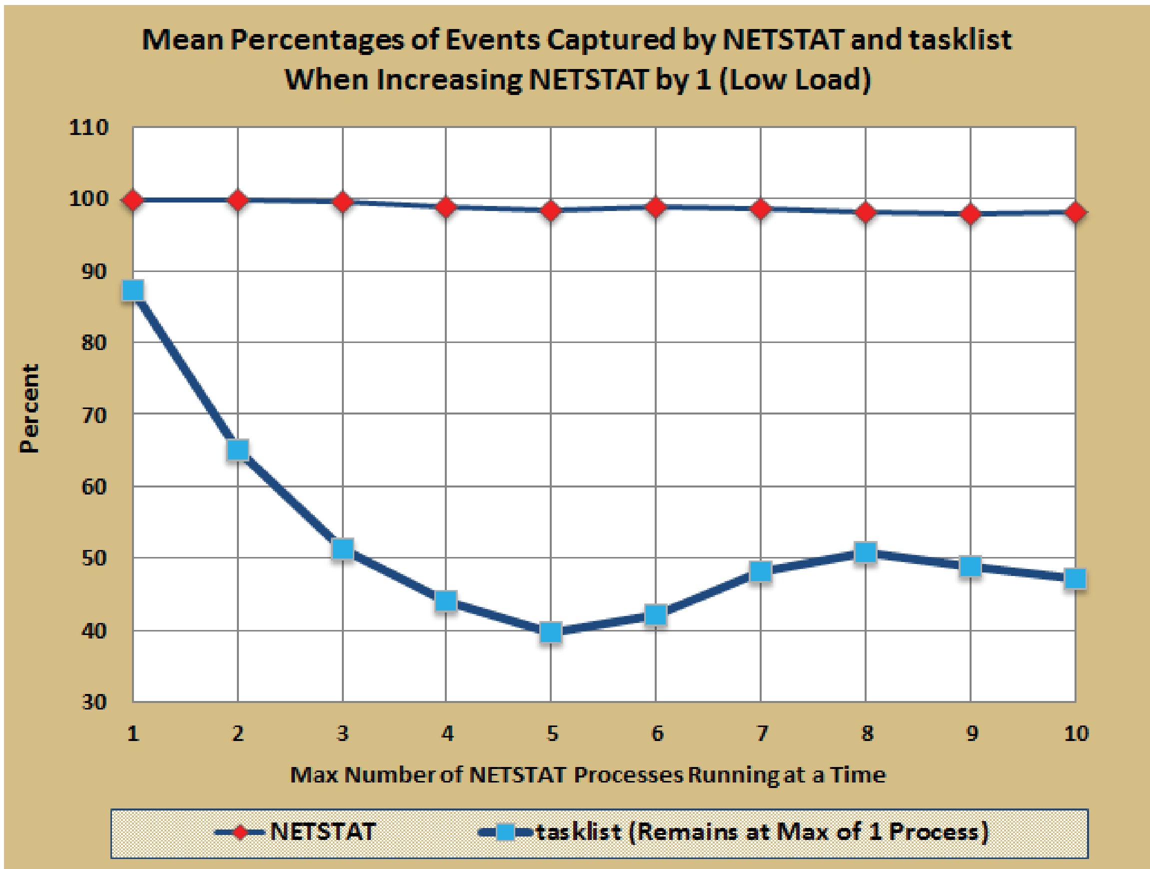


Figure 4.1: Mean percentages of events captured while increasing maximum number of `NETSTAT` processes by 1 (low load)

cluttered. The `tcpdump` and Windows® log sensors maintained a near-100% average throughout all series of tests shown in Table 4.1. Later in this subsection, the `tcpdump` sensor maintains an average accuracy of 99% across all three workloads. This leads to the belief that neither the increase in `NETSTAT` (and eventually `tasklist`) processes nor the load scripts introduced later in the experiment actually affect the ability for `tcpdump` to capture `tcpdump`-related events.

Figure 4.2 provides a visual depiction of the ordering accuracy when increasing the maximum number of `NETSTAT` processes under a low load. Essentially, a higher amount of quartets and triplets than double or singles in the results is ideal. Because events happen in

a predictable chronological sequence (tasklist event, NETSTAT event, tcpdump event, and finally a Windows® log event), the possibility exists to count their order in sets of four, followed by three, then two, and lastly one, as previously explained in Section 3.10. In the case of Figure 4.2, the number of quartets peaks right at the beginning, when NETSTAT only has a maximum of one process (1 along the X-axis) and then progressively declines. As the number of quartets declines, then the number of triplets increases from the first series to the sixth series and then it too declines. After the sixth series, as the number of quartets and triplets decline, the number of doubles and singles increases. In this instance, the highest concentration of quartets and triplets is when NETSTAT only has a maximum of one process, making up 75% of all events found on average across the first test series. Because the tasklist sensor's events are the first events in the quartet chains and its mean accuracy declines from the beginning, the number of quartets found also declines (as seen in Figure 4.2). Accordingly, the number of triplets increases as the number of quartets declines.

#### *4.2.1.2 NETSTAT Medium Load Results.*

Table 4.2 provides a summary of the experiment when increasing the maximum number of NETSTAT processes under a medium load. The Appendix contains complete performance summaries (Tables B.5 through B.8) for these tests under a medium load which include the data for all sensors within the experiment. As previously mentioned, the research presents 60 series of tests. Table 4.2 shows the NETSTAT data for 10 of these 60 series.

In Table 4.2, the results show that in order to capture the most NETSTAT-related events under a medium load while increasing the maximum number of NETSTAT processes, a maximum of two NETSTAT processes is the optimal setting. The results display an increase of 8.6% in the average percentage of events captured after the maximum number of NETSTAT processes increases from one to two, but then the results show a significant

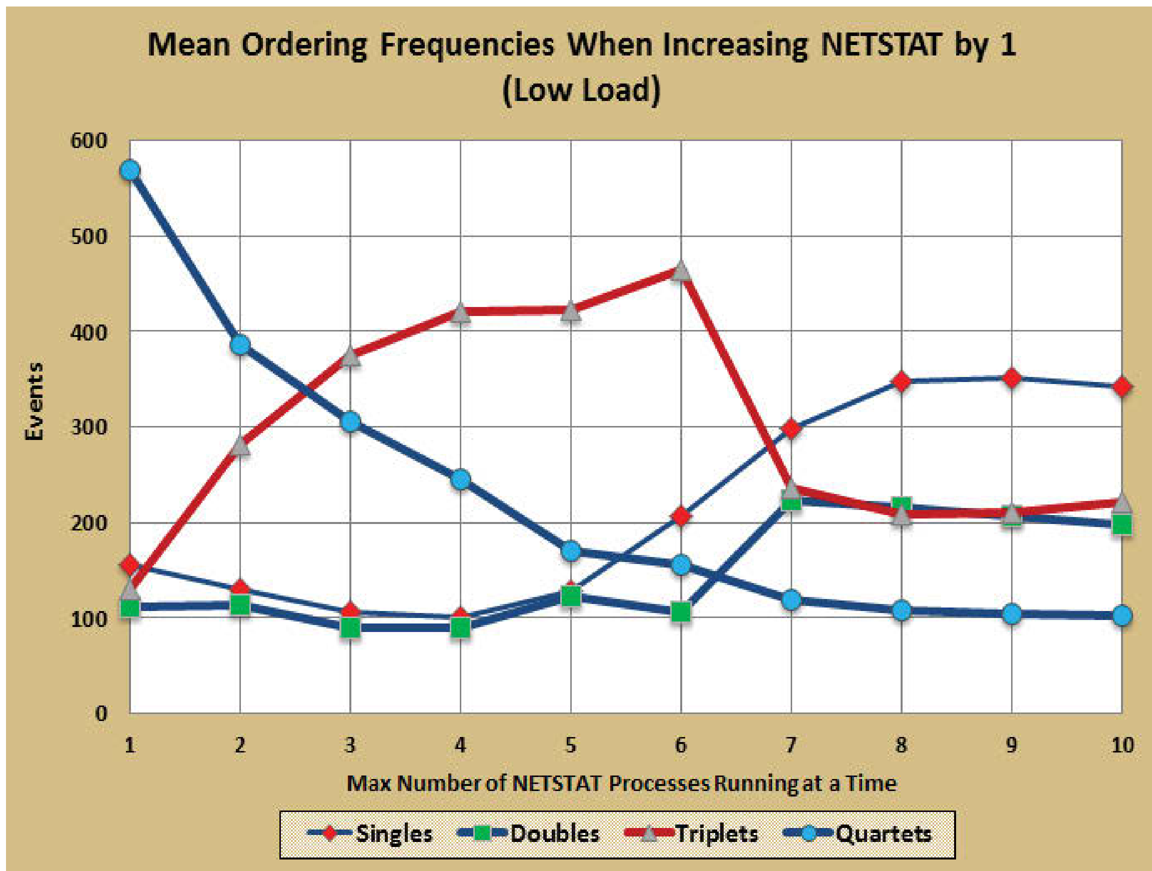

Figure 4.2: Mean ordering frequencies while increasing maximum number of NETSTAT processes by 1 (low load)

decrease of ~25% between three and four and then an additional decrease of 24% between four and five and then finally the percentages level out. The percentages levelling out, starting with the fifth series is expected behavior as launching more NETSTAT processes means that more NETSTAT-related threads are contending for the same resources, as previously described in Subsection 4.2.1.1. Of course, the last point is in addition to the medium load scripts crippling the average percentages of both NETSTAT and tasklist sensors. Enough NETSTAT processes still run in order to discover 30% of the NETSTAT-related events beginning with the fifth series (hence the plateau in the data shown in Table 4.2). The data shows that none of the series are actually able to capture 100% of the

NETSTAT-related events every time. Note the absence of a series in the table with an average of 100%. However, unlike the NETSTAT results under a low load in Section 4.2.1.1, no individual test from any series of tests under a medium load actually captures 100% of the NETSTAT-related events.

Table 4.2: Performance summary of NETSTAT capture accuracy under a medium load

| Max NETSTAT Processes | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|
| 1 | 76.18039 | 7.13361 | [74.22257 , 78.13821] |
| 2 | 84.73725 | 6.48463 | [82.95755 , 86.51696] |
| 3 | 78.55686 | 10.42369 | [75.69608 , 81.41764] |
| 4 | 53.60784 | 19.94512 | [48.13391 , 59.08178] |
| 5 | 29.48235 | 13.31088 | [25.82918 , 33.13552] |
| 6 | 26.79216 | 9.68545 | [24.13399 , 29.45033] |
| 7 | 27.32549 | 9.92080 | [24.60273 , 30.04825] |
| 8 | 27.25490 | 7.35609 | [25.23602 , 29.27378] |
| 9 | 30.72157 | 10.41599 | [27.86290 , 33.58024] |
| 10 | 28.71373 | 8.33741 | [26.42552 , 31.00193] |

Figure 4.3 provides a visual depiction of the mean percentages of NETSTAT-related events captured under low load while increasing maximum number of NETSTAT processes by one from one up to ten. Also shown in the figure is the accuracy of the tasklist sensor, which remains at a maximum of one tasklist process throughout all series of tests. The figure shows that increasing the maximum number of NETSTAT processes after the second series of tests (2 along the X-axis) actually appears to have a negative effect on the number

of captured NETSTAT-related events. Accordingly, as the amount of data captured declines, the overall amount of captured events for each series of tests also declines, which Figure 4.3 also shows. The implementation of the medium load scripts also has a negative effect on the amount of tasklist-related events the tasklist sensor captures.

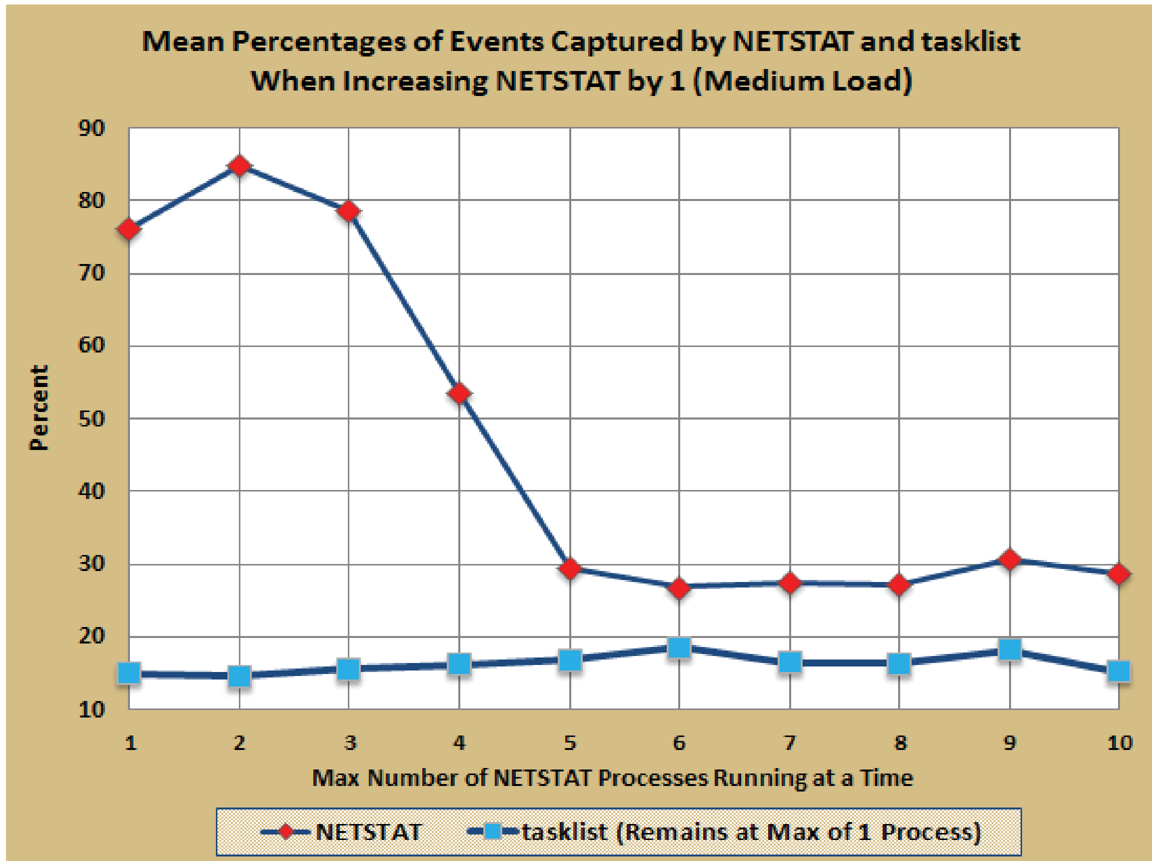


Figure 4.3: Mean percentages of events captured while increasing maximum number of NETSTAT processes by 1 (medium load)

Not every file in the scripted file-download scenario actually downloads due to the implementation of the medium load scripts (and later the high load scripts), specifically referring to the intensive Nmap scans. As a result, the Windows® log sensor still captures 100% of the events as expected, but the other sensors only capture the number of events

less than or equal to what the Windows® log sensor captures. For example, if the Windows® log sensor only captures 243 events, then the other 3 sensors can only capture a maximum of 243 events. The aforementioned issue is not an issue under a low load. However, the aforementioned issue is an issue under medium and high loads. The statistics present in this chapter take this issue into consideration. Note that neither the `tcpdump` sensor data nor the Windows® log sensor data is present in Figure 4.3 as the figure becomes cluttered.

