An Intrusion Detection System for Battery Exhaustion Attacks on Mobile Computers

Daniel C. Nash

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Dr. Thomas L. Martin
Dr. Michael Hsiao
Dr. Dong Sam Ha

May 18, 2005
Blacksburg, Virginia

Keywords: IDS, intrusion detection, power attacks, linear regression model

An Intrusion Detection System for Battery Exhaustion Attacks on Mobile Computers

Daniel C. Nash

(ABSTRACT)

Mobile personal computing devices continue to proliferate and individuals' reliance on them for day-to-day needs necessitate that these platforms be secure. Mobile computers are subject to a unique form of denial of service attack known as a battery exhaustion attack, in which an attacker attempts to rapidly drain the battery of the device. Battery exhaustion attacks greatly reduce the utility of the mobile devices by decreasing battery life. If steps are not taken to thwart these attacks, they have the potential to become as widespread as the attacks that are currently mounted against desktop systems.

This thesis presents steps in the design of an intrusion detection system for detecting these attacks, a system that takes into account the performance, energy, and memory constraints of mobile computing devices. This intrusion detection system uses several parameters, such as CPU load and disk accesses, to estimate the power consumption of two test systems using multiple linear regression models, allowing us to find the energy used on a per process basis, and thus identifying processes that are potentially battery exhaustion attacks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A key element in a successful pervasive computing environment is a personal computing device that enables the user to have continuous access to information, allows the user to go about a normal daily routine with minimal distraction, and determines user intent in order to proactively adapt itself to the users needs [20]. Users' reliance on these devices necessitates that they be secure. One security attack that is unique to these battery powered devices is a denial of service attack aimed at draining the battery. These "sleep deprivation torture" or "battery exhaustion" attacks, as called by Stajano and Anderson, prevent devices from entering their normal low power idle or sleep states [22]. Consequently, the expected battery lifetime of the devices is significantly reduced and users fail to gain the full utility of the devices.

Battery exhaustion attacks are no longer theoretical. Martin et.al. identified and implemented three different classes of these attacks [14]: (1) malignant attacks, in which a virus or Trojan horse is used to make the device consume significant power, (2) benign attacks, in which an unmodified program is given pathological data such that the program consumes excessive energy, and (3) service request attacks, a special form of the benign attack in which repeated requests are made to a network service provided by the device. The malignant attacks can be found using currently available virus scanning techniques, but the benign and

service attacks cannot be detected by them because they work on unmodified code. In addition to these proof-of-concept implementations, there is already a virus "in the wild" that has the properties of a battery exhaustion attack, although its excessive power consumption appears to have been a side effect rather than the main intent. The Cabir virus was a proof-of-concept to illustrate a vulnerability in mobile devices running Symbian OS Series 60 [6]. It transmits itself using the Bluetooth communication protocol between devices. While the goal of this virus creator does not appear to be the creation of a power attack, the operation of the virus causes one to occur. The virus causes the Bluetooth radio on the mobile device to broadcast at frequent intervals, seriously reducing the battery life of the device.

To combat these new attacks, a new line of defense must be developed and put into place. The goal of an intrusion detection system that detects these attacks would be to allow the computing device on which it operates to achieve a guaranteed percentage of its battery life. To do so, it must be to identify attacks that cause the computing device to consume too much energy. Unfortunately, most mobile devices lack the means to accurately measure their own power usage directly. Another method of estimating the power must be used.

This thesis discusses the design and implementation of an intrusion detection system designed for detecting this new form of attack, subject to the performance, memory, and energy limitations of typical pervasive computing devices. Chapter 2 discusses related work in the fields of intrusion detection and battery exhaustion attacks. Chapter 3 introduces a new form of power attack and its impact on previously established categories of power attacks. Chapter 4 discusses the multiple linear regression models used to estimate system power and their effectiveness. Chapter 5 discusses the implementation of intrusion detection systems that use the regression models to determine attacking processes. Chapter 6 draws conclusions about the developed systems and discusses possible future work.

Chapter 2

Background

The three areas of research related to this thesis are intrusion detection, battery exhaustion attacks, and automated system response.

2.1 Intrusion Detection

The problem of intrusion detection has been studied for several years with early papers on the subject appearing in the late 1970s and early 1980s [9]. While the definition of an intrusion varies slightly from paper to paper, definitions such as the following are widely accepted: "any set of actions that attempt to compromise the integrity, confidentiality or availability of a resource [12]". An intrusion detection system then is a system which attempts to detect and in some cases react to intrusions, whether on one system, group of systems, or computer network.

Early research in this topic developed two classification systems for different intrusion detection systems, anomaly detection and signature or misuse detection [3]. Anomaly detection focuses on determining abnormal usage from a previously established pattern of normal behavior. In [4], a network intrusion detection system named LISYS uses a form of anom-

aly detection that takes an immunological approach to intrusion detection. It emulates a biological immune system by generating detector "cells" that will only match to anomalous connection information. A tolerization period is first run during which time the LISYS generates random detector "cells" that are destroyed if they match with any connection information. After the tolerization period, the remaining random detector "cells" become the primary detectors for anomalous behavior. This system, like many anomaly detection systems, relies on a training period during which time it is assumed no attacks occur. The drawback to this is that it may be infeasible for some computing devices to be trained in a realistic manner without being deployed in an environment under which they could already be under attack.