Figure 4.4 provides a visual depiction of the ordering accuracy when increasing the maximum number of `NETSTAT` processes under a medium load. Given this round of tests is under a medium load, the number of quartets is expectedly lower than the tests under a low load. That said, the number of events a part of a triplet is significantly higher than previously seen under a low load. The previous statement makes sense given the number of quartets is lower. Interestingly enough, the trend of triplets actually seems to follow the same trend as the mean `NETSTAT` averages as plotted previously in Figure 4.3. As triplets decline, the number of doubles increases, which makes sense as the ordering of the events should be a `tasklist` entry, a `NETSTAT` entry, a `tcpdump` entry, and finally a Windows® log entry. So, the declination of the `NETSTAT` sensor's accuracy removes the possibility of not only quartets from occurring, but also triplets from occurring. In this instance, the highest concentration of quartets and triplets is when `NETSTAT` only has a maximum of two processes, making up ~76.6% of all events found for that test series, which makes sense given the optimal setting for `NETSTAT` under a medium load (as previously shown in Table 4.2) is a maximum of two processes.

#### *4.2.1.3 NETSTAT High Load Results.*

Table 4.3 provides a summary of the experiment when increasing the maximum number of `NETSTAT` processes under a high load. The Appendix contains complete performance summaries (Tables B.9 through B.12) for these tests under a high load which

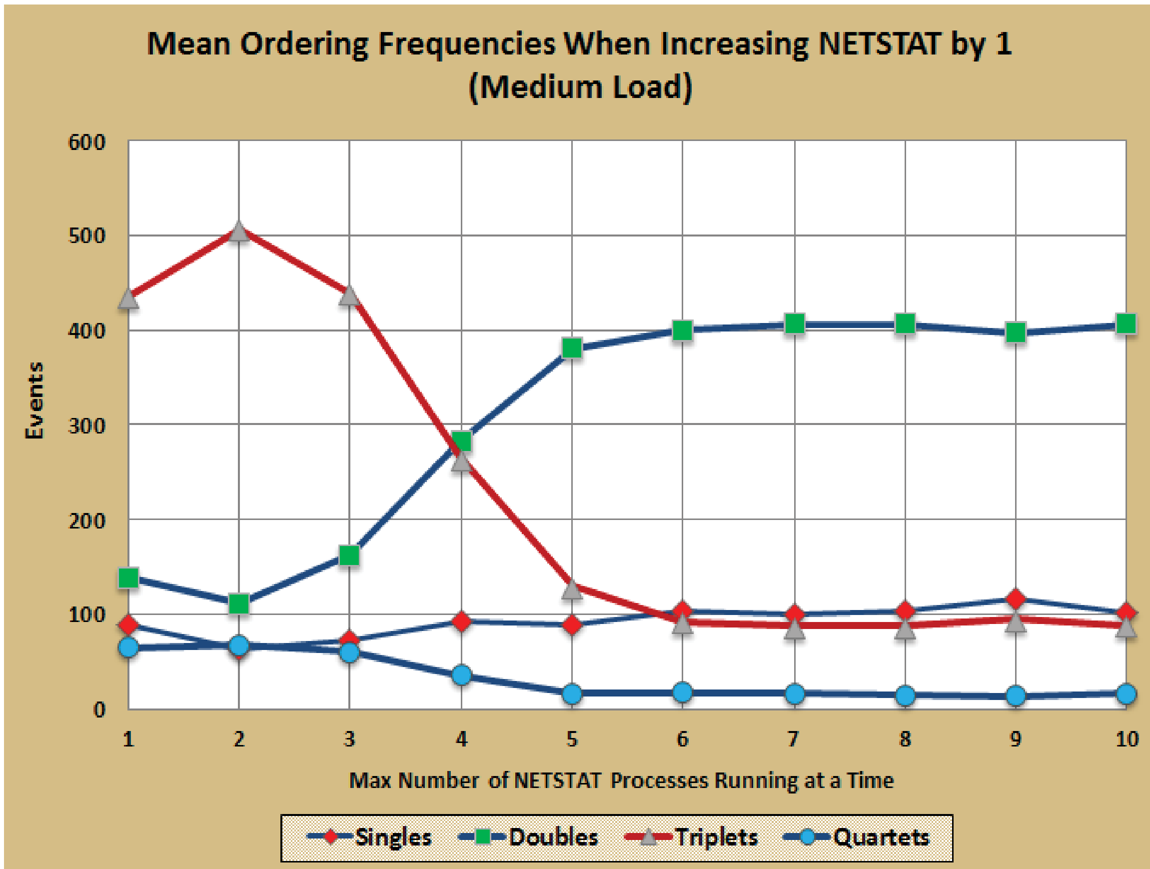


Figure 4.4: Mean ordering frequencies while increasing maximum number of NETSTAT processes by 1 (medium load)

include the data for all sensors within the experiment. As previously mentioned, the research presents 60 series of tests. Table 4.3 shows the NETSTAT data for 10 of these 60 series of tests.

In Table 4.3, the results show that in order to capture the most NETSTAT-related events under a high load while increasing the maximum number of NETSTAT processes, a maximum of either one or two NETSTAT processes suffices. Either one or two NETSTAT processes are acceptable given the confidence intervals for both series of tests overlap, with the lower bound of the second series of tests being 1.73% less and the upper bound being 0.73% more than that of the first series of tests. The confidence interval of the second series

of tests actually contains the entire confidence interval of the first series of tests. Like with the NETSTAT data under a medium load in Section 4.2.1.2, the results also show a sharp decrease of 42% in average percentage of NETSTAT events captured under a high load. The decrease occurs from the second to the fourth series of tests under a high load, unlike from the third to the fifth series of tests under a medium load. The first sharp decrease is 26% followed by another sharp decrease of ~16%. The NETSTAT and tasklist sensors capture 20% less events under a high load than they did under a medium load. Likewise, the same thought process behind the steep drop in mean accuracy under a medium load holds true under a high load. Similar to the low and medium load sections discussing NETSTAT data, the high load data shows that none of the series are actually able to capture 100% of the NETSTAT-related events every time. Like under a medium load, no individual test from any series of tests under a high load actually captures 100% of the NETSTAT-related events.

Figure 4.5 provides a visual depiction of the mean percentages of NETSTAT-related events captured under low load while increasing maximum number of NETSTAT processes by one from one up to ten, but keeping tasklist at a maximum of one process. Similar to the tests under a medium load, not every file in the scripted file-download scenario actually downloads under a high load. As a result, the Windows® log sensor still captures 100% of the events as expected, but the other sensors only capture the number of events less than or equal to what the Windows® log sensor captures. Note that neither the tcpdump sensor data nor the Windows® log sensor data is present in Figure 4.5 as the figure becomes cluttered.

Figure 4.6 provides a visual depiction of the ordering accuracy when increasing the maximum number of NETSTAT processes under a high load. Given this round of tests is under a high load, the number of quartets is expectedly lower than the tests under both low and medium loads. That said, the number of triplets is higher than previously seen under a low load, but less than previously seen under a medium load. The trend of triplets actually

Table 4.3: Performance summary of NETSTAT capture accuracy under a high load

| Max NETSTAT Processes | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|
| 1 | 60.90980 | 15.68634 | [56.60469 , 65.21492] |
| 2 | 60.40784 | 20.17103 | [54.87191 , 65.94378] |
| 3 | 34.32157 | 19.31781 | [29.01980 , 39.62334] |
| 4 | 18.07059 | 13.21713 | [14.44315 , 21.69803] |
| 5 | 12.54902 | 5.85585 | [10.94188 , 14.15616] |
| 6 | 11.24706 | 5.75313 | [9.66811 , 12.82601] |
| 7 | 11.64706 | 4.67617 | [10.36369 , 12.93043] |
| 8 | 11.21569 | 3.63182 | [10.21893 , 12.21244] |
| 9 | 12.39216 | 5.44264 | [10.89842 , 13.88589] |
| 10 | 13.05010 | 13.19165 | [9.42965 , 16.67055] |

follows the same trend as the mean NETSTAT averages as plotted previously in Figure 4.5. Similar to the medium load ordering accuracy seen in Figure 4.4, as the number of events a part of a triplet declines, the number of doubles increases. This previous statement makes logical sense as the ordering of the events should be a `tasklist` entry, a NETSTAT entry, a `tcpdump` entry, and finally a Windows® log entry, so as the NETSTAT accuracy declines, this also removes the possibility of not only quartets from occurring, but also triplets from occurring. As the number of events a part of a triplet declines, the number of doubles increases. In fact, the triplets and doubles as plotted on the figure are nearly a mirror image of each other, as was also the case under a medium load. In this instance, the highest concentration of quartets and triplets is when NETSTAT only has a maximum of two processes, making up 58% of all events found for that test series, which makes sense as

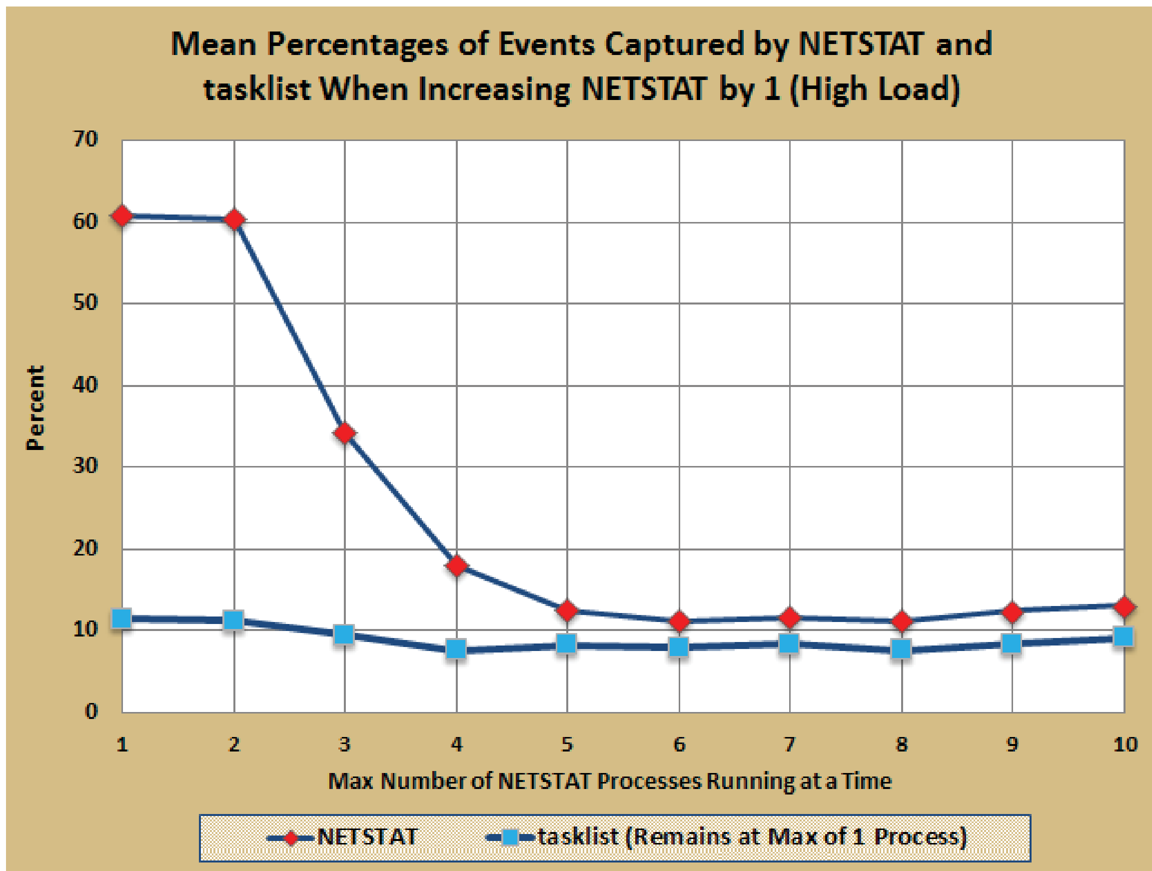


Figure 4.5: Mean percentages of events captured while increasing maximum number of `NETSTAT` processes by 1 (high load)

the optimal setting for `NETSTAT` under a high load (as previously shown in Table 4.3) is a maximum of either one or two processes.

#### *4.2.2 `tasklist` Sensor.*

This subsection presents three additional subsections. Each subsection presents the results of testing the `tasklist` sensor under the three different load factors. Beginning with the low load results, this subsection continues by presenting the results of the medium and high load tests.

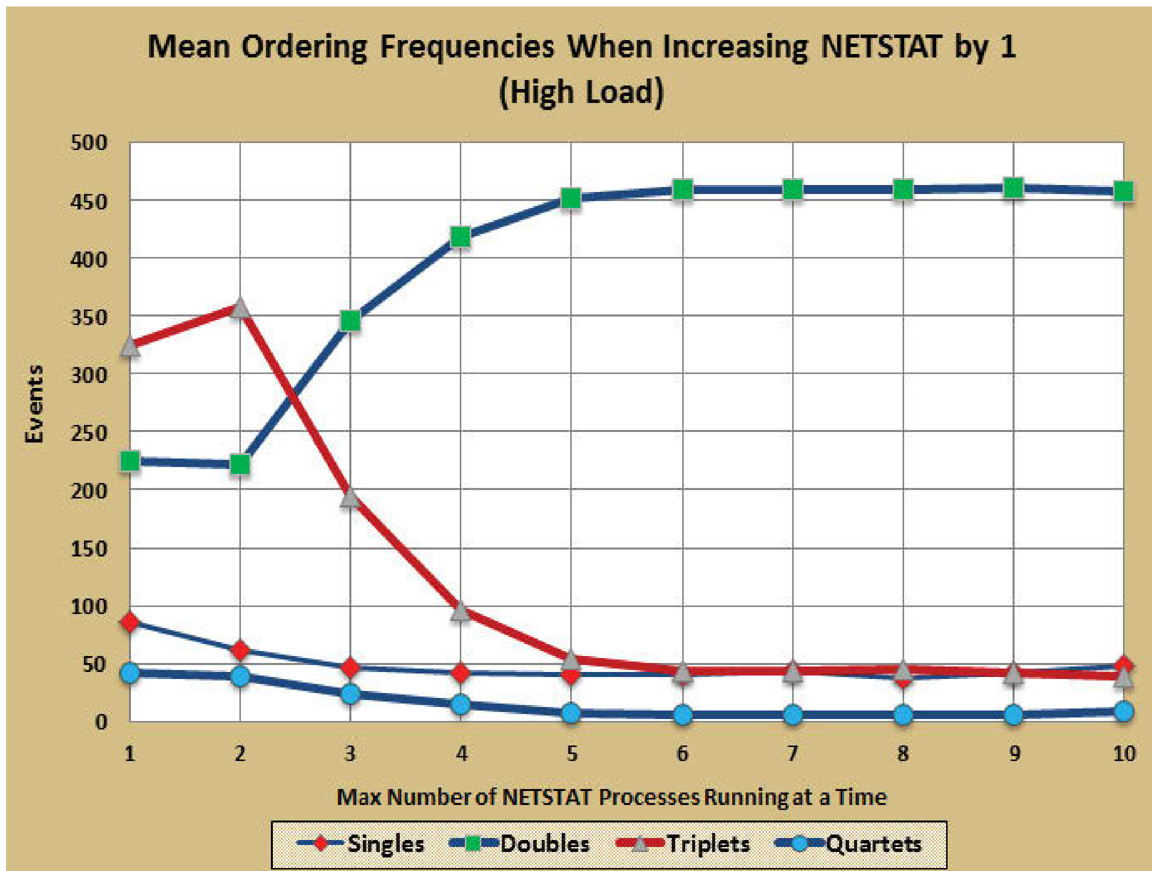


Figure 4.6: Mean ordering frequencies while increasing maximum number of `NETSTAT` processes by 1 (high load)

##### *4.2.2.1 `tasklist` Low Load Results.*

Table 4.4 provides a summary of the experiment when increasing the maximum number of `tasklist` processes under a low load. The Appendix contains complete performance summaries (Tables B.13 through B.16) for these tests under a low load which include the data for all sensors within the experiment. As previously mentioned, the research presents 60 series of tests. Table 4.4 shows the `tasklist` data for 10 of these 60 series of tests.

In Table 4.4, the results show that in order to capture the most `tasklist`-related events under a low load while increasing the maximum number of `tasklist` processes,

a maximum of two tasklist processes is ideal. Granted, the data also shows that the confidence intervals from the first test series to the eighth test series generally overlap, with the second test series having the highest upper bound and is thus considered the optimal setting. None of the series are actually able to capture 100% of the tasklist-related events every time, otherwise there would be a series in the table with an average of 100%. Likewise, no individual test from any series of tests actually captures 100% of the tasklist-related events.

Table 4.4: Performance summary of tasklist under a low load

| Max tasklist Processes | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|
| 1 | 87.19216 | 5.23098 | [85.75652 , 88.62780] |
| 2 | 88.19608 | 4.04564 | [87.08575 , 89.30640] |
| 3 | 87.35686 | 4.34885 | [86.16332 , 88.55040] |
| 4 | 86.69020 | 5.23011 | [85.25479 , 88.12560] |
| 5 | 86.12549 | 8.62313 | [83.75887 , 88.49211] |
| 6 | 85.47451 | 3.55662 | [84.49840 , 86.45062] |
| 7 | 85.09804 | 4.35162 | [83.90374 , 86.29234] |
| 8 | 85.27059 | 4.59336 | [84.00994 , 86.53123] |
| 9 | 81.99216 | 4.41560 | [80.78030 , 83.20402] |
| 10 | 80.44706 | 5.21023 | [79.01711 , 81.87701] |

Figure 4.7 provides a visual depiction of the mean percentages of tasklist-related events captured under low load while increasing maximum number of tasklist processes by one from one up to ten, but keeping NETSTAT at a maximum of one process. The figure

shows that increasing the maximum number of `tasklist` processes actually appears to have a negative effect on not only the number of captured `tasklist`-related events, but also appears to have a negative effect on the amount of captured `NETSTAT`-related events. However, unlike the `NETSTAT` sensor under a low load (previously shown in Figure 4.1) where the impact between `NETSTAT` versus `tasklist`-captured events is significant, the impact in Figure 4.7 is not as significant. Though, keep in mind that the scale along the Y-axis in the figure below is not the same as previously shown in Figure 4.7, and thus the visual declination of the data points in the figure below may seem more drastic compared to Figure 4.7. However, the results still show a decline in both the amount of `NETSTAT` and `tasklist`-captured events. As the maximum number of `tasklist` processes increases by one from one up to ten and the maximum number of `NETSTAT` processes remains at one, both the amount of data that `NETSTAT` and `tasklist` capture declines. While in the case of `NETSTAT`-captured events where the amount of events captured declines from the first series of tests, the amount of `tasklist`-captured events does not decline until after the second series of tests. The `tasklist` process creates about 5 threads per execution, while the `NETSTAT` process only creates about 3 threads per execution. The previous statement explains why when increasing the maximum number of `tasklist` processes the results show a significantly larger impact on the overall accuracy of not only the `tasklist` sensor but also the `NETSTAT` sensor. The same logic as previously described for the `NETSTAT` sensor under a low load in Subsection 4.2.1.1 holds true for this test series. The `tasklist` threads wait on the same resources, thus giving `NETSTAT` threads more time to execute. In addition to creating more threads than the `NETSTAT` process, `tasklist` processes also generate more textual data to save to disk on average, at least in the case of this research. The previous sentence describes what may be an additional factor when explaining why the `tasklist` sensor performs poorer than the `NETSTAT` sensor under every workload in this research.

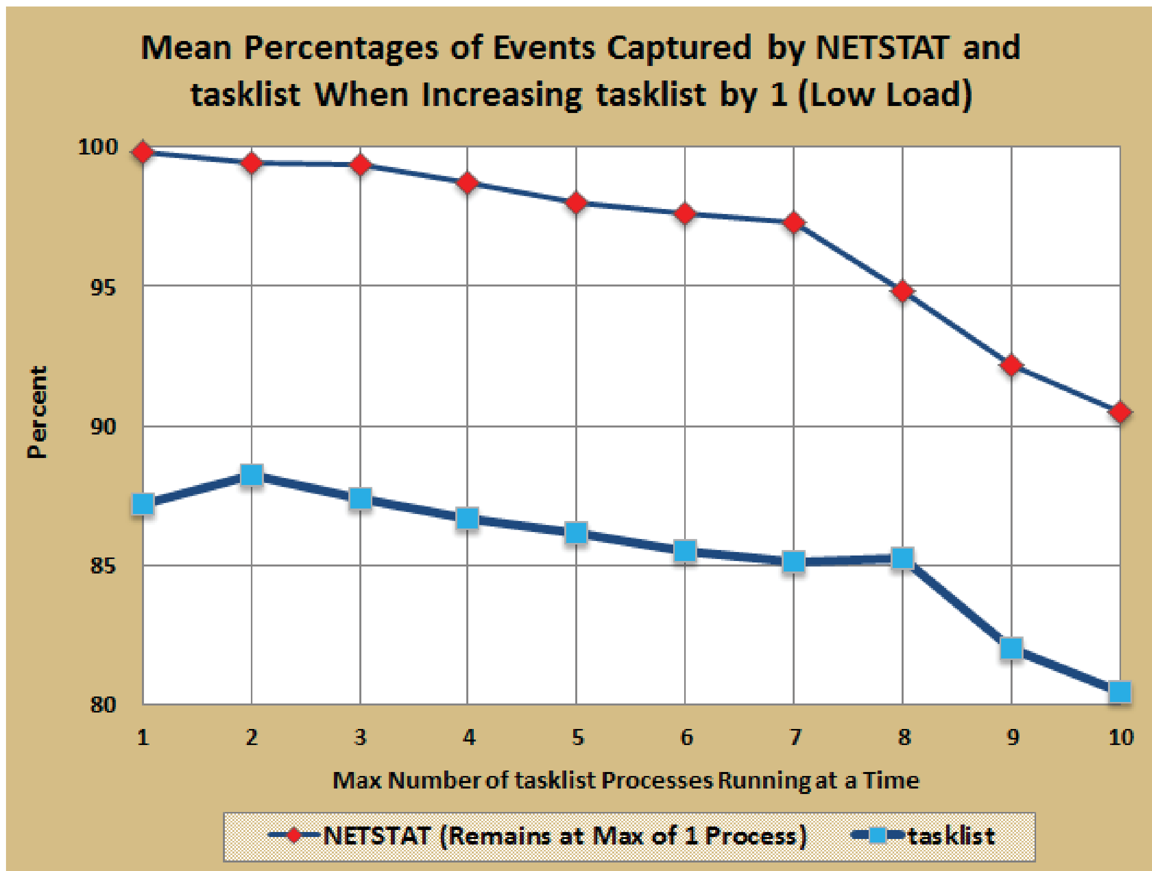


Figure 4.7: Mean percentages of events captured while increasing maximum number of `tasklist` processes by 1 (low load)

Accordingly, as the amount of data captured declines, the overall amount of captured events for each series of tests also declines, which Figure 4.7 also depicts. Note that neither the `tcpdump` sensor data nor the Windows® log sensor data is present in Figure 4.7 as it clutters the figure. The Windows® log sensor maintained a 100% average throughout all 10 series of tests shown in Table 4.4.

Figure 4.8 provides a visual depiction of the ordering accuracy when increasing the maximum number of `tasklist` processes under a low load. Similar to the ordering accuracy seen for the `NETSTAT` sensor under a low load, the number of quartets peaks right at the beginning, when `tasklist` only has a maximum of one process, and then

progressively declines, leveling out beginning with the fifth series. As the number of quartets declines, then number of triplets increases from the first series to the fifth series and then progressively declines along with the number of quartets while doubles and singles progressively increase at the same time (beginning with the fifth test series). Because both the NETSTAT and tasklist sensors accuracies decline, the number of quartets declines more drastically. In this instance, the highest concentration of quartets and triplets is when tasklist has a maximum of two processes, making up 72% of all events found for that test series. This high concentration of quartets and triplets in the second series follows the same pattern as with the optimal setting for maximum number of tasklist processes under a low load (which is two).

##### *4.2.2.2 tasklist Medium Load Results.*

Table 4.5 provides a summary of the experiment when increasing the maximum number of tasklist processes under a medium load. The Appendix contains complete performance summaries (Tables B.17 through B.20) for these tests under a medium load which include the data for all sensors within the experiment. As previously mentioned, the research presents 60 series of tests. Table 4.5 shows the NETSTAT data for 10 of these 60 series of tests.

The data provided in Table 4.5 is not as definitive as was the case in some of the previous sections. Based on the data in Table 4.5, the optimal setting for tasklist under a medium load is a maximum of five tasklist processes. However, the data also shows that the confidence intervals for maximum tasklist processes of four and six through nine overlap with the confidence interval of a maximum of five tasklist processes, which means that any test using a maximum of four to nine tasklist processes may produce about the same percentage of events captured. When increasing the maximum number of NETSTAT processes under a medium load (in Section 4.2.1.2), the results display an increase 7.3% at the beginning in average percentage of events captured followed by a

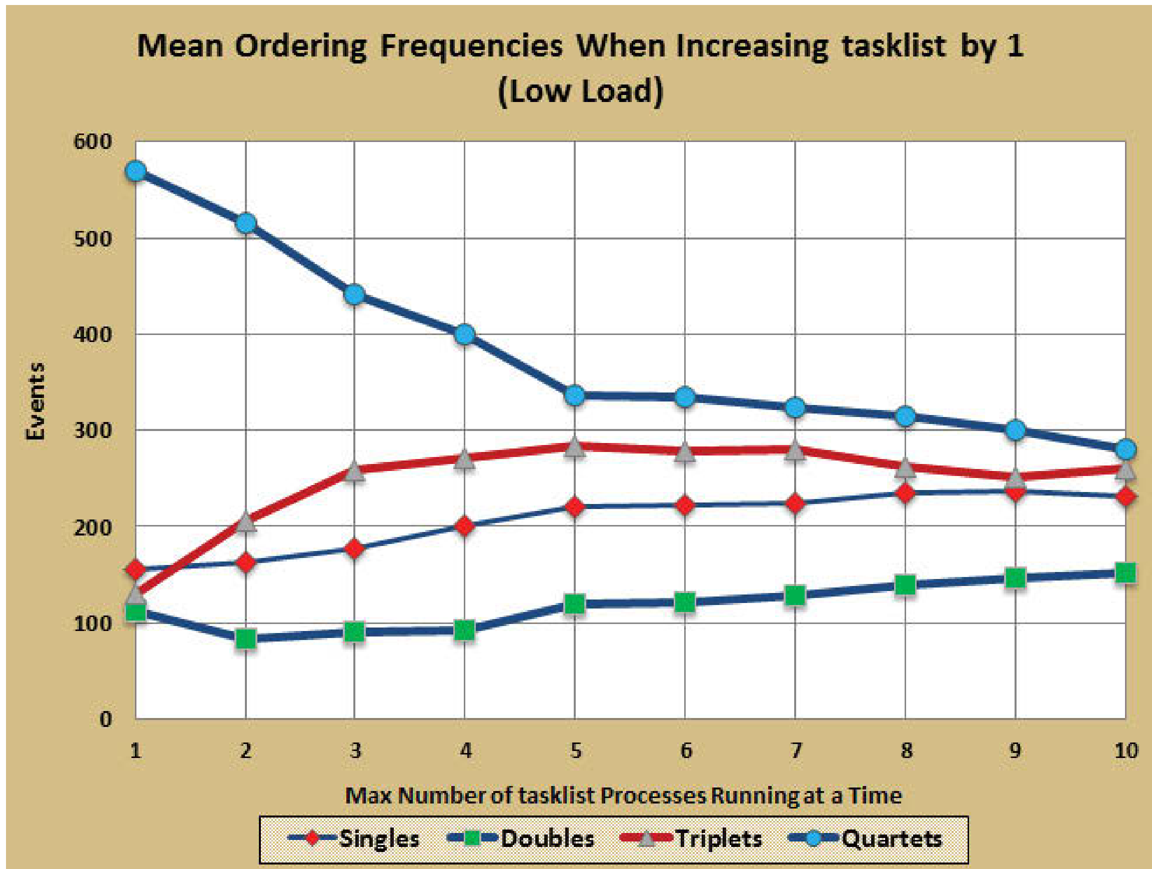


Figure 4.8: Mean ordering frequencies while increasing maximum number of `tasklist` processes by 1 (low load)

significant decline of 21% between three and four and then an additional declination of 22% between four and five and then finally the percentages level out. However, when increasing the maximum number of `tasklist` processes under a medium load, the results display an increase of 6.5% at the beginning in average percentage of events captured, but a significant decline does not follow. Instead, the average percentage of `tasklist`-related events captured stays around the same percentage, between 30% and 40%.

Figure 4.9 provides a visual depiction of the mean percentages of `tasklist`-related events captured under medium load while increasing maximum number of `tasklist`

Table 4.5: Performance summary of `tasklist` under a medium load

| Max `tasklist` Processes | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|
| 1 | 24.87288 | 5.95687 | [23.23802 , 26.50774] |
| 2 | 31.37596 | 6.39101 | [29.62195 , 33.12997] |
| 3 | 32.57414 | 7.83719 | [30.42322 , 34.72505] |
| 4 | 34.59516 | 7.05151 | [32.65987 , 36.53044] |
| 5 | 37.62518 | 9.29846 | [35.07321 , 40.17714] |
| 6 | 34.40239 | 6.13960 | [32.71738 , 36.08740] |
| 7 | 36.51841 | 7.57323 | [34.43994 , 38.59688] |
| 8 | 33.39369 | 6.73710 | [31.54469 , 35.24268] |
| 9 | 34.35955 | 8.21924 | [32.10378 , 36.61532] |
| 10 | 28.38737 | 9.50078 | [25.77989 , 30.99486] |

processes by one from one up to ten, but keeping `NETSTAT` at a maximum of one process. The `NETSTAT` and `tasklist` sensors appear to hold about the same trend in the figure, specifically maintaining about the same accuracy for their respective events with small increases and decreases, and a slightly more drastic decrease from the ninth to the tenth series of tests (9 to 10 along the X-axis). The `NETSTAT` sensor data and `tasklist` sensor data also had mimicking trends when increasing the maximum number of `tasklist` processes under a low load, as previously shown in Figure 4.7. The average percentage of `NETSTAT`-related events captured stays around the same percentage, between 71% and 76.6%, which is quite interesting given the `NETSTAT` sensor's steep declination back in Subsection 4.2.1.2, specifically referencing Figure 4.3. Given the `NETSTAT` sensor remains at a maximum of one process in this instance, the case may be that *enough* `NETSTAT`

processes launch without forcing an excessive amount of `NETSTAT`-related threads have to wait on the same resources, as was the theoretical reasoning behind the steep declination in Figure 4.3.

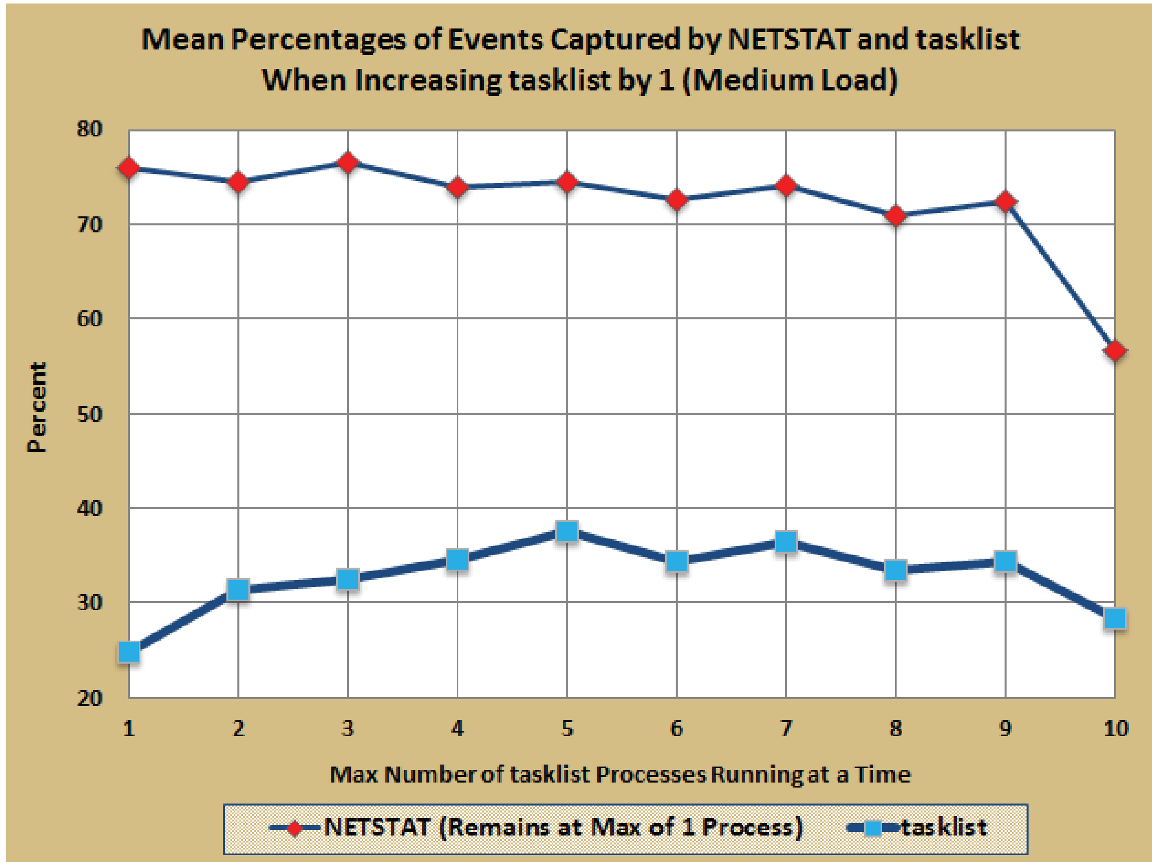


Figure 4.9: Mean percentages of events captured while increasing maximum number of `tasklist` processes by 1 (medium load)

Figure 4.10 provides a visual depiction of the ordering accuracy when increasing the maximum number of `tasklist` processes under a medium load. As the figure shows, the amount of quartets, triplets, doubles, and singles actually maintains about the same consistency from start to finish, with the exception of the last test series. Accordingly, the declination shown in the ordering Figure 4.10 (from the ninth to the tenth test series) makes

sense because both the NETSTAT and tasklist sensors' capture accuracy decreases from the ninth to the tenth test series (as previously shown in Figure 4.9).

As the test series overall accuracy declines, so does the frequency of events that are quartets and triplets. Note that as seen in certain cases before, the triplets trend is nearly the same as the NETSTAT sensor's trend shown in Figure 4.9. This makes logical sense given that if the accuracy of the NETSTAT sensor goes down, then the number of quartets and triplets will decline. However, given the NETSTAT sensor's accuracy does not decline much, neither does the number of quartets or triplets shown in Figure 4.10. Likewise, the tasklist sensor's accuracy is low and thus the number of quartets is low given the tasklist-related entry is the first in the quartet chain. This observation also explains why the results show more triplets than anything else. In this instance, the highest concentration of quartets and triplets is when tasklist has a maximum of one process, making up 70% of all events found for that test series.

#### *4.2.2.3 tasklist High Load Results.*

Table 4.6 provides a summary of the experiment when increasing the maximum number of tasklist processes under a high load. The Appendix contains complete performance summaries (Tables B.21 through B.24) for these tests under a high load which include the data for all sensors within the experiment. As previously mentioned, the research presents 60 series of tests. Table 4.6 shows the tasklist data for 10 of these 60 series of tests.

The data provided in Table 4.6 is not as definitive as some of the previous sections, particularly NETSTAT under low and medium loads and also tasklist under a low load. Though, the results for this test series are essentially the same as the previous test series under a medium load, the only difference is that the NETSTAT and tasklist sensors are ~10 to 15% lower in this instance. Based on the data in Table 4.6, the optimal setting for the tasklist sensor under a high load can be anywhere from three to ten tasklist

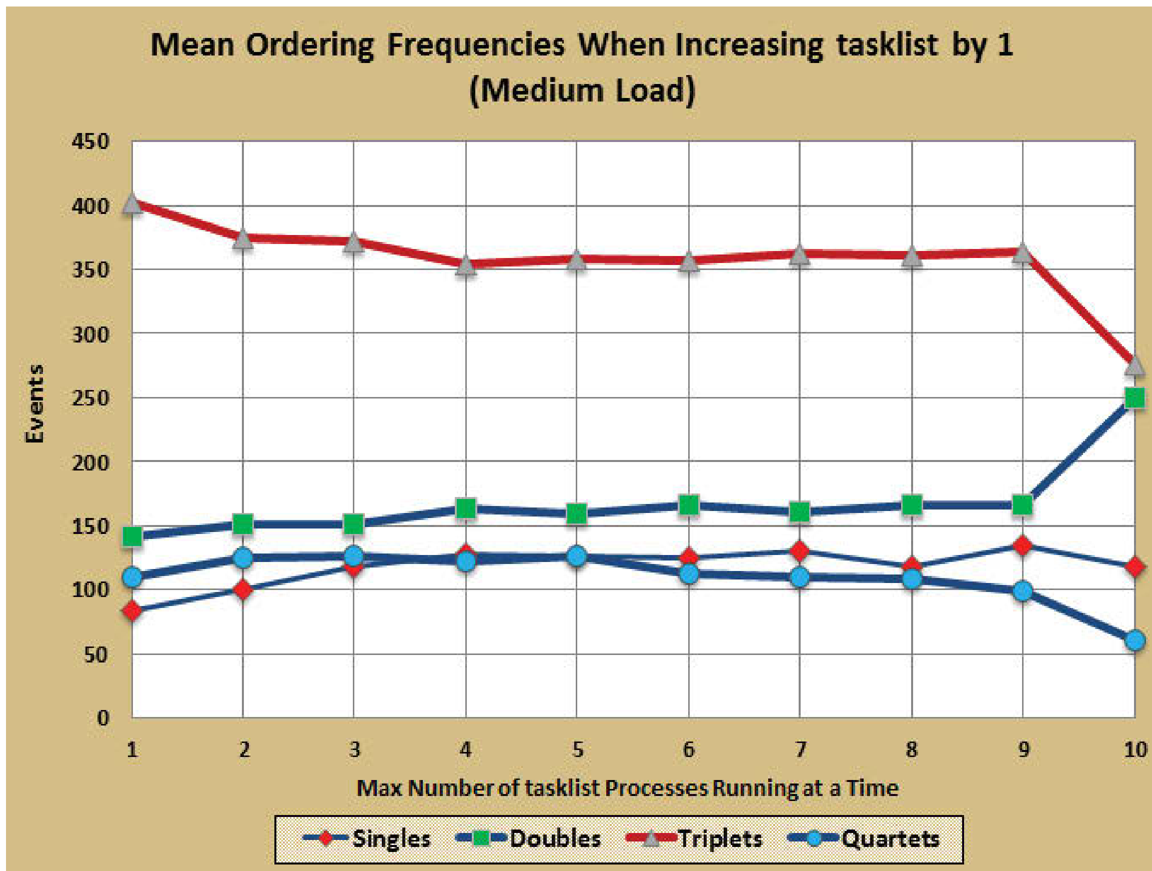


Figure 4.10: Mean ordering frequencies while increasing maximum number of `tasklist` processes by 1 (medium load)

processes, with a maximum of six `tasklist` processes providing the highest accuracy. However, it is worth noting that the confidence interval for a maximum of six `tasklist` processes is the widest of all `tasklist` settings shown in Table 4.6. Similar to the low and medium load sections discussing `tasklist` data, the high load data shows that none of the series are actually able to capture 100% of the `tasklist`-related events every time. Note the absence of a series in the table with an average of 100%. Like with the results under a low and medium load in Sections 4.2.2.1 and 4.2.2.2, respectively, no individual test from any test series under a high load actually captures 100% of the `NETSTAT`-related events.

Table 4.6: Performance summary of `tasklist` capture accuracy under a high load

| Max `tasklist` Processes | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|
| 1 | 11.45098 | 3.64940 | [10.44940 , 12.45256] |
| 2 | 17.27843 | 5.24779 | [15.83818 , 18.71869] |
| 3 | 21.12157 | 5.41643 | [19.63503 , 22.60811] |
| 4 | 22.46275 | 5.79565 | [20.87213 , 24.05336] |
| 5 | 20.83137 | 5.82345 | [19.23313 , 22.42962] |
| 6 | 23.13725 | 6.86714 | [21.25257 , 25.02194] |
| 7 | 20.98827 | 6.81429 | [19.11809 , 22.85846] |
| 8 | 20.61176 | 5.93369 | [18.98326 , 22.24026] |
| 9 | 22.39216 | 5.94615 | [20.76024 , 24.02408] |
| 10 | 22.10196 | 6.40991 | [20.34276 , 23.86116] |

Figure 4.11 provides a visual depiction of the mean percentages of `tasklist`-related events captured under a high load while increasing maximum number of `tasklist` processes by one from one up to ten, but keeping `NETSTAT` at a maximum of one process. The figure shows that after increasing the maximum number of `tasklist` processes to three (3 along the X-axis), the rest of the series of tests (from 3 to 10) display little to no improvement. The average percentage of `NETSTAT`-related events captured stays around the same percentage, between 53% and 61%. Similar to the tests under a medium load, the `NETSTAT` sensor appears to execute enough processes throughout each test series without forcing an excessive amount of `NETSTAT`-related threads have to wait on the same resources. Again, in this instance, not every file in the scripted file-download scenario actually downloads under a high load, due to the implementation of the high load scripts.

As a result, the Windows® log sensor still captures 100% of the events as expected, but the other sensors only capture the number of events less than or equal to what the Windows® log sensor captures. Note that neither the `tcpdump` sensor data nor the Windows® log sensor data is present in Figure 4.11 as the figure becomes cluttered.

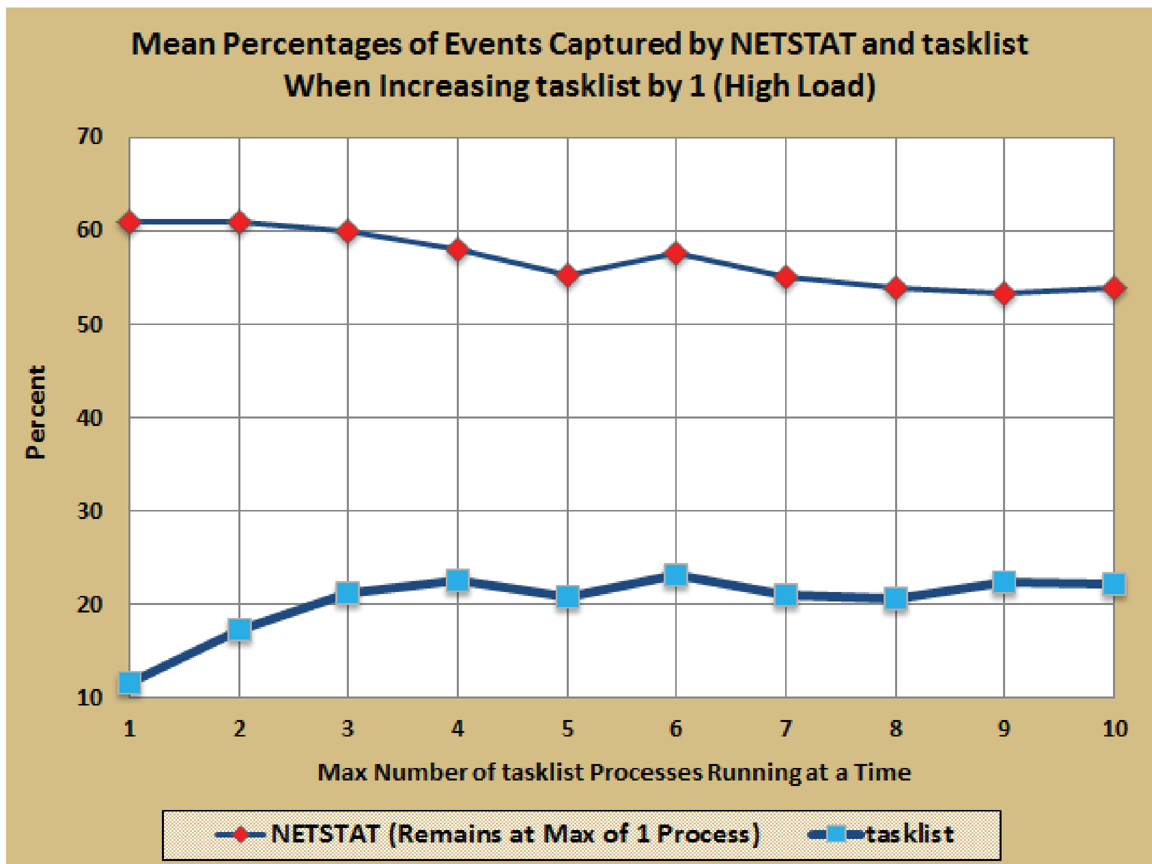


Figure 4.11: Mean percentages of events captured while increasing maximum number of `tasklist` processes by 1 (high load)

Figure 4.12 provides a visual depiction of the ordering accuracy when increasing the maximum number of `tasklist` processes under a high load. Given this round of tests is under a high load, the number of quartets is expectedly lower than the tests under both a low and medium load. The number of triplets is higher than previously seen under a low

load, but less than previously seen under a medium load. The previous statement makes sense because there are more quartets under a low load. As the number of doubles and triplets declines, the number of singles increases, as is seen in series two through four (2 through 4 along the X-axis).

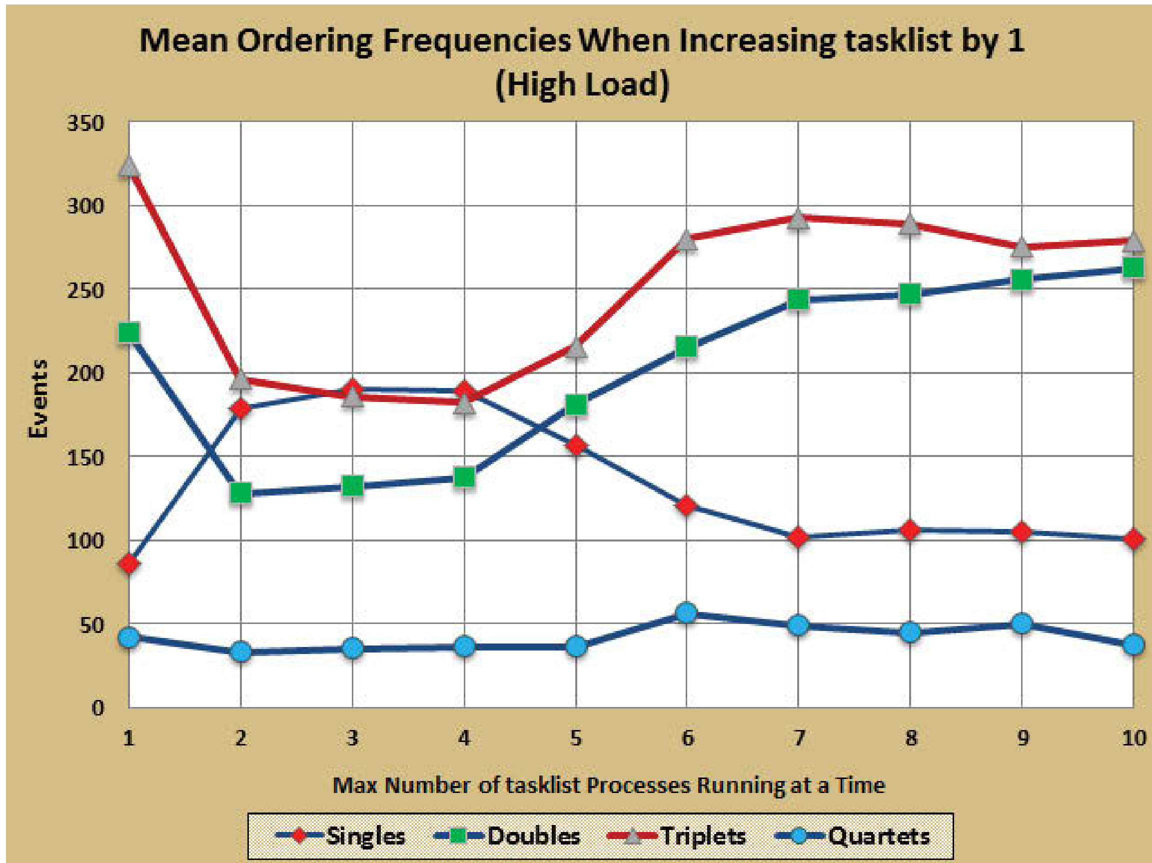


Figure 4.12: Mean ordering frequencies while increasing maximum number of `tasklist` processes by 1 (high load)

In this instance, the results show no discernable pattern like in the previous ordering accuracy figures. Normally, the trend that the triplets plot follows is the same trend as the `NETSTAT` sensor's captured-event accuracy. However, in this instance, the order accuracy of `NETSTAT` sensor, while capturing about the same number of events from the first to the

tenth series, somehow becomes more accurate toward the latter series of tests. While this explanation does not seem to make logical sense, referring back to Figure 4.5 shows that executing more `NETSTAT` processes results in the `NETSTAT` sensor's significant accuracy decline. But, in the case of Figure 4.11, the `NETSTAT` sensor remains at a maximum of only one process and maintains about the same capture accuracy throughout. Therefore, this theory, while illogical-sounding, may not be incorrect after all. Unfortunately, confirmation of the previous theory is impossible without data pertaining to the OS scheduler throughout each of these tests. Regardless, regarding the ordering accuracy in Figure 4.12, the highest concentration of quartets and triplets is when the `tasklist` sensor only has a maximum of one process, making up 54% of all events found for that test series.

### 4.3 Summary

The system under test proves to be mostly inaccurate regarding chronological event ordering. Additionally, the system under test considerably taxes the CPU of the experimental system, effectively maxing it out at 100% from the start of each test to the finish, making even minimal use of the OS a difficult task. This chapter summarizes the analysis and statistical results of the experiments conducted during this research. This chapter also attemps to logically explain some of the unforeseen results within the experiments.

# V. Conclusions and Recommendations

## 5.1 Chapter Overview

This chapter concludes the thesis. It begins by presenting several of the main points derived from the research. The next section highlights the contributions of the research. Finally, this chapter concludes by presenting three ideas for future research.

## 5.2 Conclusions of Research

One goal of this research is to investigate and subsequently identify a method that utilizes multithreading for producing a timeline consisting of locally-produced and collected network and system-related events pertaining to file downloads. Identification of such a method occurred. However, the previously presented results in Chapter IV prove that the identified method is neither overly accurate nor consistent under all imposed system and network-traffic workloads.

The results show that this type of multithreading logging technique is not overly accurate, in not only events captured, but also the ordering of events captured, at least pertaining to the `NETSTAT` and `tasklist` Windows® utilities. Even so, logging events is only half of the challenge with the other half being the actual logging of the events in actual order of occurrence. For example, the resulting timeline of the logging suite should show the *Wget* executable running in `tasklist` prior to `NETSTAT` showing the connection that same *Wget* executable established with a remote server, however the analysis of the results shows that the *Wget* executable does appear in `tasklist` prior to `NETSTAT`.

Creating new threads, other than what the logging suite creates itself, means that far more threads contend for CPU time. Creating more threads while the logging suite is active will inevitably decrease the overall accuracy of the logging suite, as previously seen in the

results. Even so, this research was able to determine the optimal settings for the `NETSTAT` and `tasklist` sensors under low, medium, and high loads.

### 5.3 Research Contributions

This research identifies a method that utilizes multithreading for producing a timeline consisting of locally-produced and collected network and system-related events pertaining to file downloads. Specifically, this research investigates how differences in thread allocation amongst the sensors, as well as imposed system and network-traffic workloads, affect the accuracy of the resulting timeline. The following list describes the contributions of this research:

- an exploration into the effects of sensor multithreading,
- an application and subsequent statistical analysis of sensor multithreading for the purpose of logging events pertaining to specific file downloads, and
- finding the optimal maximum thread count for the `NETSTAT` and `tasklist` sensors within the logging suite for the scenario and values tested.

### 5.4 Recommendations for Future Research

The research saw some unexpected results. Exploration into why the results displayed unexpected results provide some ideas for future research. The following list provides recommendaions for future research.

- The mean average percentage of events captured by the `NETSTAT` and `tasklist` sensors when increasing the `NETSTAT` sensor by one process under a low load saw the `tasklist` sensor's percentage drastically dip, but then gradually increase, and then finally decline again. The focus of this research was mainly on the resulting timelines and the accuracy of the deployed sensors, and only limited visibility into the OS scheduling is available. One course for future research is to execute

this experiment, but instead of focusing on the timelines themselves, focus on OS scheduling internals. One specific result to analyze is why the results of this research show an increase of a sensor's (`tasklist`) accuracy that had fewer resources (launched processes) than another sensor (`NETSTAT`).

- Analyze why the particular `tcpdump` filter used did not always capture 100% of the events. This research did not focus specifically on the `tcpdump` sensor enough to investigate this issue. Regardless, investigating the questions of how and why, pertaining to the aforementioned anomaly, may prove to be a worthwhile research topic.

- Trying to implement the method proposed and scenario outlined in this research in another version of Windows® or on Linux® to see if there are similar or dissimilar results may also prove to be a worthwhile research topic.

## Appendix A: Miscellaneous Figures

**<u>tcpdump.exe -l -i 3 -n -s 0 "tcp[((tcp[12:1] & 0xf0) >> 2):4] = 0x47455420"</u>**

- **<u>tcpdump.exe</u>** = path to the tcpdump executable
- **<u>-l</u>** = buffers stdout such that it can be either outputted to the command prompt/terminal or outputted to a file
- **<u>-i 3</u>** = the network interface on which to listen
- **<u>-n</u>** = Disables name resolution
- **<u>-s 0</u>** = Sets the snapshot length (*snaplen*) or in other words, how much data will actually be captured; a value of 0 defaults to 65,535 bytes
- **<u>((tcp[12:1] & 0xf0) >> 2)</u>** = length of the TCP header
- **<u>tcp[((...)):4] = 0x47455420</u>** = looks for 4 bytes (0x47455420) which correspond to the ASCII characters **G** (0x47), **E** (0x45), **T** (0x54), and ***space*** (0x20)

Figure A.1: Breakdown of the `tcpdump` command used within the experiment

## Appendix B: Complete Performance Summaries

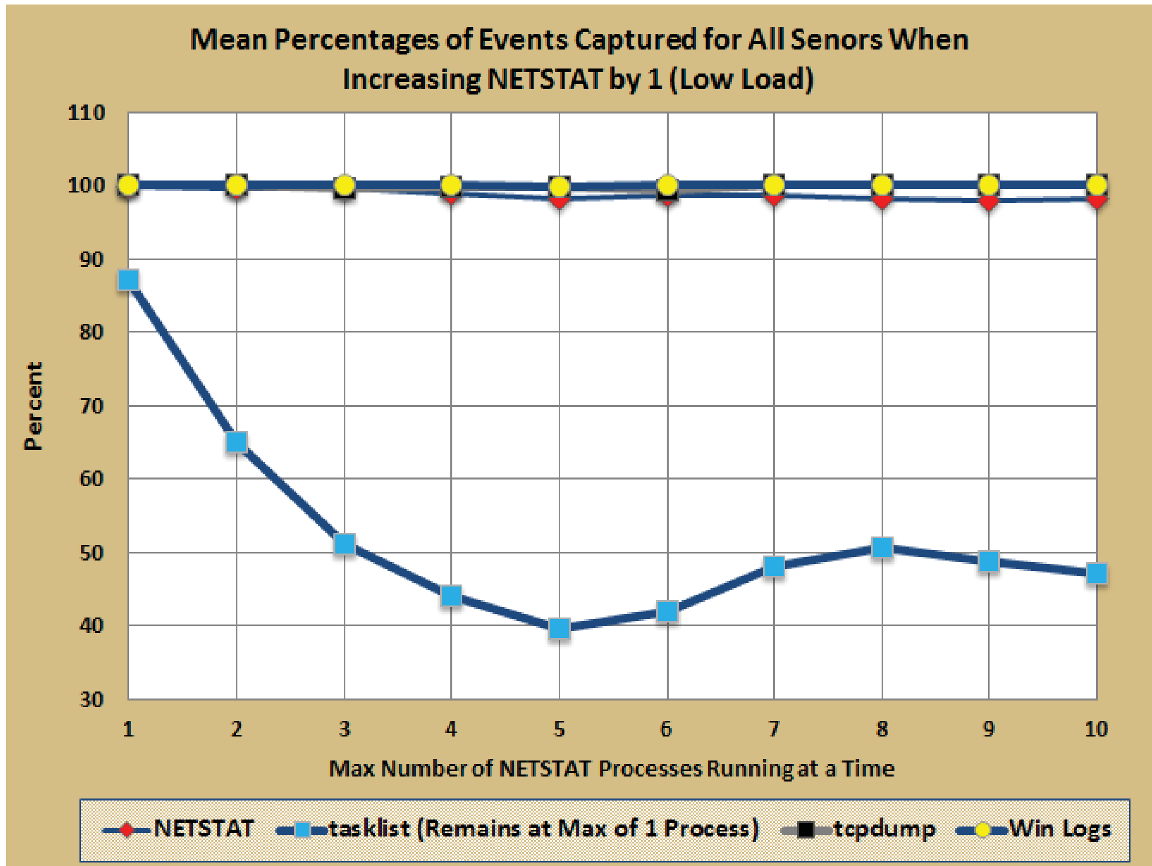


Figure B.1: Mean percentages of events captured for all sensors when increasing NETSTAT by 1 (low load)

Table B.1: Complete performance summary of NETSTAT under a low load (1 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 1 | NETSTAT | 99.83529 | 0.40141 | [99.72513 , 99.94546] |
| | tasklist | 87.19216 | 5.23098 | [85.75652 , 88.62780] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 96.75686 | 1.32034 | [96.39450 , 97.11923] |
| 2 | NETSTAT | 99.70981 | 0.58284 | [99.54984 , 99.86976] |
| | tasklist | 65.11373 | 9.01317 | [62.64006 , 67.58739] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 91.20588 | 2.28966 | [90.57748 , 91.83428] |
| 3 | NETSTAT | 99.49804 | 0.71484 | [99.30185 , 99.69423] |
| | tasklist | 51.20000 | 9.18468 | [48.67927 , 53.72073] |
| | tcpdump | 99.70980 | 0.85071 | [99.47633 , 99.94328] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 87.60196 | 2.29813 | [86.97124 , 88.23268] |

Table B.2: Complete performance summary of NETSTAT under a low load (2 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 4 | NETSTAT | 98.88627 | 1.22720 | [98.54947 , 99.22308] |
| | tasklist | 44.11765 | 10.31110 | [41.28777 , 46.94753] |
| | tcpdump | 99.85098 | 0.59308 | [99.68821 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 85.71373 | 2.45528 | [85.03987 , 86.38758] |
| 5 | NETSTAT | 98.23529 | 1.41574 | [97.84674 , 98.62384] |
| | tasklist | 39.59216 | 5.25843 | [38.14898 , 41.03533] |
| | tcpdump | 99.73333 | 0.71405 | [99.53736 , 99.92930] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 84.39020 | 1.46864 | [83.98713 , 84.79326] |
| 6 | NETSTAT | 98.76078 | 1.33448 | [98.39454 , 99.12703] |
| | tasklist | 42.38431 | 6.51986 | [40.59494 , 44.17369] |
| | tcpdump | 99.48235 | 1.24269 | [99.14130 , 99.82341] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 86.97843 | 2.06972 | [86.41040 , 87.54646] |

Table B.3: Complete performance summary of NETSTAT under a low load (3 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 7 | NETSTAT | 98.00000 | 1.59599 | [97.56198 , 98.43802] |
| | tasklist | 53.38039 | 6.55967 | [51.58009 , 55.18069] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 87.84510 | 1.86068 | [87.33444 , 88.35576] |
| 8 | NETSTAT | 97.98431 | 1.40618 | [97.59839 , 98.37024] |
| | tasklist | 53.00392 | 6.67527 | [51.17190 , 54.83595] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 87.74706 | 1.84958 | [87.23944 , 88.25468] |
| 9 | NETSTAT | 97.89804 | 2.22832 | [97.28648 , 98.50960] |
| | tasklist | 53.02745 | 5.71104 | [51.46006 , 54.59485] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 87.73137 | 1.81389 | [87.23355 , 88.22919] |

Table B.4: Complete performance summary of NETSTAT under a low load (4 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 10 | NETSTAT | 96.73725 | 2.33015 | [96.09775 , 97.37676] |
| | tasklist | 47.56863 | 6.28947 | [45.84248 , 49.29477] |
| | tcpdump | 99.86667 | 0.54062 | [99.71829 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 86.04314 | 1.91961 | [85.51630 , 86.56997] |

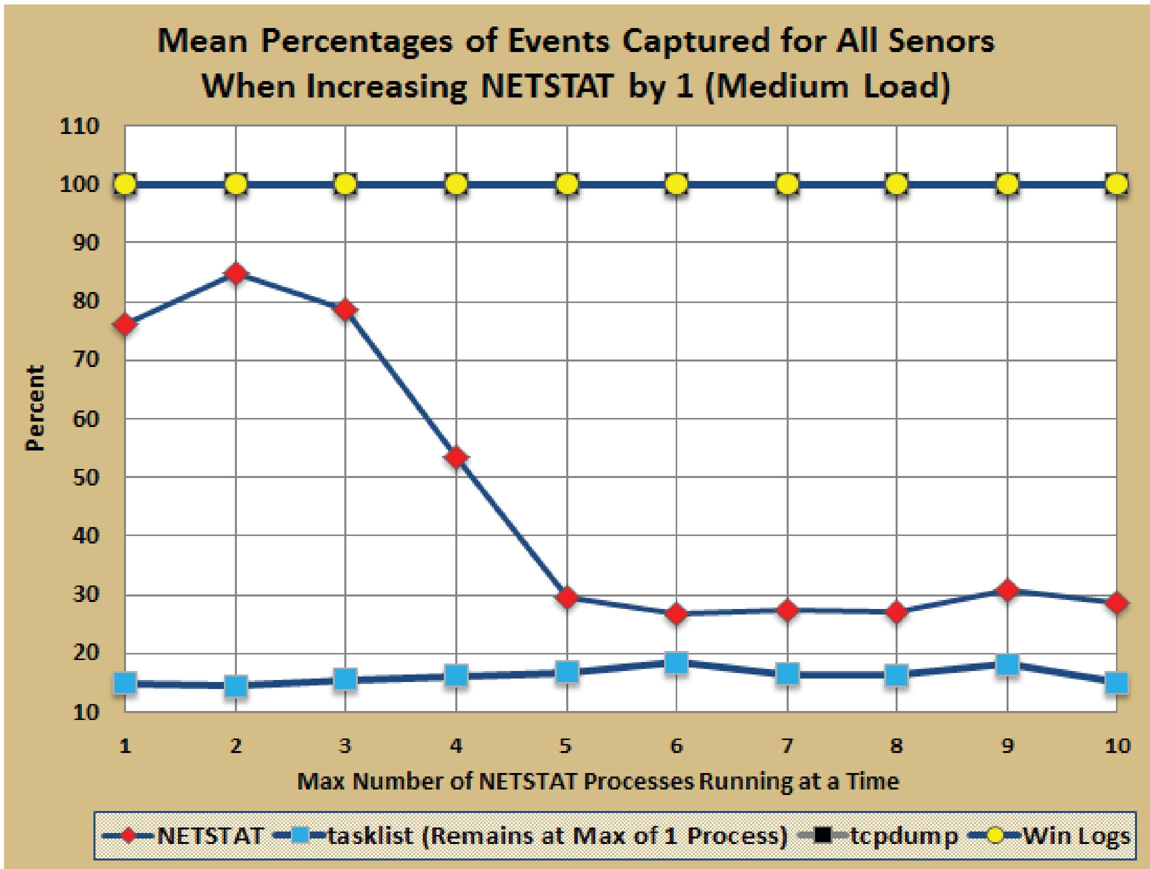


Figure B.2: Mean percentages of events captured for all sensors when increasing NETSTAT by 1 (medium load)

Table B.5: Complete performance summary of NETSTAT under a medium load (1 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 1 | NETSTAT | 76.18039 | 7.13361 | [74.22257 , 78.13821] |
| | tasklist | 14.93333 | 2.89846 | [14.13785 , 15.72882] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 72.77843 | 2.08617 | [72.20588 , 73.35098] |
| 2 | NETSTAT | 84.73725 | 6.48463 | [82.95755 , 86.51696] |
| | tasklist | 14.52549 | 2.79370 | [13.47172 , 15.57926] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 74.81569 | 1.98750 | [74.27022 , 75.36115] |
| 3 | NETSTAT | 78.55686 | 10.42369 | [75.69608 , 81.41764] |
| | tasklist | 15.49804 | 3.54612 | [14.52481 , 16.47127] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 73.51373 | 2.79578 | [72.74642 , 74.28103] |

Table B.6: Complete performance summary of NETSTAT under a medium load (2 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 4 | NETSTAT | 53.60784 | 19.94512 | [48.13391 , 59.08178] |
| | tasklist | 16.15686 | 4.09103 | [15.03408 , 17.27964] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 67.44118 | 4.91443 | [66.09241 , 68.78994] |
| 5 | NETSTAT | 29.48235 | 13.31088 | [25.82918 , 33.13552] |
| | tasklist | 16.79216 | 6.95885 | [14.88230 , 18.70201] |
| | tcpdump | 99.96078 | 0.28006 | [99.88392 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 61.55882 | 3.94686 | [60.47561 , 62.64204] |
| 6 | NETSTAT | 26.79216 | 9.68545 | [24.13399 , 29.45033] |
| | tasklist | 18.49412 | 5.43682 | [17.00198 , 19.98625] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 61.32157 | 3.23186 | [60.43458 , 62.20855] |

Table B.7: Complete performance summary of NETSTAT under a medium load (3 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 7 | NETSTAT | 27.32549 | 9.92080 | [24.60273 , 30.04825] |
| | tasklist | 16.47843 | 6.66886 | [14.64816 , 18.30870] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 60.95098 | 3.66559 | [59.94496 , 61.95700] |
| 8 | NETSTAT | 27.25490 | 7.35609 | [25.23602 , 29.27378] |
| | tasklist | 16.32941 | 5.92686 | [14.70278 , 17.95604] |
| | tcpdump | 99.98431 | 0.11202 | [99.95357 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 60.89216 | 2.88339 | [60.10081 , 61.68350] |
| 9 | NETSTAT | 30.72157 | 10.41599 | [27.86290 , 33.58024] |
| | tasklist | 18.14118 | 6.95256 | [16.23305 , 20.04931] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 62.21569 | 4.02305 | [61.11156 , 63.31981] |

Table B.8: Complete performance summary of NETSTAT under a medium load (4 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 10 | NETSTAT | 28.71373 | 8.33741 | [26.42552 , 31.00193] |
| | tasklist | 15.15294 | 6.80172 | [13.28621 , 17.01967] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 60.96667 | 3.30894 | [60.05853 , 61.87480] |

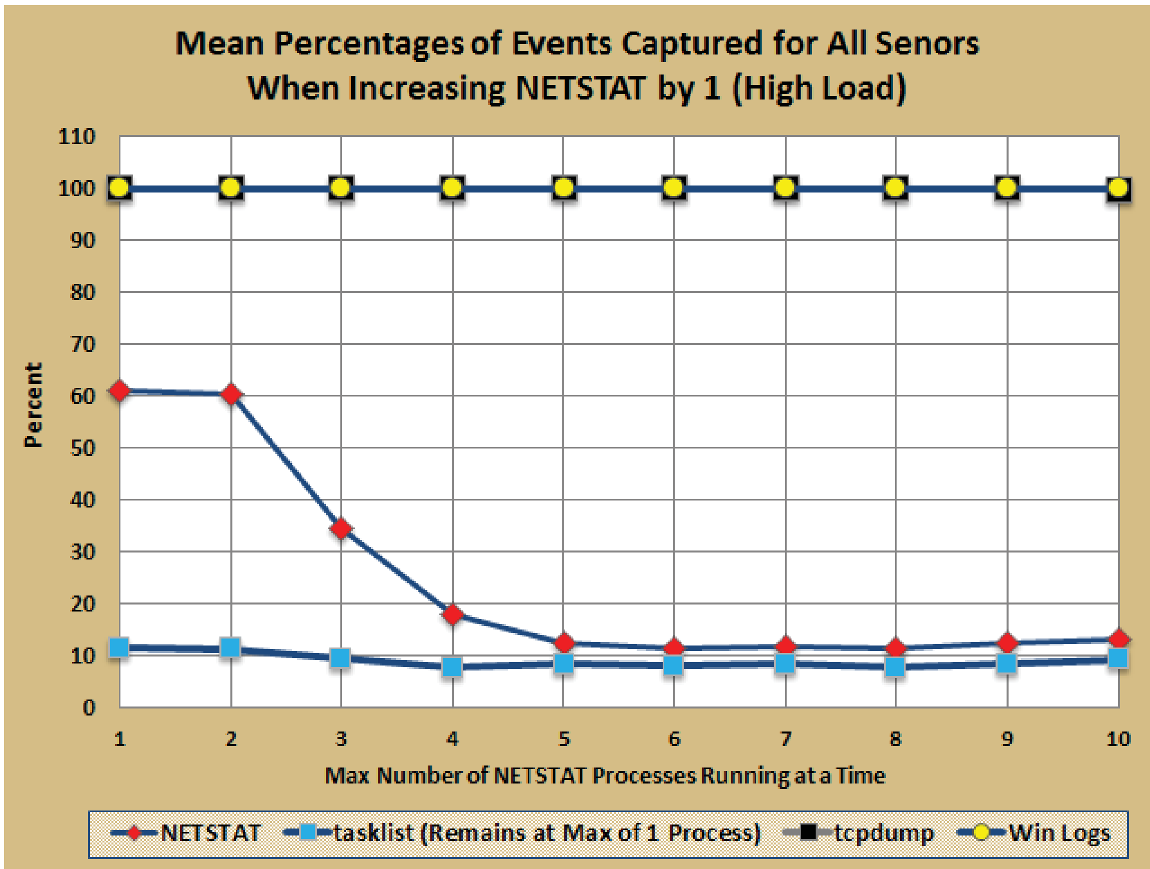


Figure B.3: Mean percentages of events captured for all sensors when increasing `NETSTAT` by 1 (high load)

Table B.9: Complete performance summary of NETSTAT under a high load (1 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 1 | NETSTAT | 60.90980 | 15.68634 | [56.60469 , 65.21492] |
| | tasklist | 11.45098 | 3.64940 | [10.44940 , 12.45256] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 68.09020 | 4.42377 | [66.87609 , 69.30430] |
| 2 | NETSTAT | 60.40784 | 20.17103 | [54.87191 , 65.94378] |
| | tasklist | 11.29412 | 3.57555 | [10.31281 , 12.27543] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 67.92745 | 5.65466 | [66.37553 , 69.47937] |
| 3 | NETSTAT | 34.32157 | 19.31781 | [29.01980 , 39.62334] |
| | tasklist | 9.43529 | 4.79011 | [8.12065 , 10.74994] |
| | tcpdump | 99.97647 | 0.16803 | [99.93035 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 60.93333 | 5.63247 | [59.38750 , 62.47916] |

Table B.10: Complete performance summary of NETSTAT under a high load (2 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 4 | NETSTAT | 18.07059 | 13.21713 | [14.44315 , 21.69803] |
| | tasklist | 7.63922 | 4.35045 | [6.44523 , 8.83320] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 56.42745 | 4.09956 | [55.30233 , 57.55258] |
| 5 | NETSTAT | 12.54902 | 5.85585 | [10.94188 , 14.15616] |
| | tasklist | 8.29804 | 4.46199 | [7.07345 , 9.52263] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 55.21176 | 2.21284 | [54.60445 , 55.81908] |
| 6 | NETSTAT | 11.24706 | 5.75313 | [9.66811 , 12.82601] |
| | tasklist | 7.94510 | 5.09181 | [6.54765 , 9.34254] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 54.79804 | 2.27486 | [54.17370 , 55.42237] |

Table B.11: Complete performance summary of NETSTAT under a high load (3 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 7 | NETSTAT | 11.64706 | 4.67617 | [10.36369 , 12.93043] |
| | tasklist | 8.41569 | 4.74904 | [7.11231 , 9.71906] |
| | tcpdump | 99.98431 | 0.11202 | [99.95357 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 55.01176 | 1.97804 | [54.46889 , 55.55464] |
| 8 | NETSTAT | 11.21569 | 3.63182 | [10.21893 , 12.21244] |
| | tasklist | 7.58431 | 4.63238 | [6.31296 , 8.85567] |
| | tcpdump | 99.99216 | 0.05601 | [99.97678 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 54.69804 | 1.64651 | [54.24615 , 55.14992] |
| 9 | NETSTAT | 12.39216 | 5.44264 | [10.89842 , 13.88589] |
| | tasklist | 8.40784 | 5.03332 | [7.02645 , 9.78924] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 55.20000 | 2.29390 | [54.57044 , 55.82956] |

Table B.12: Complete performance summary of NETSTAT under a high load (4 of 4)

| Max NETSTAT Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 10 | NETSTAT | 13.05010 | 13.19165 | [9.42965 , 16.67055] |
| | tasklist | 9.17943 | 7.22296 | [7.19709 , 11.16177] |
| | tcpdump | 99.83953 | 0.86377 | [99.60247 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 55.51727 | 4.81510 | [54.19577 , 56.83878] |

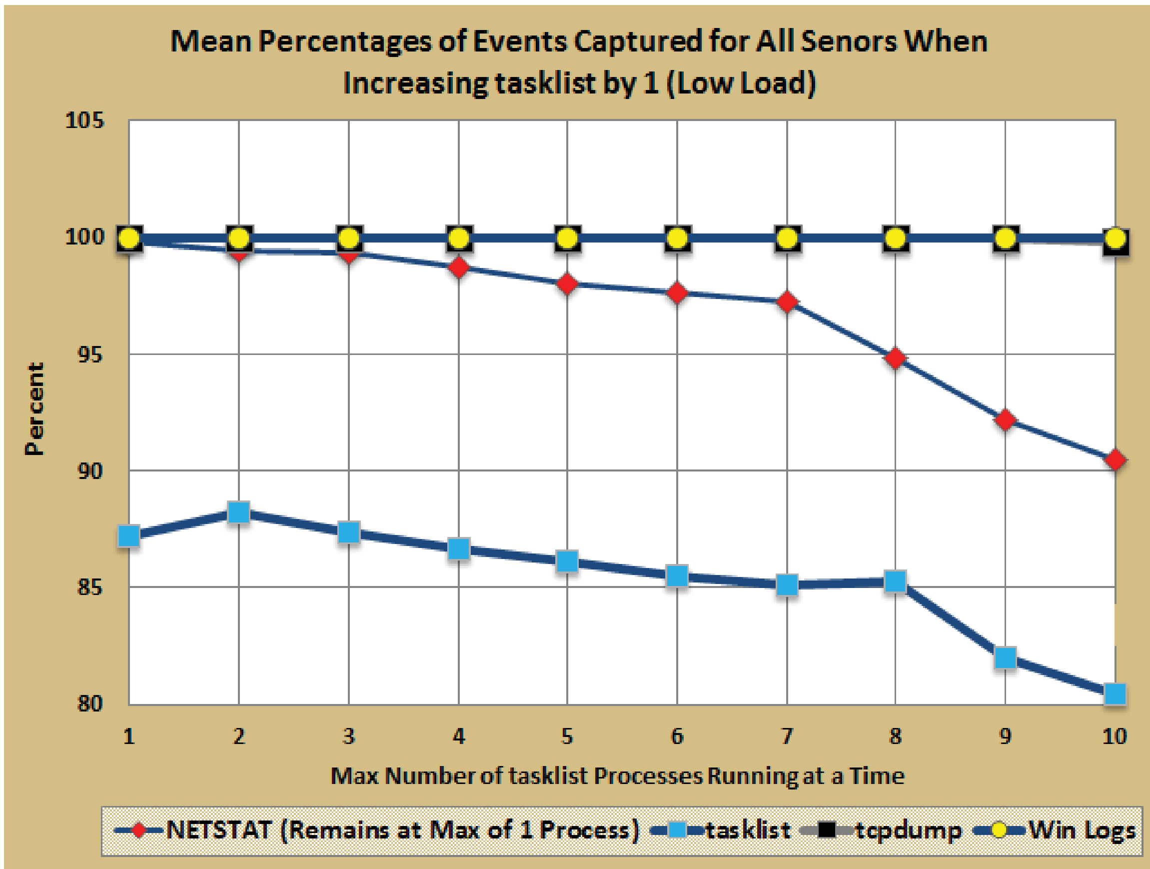


Figure B.4: Mean percentages of events captured for all sensors when increasing `tasklist` by 1 (low load)

Table B.13: Complete performance summary of tasklist under a low load (1 of 4)

| Max tasklist Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 1 | NETSTAT | 99.83529 | 0.40141 | [99.72513 , 99.94546] |
| | tasklist | 87.19216 | 5.23098 | [85.75652 , 88.62780] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 96.75686 | 1.32034 | [96.39450 , 97.11923] |
| 2 | NETSTAT | 99.41176 | 0.79741 | [99.19292 , 99.63061] |
| | tasklist | 88.19608 | 4.04564 | [87.08575 , 89.30640] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 96.90196 | 1.09279 | [96.60204 , 97.20188] |
| 3 | NETSTAT | 99.38039 | 0.73892 | [99.17759 , 99.58319] |
| | tasklist | 87.35686 | 4.34885 | [86.16332 , 88.55040] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 96.68431 | 1.13919 | [96.37166 , 96.99696] |

Table B.14: Complete performance summary of `tasklist` under a low load (2 of 4)

| Max `tasklist` Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 4 | `NETSTAT` | 98.73725 | 1.07740 | [98.44156 , 99.03295] |
| | `tasklist` | 86.69020 | 5.23011 | [85.25479 , 88.12560] |
| | `tcpdump` | 99.96078 | 0.18339 | [99.91045 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 96.34706 | 1.45456 | [95.94786 , 96.74626] |
| 5 | `NETSTAT` | 98.03137 | 1.47262 | [97.62721 , 98.43553] |
| | `tasklist` | 86.12549 | 8.62313 | [83.75887 , 88.49211] |
| | `tcpdump` | 99.96078 | 0.28006 | [99.88392 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 96.02941 | 2.21768 | [95.42077 , 96.63805] |
| 6 | `NETSTAT` | 97.60784 | 1.58693 | [97.17231 , 98.04338] |
| | `tasklist` | 85.47451 | 3.55662 | [84.49840 , 86.45062] |
| | `tcpdump` | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 95.77059 | 1.05836 | [95.48012 , 96.06105] |

Table B.15: Complete performance summary of tasklist under a low load (3 of 4)

| Max tasklist Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 7 | NETSTAT | 97.27843 | 1.93828 | [96.74647 , 97.81039] |
| | tasklist | 85.09804 | 4.35162 | [83.90374 , 86.29234] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 95.59412 | 1.35947 | [95.22101 , 95.96722] |
| 8 | NETSTAT | 94.86275 | 4.29502 | [93.68398 , 96.04151] |
| | tasklist | 85.27059 | 4.59336 | [84.00994 , 86.53123] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 95.03333 | 1.72217 | [94.56068 , 95.50598] |
| 9 | NETSTAT | 92.15686 | 5.02288 | [90.77833 , 93.53539] |
| | tasklist | 81.99216 | 4.41560 | [80.78030 , 83.20402] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 93.53725 | 1.87147 | [93.02363 , 94.05088] |

Table B.16: Complete performance summary of `tasklist` under a low load (4 of 4)

| Max `tasklist` Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 10 | `NETSTAT` | 90.46275 | 6.54553 | [88.66632 , 92.25917] |
| | `tasklist` | 80.44706 | 5.21023 | [79.01711 , 81.87701] |
| | `tcpdump` | 99.82745 | 0.95081 | [99.56650 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 92.68431 | 2.19876 | [92.08086 , 93.28776] |

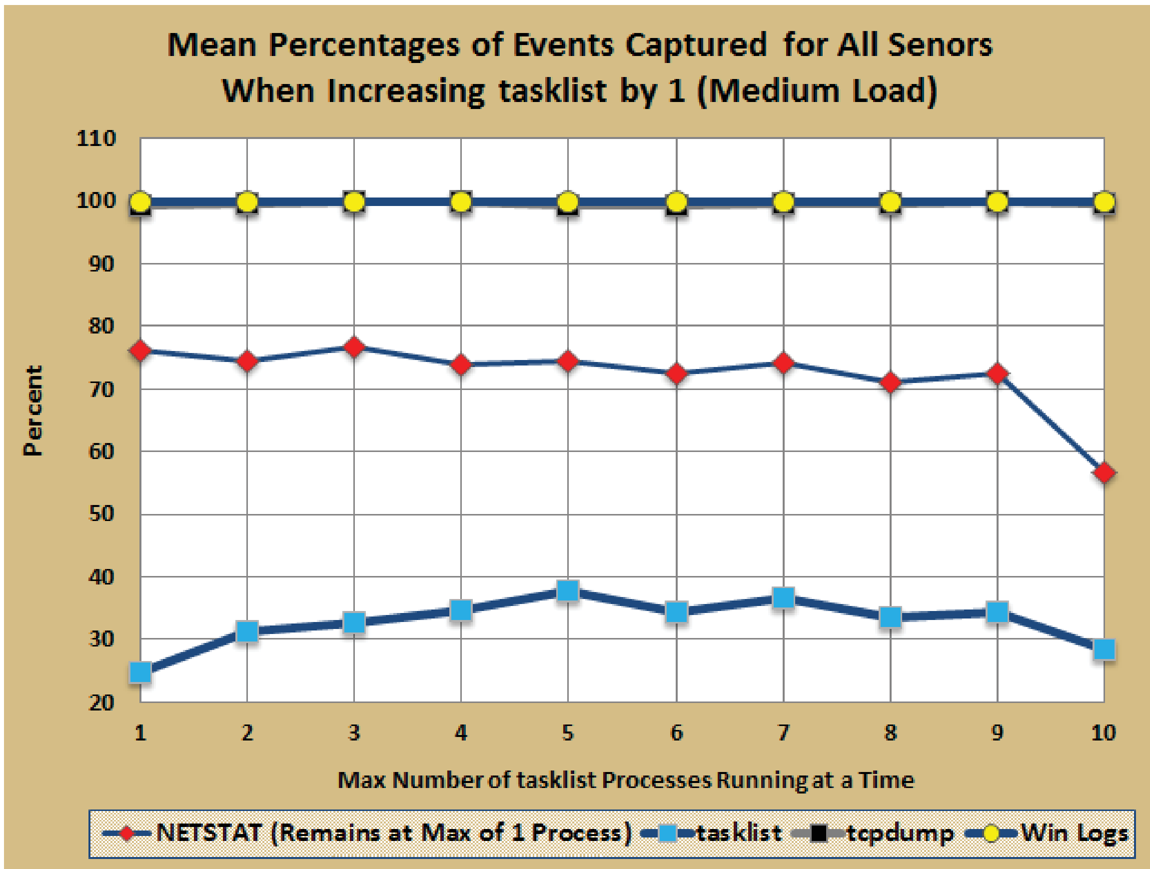


Figure B.5: Mean percentages of events captured for all sensors when increasing `tasklist` by 1 (medium load)

Table B.17: Complete performance summary of `tasklist` under a medium load (1 of 4)

| Max `tasklist` Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 1 | `NETSTAT` | 76.01896 | 7.41148 | [73.98488 , 78.05304] |
| | `tasklist` | 24.87288 | 5.95687 | [23.23802 , 26.50774] |
| | `tcpdump` | 99.26073 | 2.33076 | [98.62105 , 99.90040] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 75.03814 | 2.46021 | [74.36293 , 75.71334] |
| 2 | `NETSTAT` | 74.56969 | 7.67291 | [72.46386 , 76.67552] |
| | `tasklist` | 31.37596 | 6.39101 | [29.62195 , 33.12997] |
| | `tcpdump` | 99.46037 | 1.28919 | [99.10655 , 99.81419] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 76.35153 | 2.83184 | [75.57433 , 77.12873] |
| 3 | `NETSTAT` | 76.58782 | 8.86568 | [74.15464 , 79.02101] |
| | `tasklist` | 32.57414 | 7.83719 | [30.42322 , 34.72505] |
| | `tcpdump` | 99.92916 | 0.31777 | [99.84194 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 77.27278 | 3.54045 | [76.30111 , 78.24446] |

Table B.18: Complete performance summary of `tasklist` under a medium load (2 of 4)

| Max `tasklist` Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 4 | `NETSTAT` | 73.93898 | 8.82234 | [71.51769 , 76.36027] |
| | `tasklist` | 34.59516 | 7.05151 | [32.65987 , 36.53044] |
| | `tcpdump` | 99.76549 | 1.12347 | [99.45715 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 77.07488 | 3.20833 | [76.19436 , 77.95541] |
| 5 | `NETSTAT` | 74.44890 | 8.14271 | [72.21414 , 76.68367] |
| | `tasklist` | 37.62518 | 9.29846 | [35.07321 , 40.17714] |
| | `tcpdump` | 99.37220 | 1.68053 | [98.91097 , 99.83342] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 77.86153 | 3.41433 | [76.92447 , 78.79859] |
| 6 | `NETSTAT` | 72.59655 | 8.87302 | [70.16135 , 75.03175] |
| | `tasklist` | 34.40239 | 6.13960 | [32.71738 , 36.08740] |
| | `tcpdump` | 99.24033 | 1.79112 | [98.74876 , 99.73191] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 76.55982 | 2.52083 | [75.86798 , 77.25167] |

Table B.19: Complete performance summary of `tasklist` under a medium load (3 of 4)

| Max `tasklist` Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 7 | `NETSTAT` | 74.14318 | 11.18834 | [71.07254 , 77.21382] |
| | `tasklist` | 36.51841 | 7.57323 | [34.43994 , 38.59688] |
| | `tcpdump` | 99.45502 | 1.39738 | [99.07151 , 99.83853] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 77.52918 | 3.57456 | [76.54814 , 78.51021] |
| 8 | `NETSTAT` | 71.03612 | 11.53321 | [67.87083 , 74.20141] |
| | `tasklist` | 33.39369 | 6.73710 | [31.54469 , 35.24268] |
| | `tcpdump` | 99.64041 | 0.93122 | [99.38484 , 99.89599] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 76.01759 | 3.08322 | [75.17140 , 76.86378] |
| 9 | `NETSTAT` | 72.45478 | 14.03060 | [68.60409 , 76.30548] |
| | `tasklist` | 34.35955 | 8.21924 | [32.10378 , 36.61532] |
| | `tcpdump` | 99.76249 | 0.85753 | [99.52714 , 99.99784] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 76.64420 | 4.78209 | [75.33175 , 77.95664] |

Table B.20: Complete performance summary of `tasklist` under a medium load (4 of 4)

| Max `tasklist` Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 10 | `NETSTAT` | 56.62700 | 21.83033 | [50.63567 , 62.61833] |
| | `tasklist` | 28.38737 | 9.50078 | [25.77989 , 30.99486] |
| | `tcpdump` | 99.72029 | 0.90750 | [99.47123 , 99.96936] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 71.18365 | 7.05508 | [69.24738 , 73.11991] |

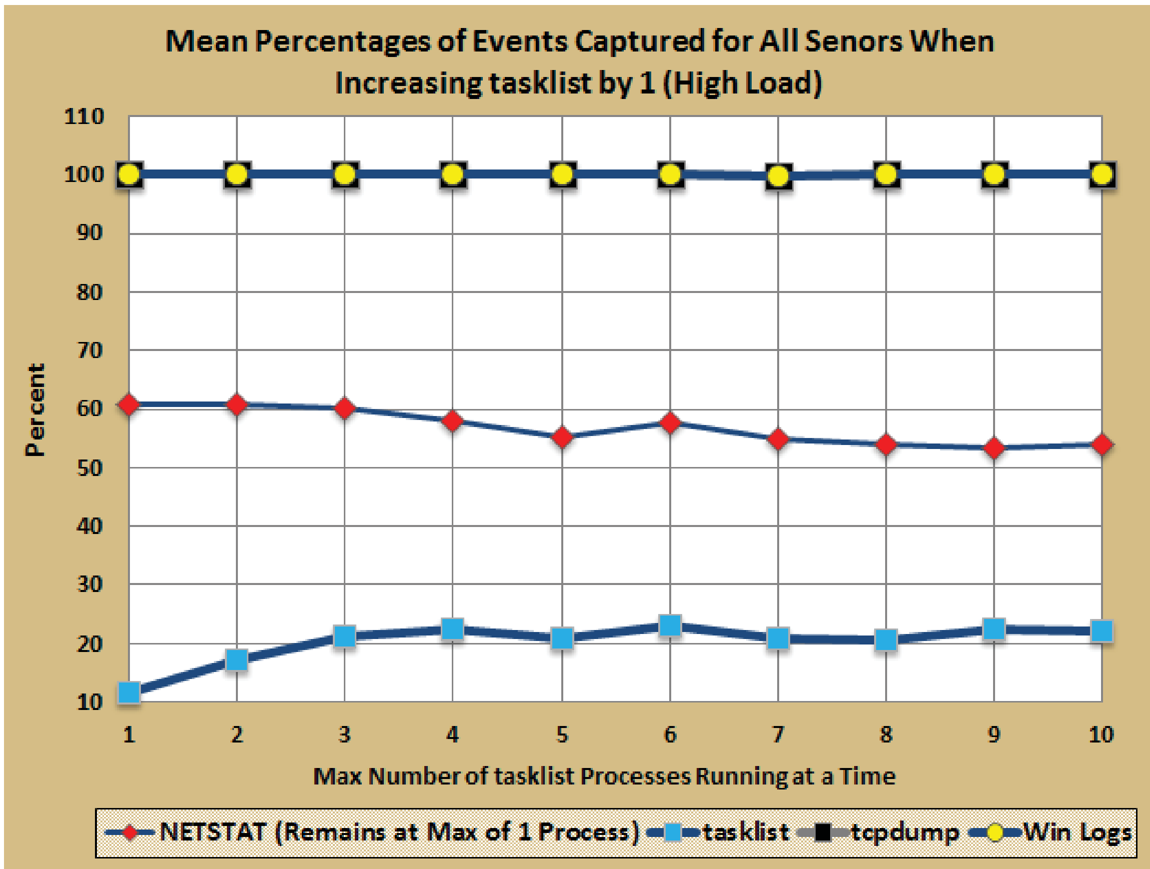


Figure B.6: Mean percentages of events captured for all sensors when increasing `tasklist` by 1 (high load)

Table B.21: Complete performance summary of `tasklist` under a high load (1 of 4)

| Max `tasklist` Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 1 | `NETSTAT` | 60.90980 | 15.68634 | [56.60469 , 65.21492] |
| | `tasklist` | 11.45098 | 3.64940 | [10.44940 , 12.45256] |
| | `tcpdump` | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 68.09020 | 4.42377 | [66.87609 , 69.30430] |
| 2 | `NETSTAT` | 60.87059 | 13.80521 | [57.08175 , 64.65943] |
| | `tasklist` | 17.27843 | 5.24779 | [15.83818 , 18.71869] |
| | `tcpdump` | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 69.53725 | 3.91837 | [68.46186 , 70.61265] |
| 3 | `NETSTAT` | 60.02353 | 15.40730 | [55.79500 , 64.25206] |
| | `tasklist` | 21.12157 | 5.41643 | [19.63503 , 22.60811] |
| | `tcpdump` | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 70.28627 | 4.41769 | [69.07384 , 71.49871] |

Table B.22: Complete performance summary of tasklist under a high load (2 of 4)

| Max tasklist Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 4 | NETSTAT | 57.93725 | 17.44759 | [53.14877 , 62.72574] |
| | tasklist | 22.46275 | 5.79565 | [20.87213, 24.05336] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 70.10000 | 4.91007 | [68.75243 , 71.44757] |
| 5 | NETSTAT | 55.30980 | 17.37218 | [50.54201 , 60.07760] |
| | tasklist | 20.83137 | 5.82345 | [19.23313 , 22.42962] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 69.03529 | 4.88822 | [67.69372 , 70.37687] |
| 6 | NETSTAT | 57.63922 | 18.19279 | [52.64621 , 62.63222] |
| | tasklist | 23.13725 | 6.86714 | [21.25257 , 25.02194] |
| | tcpdump | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 70.19412 | 5.22767 | [68.75938 , 71.62885] |

Table B.23: Complete performance summary of `tasklist` under a high load (3 of 4)

| Max `tasklist` Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 7 | `NETSTAT` | 54.97308 | 21.46372 | [49.08236 , 60.86379] |
| | `tasklist` | 20.98827 | 6.81429 | [19.11809 , 22.85846] |
| | `tcpdump` | 99.92633 | 0.41180 | [99.81331 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 68.97192 | 6.20805 | [67.26812 , 70.67572] |
| 8 | `NETSTAT` | 53.82745 | 21.49504 | [47.92814 , 59.72676] |
| | `tasklist` | 20.61176 | 5.93369 | [18.98326 , 22.24026] |
| | `tcpdump` | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 68.60980 | 6.09003 | [66.93840 , 70.28121] |
| 9 | `NETSTAT` | 53.20784 | 18.79276 | [48.05017 , 58.36551] |
| | `tasklist` | 22.39216 | 5.94615 | [20.76024 , 24.02408] |
| | `tcpdump` | 99.94510 | 0.39208 | [99.83749 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 68.88627 | 5.25110 | [67.44511 , 70.32744] |

Table B.24: Complete performance summary of `tasklist` under a high load (4 of 4)

| Max `tasklist` Processes | Item | Average Percentage (s) of Events Captured | Standard Deviation (s) | Confidence Intervals |
|---|---|---|---|---|
| 10 | `NETSTAT` | 53.90588 | 21.29317 | [48.06198 , 59.74979] |
| | `tasklist` | 22.10196 | 6.40991 | [20.34276 , 23.86116] |
| | `tcpdump` | 100 | 0 | [100 , 100] |
| | Win Logs | 100 | 0 | [100 , 100] |
| | Total (Based on Averages) | 69.00196 | 6.05004 | [67.34153 , 70.66239] |

## Bibliography


[1] P. Bradford and D. Ray, "Using digital chains of custody on constrained devices to verify evidence," in *Intelligence and Security Informatics, 2007 IEEE*, pp. 8–15, 2007.

[2] C. Grobler, C. Louwrens, and S. Von Solms, "A multi-component view of digital forensics," in *ARES '10 International Conference on Availability, Reliability, and Security, 2010.*, pp. 647–652, 2010.

[3] J. Cosic and M. Baca, "(im)proving chain of custody and digital evidence integrity with time stamp," in *MIPRO, 2010 Proceedings of the 33rd International Convention*, pp. 1226–1230, 2010.

[4] Python Software Foundation, "Python 3.3.2." http://www.python.org/download/releases/3.3.2/, May 2013 [Nov. 5, 2013].

[5] H. G. Deepak Meena, "Digital crime investigation using various logs and fuzzy rules: A review," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 2, pp. 1785–1788, 2013.

[6] O. Khaled and H. Hosny, "Making efficient logging a common practice in software development," in *The 3rd ACS/IEEE International Conference on Computer Systems and Applications, 2005.*, pp. 969–972, 2005.

[7] A. Chuvakin and G. Peterson, "How to do application logging right," *Security Privacy, IEEE*, vol. 8, no. 4, pp. 82–85, 2010.

[8] T. Dhaliwal, F. Khomh, and Y. Zou, "Classifying field crash reports for fixing bugs: A case study of Mozilla Firefox," in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pp. 333–342, 2011.

[9] W. Shang, "Bridging the divide between software developers and operators using logs," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1583–1586, 2012.

[10] Mozilla, "Get to know Mozilla." http://www.mozilla.org/en-US/about/, 2013 [Feb. 4, 2014].

[11] K. Kowalski and M. Beheshti, "Analysis of log files intersections for security enhancement," in *Third International Conference on Information Technology: New Generations, 2006*, pp. 452–457, 2006.

[12] M. Garuba, C. Liu, and D. Fraites, "Intrusion techniques: Comparative study of network intrusion detection systems," in *Fifth International Conference on Information Technology: New Generations, 2008*, pp. 592–598, 2008.

[13] S. Vasanthi and S. Chandrasekar, "A study on network intrusion detection and prevention system current status and challenging issues," in *3rd International Conference on Advances in Recent Technologies in Communication and Computing (ARTCom 2011)*, pp. 181–183, 2011.

[14] S. Bosworth and M. E. Kabay, *Computer Security Handbook*. New York, NY and Canada: John Wiley and Sons, Inc., 2002.

[15] Microsoft Corporation, "Microsoft management console - overview." http://technet.microsoft.com/en-us/library/bb742441.aspx, 2013 [Feb. 4, 2014].

[16] Microsoft Corporation, "Event logs." http://technet.microsoft.com/en-us/library/cc722404.aspx, 2013 [Feb. 4, 2014].

[17] J.-h. Huang, M.-q. Zhang, and Y.-l. Jiang, "The design and implement of the centralized log gathering and analysis system," in *2012 IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, vol. 2, pp. 268–273, 2012.

[18] J. Myers, M. Grimaila, and R. Mills, "Log-based distributed security event detection using simple event correlator," in *44th Hawaii International Conference on System Sciences (HICSS), 2011*, pp. 1–7, 2011.

[19] Verizon Communications Incorporated, "The RISK Team." http://www.verizonenterprise.com/products/security/risk-team/, 2013 [Feb. 3, 2014].

[20] Verizon Business RISK Team, "2008 data breach investigations report." http://www.verizonenterprise.com/resources/security/databreachreport.pdf?r=95, 2008 [Feb. 3, 2014].

[21] Verizon Business RISK Team, "2009 data breach investigations report." http://www.verizonenterprise.com/resources/security/reports/2009_databreach_rp.pdf, 2009 [Feb. 3, 2014].

[22] Verizon Business RISK Team, "2010 data breach investigations report." http://www.verizonenterprise.com/resources/reports/rp_2010-data-breach-report_en_xg.pdf, 2010 [Feb. 3, 2014].

[23] Verizon Business RISK Team, "2011 data breach investigations report." http://www.verizonenterprise.com/resources/reports/rp_data-breach-investigations-report-2011_en_xg.pdf, 2011 [Feb. 3, 2014].

[24] Verizon Business RISK Team, "2012 data breach investigations report." http://www.verizonenterprise.com/resources/reports/rp_data-breach-investigations-report-2012-ebk_en_xg.pdf, 2012 [Feb. 3, 2014].

[25] National Institute of Justice, "Digital evidence and forensics." http://www.nij.gov/topics/forensics/evidence/digital/, Nov. 2010 [Feb. 4, 2014].

[26] National Institute of Justice, "Electronic crime scene investigation: A guide for first responders." http://www.nij.gov/nij/publications/ecrime-guide-219941/ch1-electronic-devices/computer-networks.htm, Apr. 2008 [Feb. 4, 2014].

[27] A. Jansen, "Digital records forensics: Ensuring authenticity and trustworthiness of evidence over time," in *Fifth IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE), 2010*, pp. 84–88, 2010.

[28] B. Inglot, L. Liu, and N. Antonopoulos, "A framework for enhanced timeline analysis in digital forensics," in *2012 IEEE International Conference on Green Computing and Communications (GreenCom)*, pp. 253–256, 2012.

[29] E. Skoudis and T. Liston, *Counter Hack Reloaded*. Boston, MA: Pearson Education, Inc., 2006.

[30] P. Matulis, "Centralised logging with rsyslog." http://insights.ubuntu.com/wp-content/uploads/Whitepaper-CentralisedLogging-v11.pdf, 2009 [Feb. 4, 2014].

[31] R. Gerhards, "The syslog protocol." http://tools.ietf.org/html/rfc5424, Mar. 2009 [Feb. 4, 2014].

[32] D. Edwards, "Computer forensic timeline analysis with tapestry," 2011 [Feb. 3, 2014].

[33] F. Wiki, "Timestomp." http://www.forensicswiki.org/wiki/Timestomp, Nov. 2010 [Feb. 4, 2014].

[34] S. Azadegan, W. Yu, H. Liu, M. Sistani, and S. Acharya, "Novel anti-forensics approaches for smart phones," in *45th Hawaii International Conference on System Science (HICSS), 2012*, pp. 5424–5431, 2012.

[35] Y. Zhou and K. Jiang, "An analysis system for computer forensic education, training, and awareness," in *2012 International Conference on Computing, Measurement, Control and Sensor Network (CMCSN)*, pp. 48–51, 2012.

[36] S. Garfinkel, "Anti-forensics: Techniques, detection and countermeasures," in *2nd International Conference on i-Warfare and Security*, pp. 77–84, 2007.

[37] M. Russinovich, D. A. Solomon, and A. Ionescu, *Windows Internals Part 1*. Redmond, Washington: Microsoft Press, 6th ed., 2012.

[38] Microsoft Corporation, "Scheduling priorities." http://msdn.microsoft.com/en-us/library/windows/desktop/ms685100(v=vs.85).aspx, Nov. 2013 [Nov. 20, 2013].

[39] Microsoft Corporation, "Context switches." http://msdn.microsoft.com/en-us/library/windows/desktop/ms682105(v=vs.85).aspx, Nov. 2013 [Nov. 20, 2013].

[40] H. Niksic and G. Scrivano, "Wget." http://www.gnu.org/software/wget/, Sept. 2012 [Feb. 2, 2014].

[41] Microsoft Corporation, "Netstat." http://technet.microsoft.com/en-us/library/ff961504.aspx, Apr. 2012 [Nov. 5, 2013].

[42] Microsoft Corporation, "Tasklist." http://technet.microsoft.com/en-us/library/cc730909.aspx, Apr. 2012 [Nov. 5, 2013].

[43] Tcpdump/Libpcap, "Tcpdump." http://www.tcpdump.org/, 2014 [Feb. 4, 2014].

[44] G. Lyon, "nmap." http://www.nmap.org/, Nov. 2013 [Feb. 3, 2014].

## Vita

Captain Daniel M. Gallagher graduated from Norwich University with a B.S. in Computer Security and Information Assurance in 2009. The day following graduation, Captain Gallagher commissioned as a Second Lieutenant in the United States Air Force. His first assignment was to Pope Air Force Base (AFB), now Pope Field, located in North Carolina where he worked as a Communications Squadron's Base Realignment and Closure (BRAC) Project Manager. Next, he was assigned to Wright-Patterson AFB and worked as an Enterprise Services Flight Commander in the 83 Network Operations Squadron (Detachment 3) for ~1 year, though ~6 of those months were spent TDY to Scott AFB working as part of the Air Force Network (AFNet) Migration Team. The assignment as part of Detachment 3 was cut short after being selected to attend the Air Force Institute of Technology (AFIT) in May of 2012. He is currently pursuing a Master of Science degree with a major in Cyber Operations.

# REPORT DOCUMENTATION PAGE

*Form Approved*
*OMB No. 0704–0188*

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704–0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202–4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**

| 1. REPORT DATE *(DD–MM–YYYY)* | 2. REPORT TYPE | 3. DATES COVERED *(From — To)* |
|---|---|---|
| 27–03–2014 | Master's Thesis | June 2012 - March 2014 |

**4. TITLE AND SUBTITLE**
Analysis of Effects of Sensor Multithreading to Generate Local System Event Timelines

**5a. CONTRACT NUMBER**

**5b. GRANT NUMBER**

**5c. PROGRAM ELEMENT NUMBER**

**6. AUTHOR(S)**
Gallagher, Daniel M., Captain, USAF

**5d. PROJECT NUMBER**

**5e. TASK NUMBER**

**5f. WORK UNIT NUMBER**

**7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)**
Air Force Institute of Technology
Graduate School of Engineering and Management (AFIT/EN)
2950 Hobson Way
WPAFB, OH 45433-7765

**8. PERFORMING ORGANIZATION REPORT NUMBER**
AFIT-ENG-14-M-31

**9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES)**
Center for Cyberspace Research
Attn: Dr. Robert Mills
2950 Hobson Way
WPAFB, OH 45433-7765
(937) 255-3636, Ext. 4738, Robert.Mills@afit.edu

**10. SPONSOR/MONITOR'S ACRONYM(S)**
AFIT/CCR

**11. SPONSOR/MONITOR'S REPORT NUMBER(S)**

**12. DISTRIBUTION / AVAILABILITY STATEMENT**
DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

**13. SUPPLEMENTARY NOTES**
This work is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

**14. ABSTRACT**
In practice, organizations with their own information technology infrastructure normally log or otherwise monitor network information at boundary routers and similar network devices that are log-capable. However, not all organizations opt to log local system information, such as an employee's organization-owned workstation activity. This research explores one approach to logging pertinent local system information using multithreading and free software designed for such logging purposes as well as utilities that come with the Microsoft Windows® 7 Operating System.
Research focuses on file downloads on the local system and combines the aforementioned pieces of software into an event logging suite. The event logging suite consists of four different sensors and utilizes multithreading in an attempt to effectively capture as many pertinent events as possible, with the ultimate goal of capturing 100% of the events in chronological order of actual occurrence. Specifically, the event logging suite increases the number of processes and thus threads that two of the four sensors, Windows® `NETSTAT` and `tasklist` utilities respectively, in the suite execute in order to determine the optimal settings for the two sensors. To add some realism to the experiments, this research implements three different system loads to simulate user activity on the system while a scripted file-download scenario executes and the logging suite actively captures events.
Ultimately, the performance accuracies of the `NETSTAT` and `tasklist` sensors across numerous tests show that while the sensors can capture above 85% of the expected number of events, neither are capable of consistently achieving this accuracy, even under a low system load.

**15. SUBJECT TERMS**
event logging, log files, event timeline, multithreading, Windows® 7

| 16. SECURITY CLASSIFICATION OF: a. REPORT | b. ABSTRACT | c. THIS PAGE | 17. LIMITATION OF ABSTRACT | 18. NUMBER OF PAGES | 19a. NAME OF RESPONSIBLE PERSON |
|---|---|---|---|---|---|
| U | U | U | UU | 113 | Maj Thomas E. Dube (AFIT/ENG) |

**19b. TELEPHONE NUMBER** *(include area code)*
(937) 255-3636 x4613 Thomas.Dube@afit.edu