Signature detection attempts to match system usage to established rules, signatures, or model of intrusive or illegal behavior. In [3], one of the systems surveyed, MIDAS, uses heuristic rules to match to the characteristics of known bad behavior. Audit logs are captured and compared against these heuristics to determine if intrusions have occurred. The flaw of many signature detection systems is their inability to detect and react to new forms of attack for which they have no related signature. Including elements of anomaly detection system can improve this flaw. Additionally, the number of signatures required can become quite large in terms of space required and also difficulty in managing them.

In [13], several areas of intrusion detection system research were identified including foundations, data collection, detection methods, response, intrusion detection system environment and architecture, intrusion detection system security, testing and evaluation, operational aspects, and social aspects. Of particular interest to the problem of battery exhaustion attacks against mobile devices are the areas of data collection and detection method. Some research in the implementation of intrusion detection systems in a constrained environment appears to have been done, but not in these two areas in particular [13]. Methods used in commercial intrusion detection systems may be difficult if not impossible to implement. For instance, the extensive audit data collected and analyzed by an intrusion detection system may simply not be present or too consuming on device resources to collect. Also, extensive

analysis of this data may consume too much power or make a system too unresponsive for a user for it to be worthwhile.

2.2 Battery Exhaustion Attacks

A new area of research in the field of mobile computing devices that has received very little attention as of yet is that of battery exhaustion attacks against mobile computing devices [22][11]. As mentioned before, these attacks are no longer theoretical with the Cabir virus being an excellent example. Stajano and Anderson investigated security issues for ad-hoc wireless sensor networks, including the area of power security. In such a network, the power consumption of the wireless radio is the dominating factor of the battery life of the sensor nodes. The nodes normally enter a low power sleep state to conserve energy and extend their battery life. Sleep deprivation torture attacks are a form of battery exhaustion attack that prevents the sensor nodes from entering their low power states, thereby draining their batteries at much higher rates than would normally be seen under normal use. The primary defense of the sensor nodes is to prioritize their wireless communication to reduce the effectiveness of these attacks.

For general purpose computing devices, the manner in which they can be attacked is broader and can be harder to defend against. Three categories of attacks were developed in [14]: malignant attacks, benign attacks, and service attacks. Malignant attacks are programs created or modified for the purpose of consuming more power on a computing device, thereby draining its battery more quickly. In [14], this attack took the form of a program, named cachesize, that manipulated its data in such a manner as to eventually cause the system to experience repeated cache misses. On several of the systems it was run, this behavior caused an increase in the systems' power consumption above and beyond what was observed when cache misses were not occurring. On other of the test systems, a power decrease was actually observed for cache misses, with cache hits requiring more power. More importantly, it was

shown that a cleverly written program can tune itself to cause the greatest possible power consumption by the system.

The next category, benign attacks, are attacks that give an unmodified, normally benign program pathological data such that the program consumes excessive energy. In [14], an animated GIF with the same image in a continuous loop was used to cause a web browser to atypically consume more processor cycles, increasing the system's power consumption. This form of attack is especially insidious as, without visual clues to indicate anything abnormal taking place, a user would not know he was under attack until the long term effects of elevated power consumption are seen in a depleted battery.

The last category of attacks, network service attacks, are identified as a special form of a benign attack in which repeated network service requests are made to services offered by the device under attack. In [14], repeated login requests were made to secure shell server (SSH). Unlike a typical networking attack, the goal of the requests was not to successfully log into the server, but to cause it to perform calculations to determine if the login was successful. The repeated requests caused spikes of increased power consumption on the target device.

While an informal topology of these categories was implied, a formal one has not been given [11]. In Chapter 3 a formal taxonomy is developed. Chapter 3 also discusses is a new form of attack that does not clearly fit into the above categories. Its place in the taxonomy is analyzed and a new taxonomy proposed.

2.3 IDS Reaction Mechanisms

Automated intrusion response is a difficult security problem and a field of active research. The possibility of false positives makes it difficult to take an active response when intrusions are detected. Anil Somayaji and Stephanie Forrest presented a unique form of automated intrusion response using system-call delays [21]. Traditional responses employed by commercial IDSs include terminating network connections and processes. Somayaji and Forrest

recognized that these responses are not widely employed because of the risk of responding when an intrusion was not taking place. The intrusion detection system described in [21], pH, which stands for process homeostasis, monitors system-calls by processes running on a computer to determine anomalous behavior. When it believes such behavior has occurred, pH is able to delay or abort system calls to correct the state of the system. This response behaves much differently from the all-or-nothing responses mentioned before. In the case of false positives, the effect of this response is minimal. By also generating the delay based on a function of recent anomalous behavior, pH can increase the severity of its response as warranted by prolonged intrusions. Using a similar mechanism in a battery exhaustion IDS would work well in reducing or eliminating the effectiveness of battery exhaustion attacks while minimizing the impact of a false positive for legitimate processes.

Chapter 3

Taxonomy of Battery Exhaustion Attacks

This chapter first discusses a taxonomy for previously identified categories of battery exhaustion attacks. It then describes a new attack discovered during the creation of the power intrusion detection system. Its position within the existing taxonomy of battery exhaustion attacks is discussed and a revision of the taxonomy is suggested. The inability of the power IDS to properly identify the attacking process is analyzed and possible measures needed to correctly handle these attacks are discussed.

3.1 Taxonomy of Existing Attacks

In computer networking, some forms of attack occur as distributed attacks in which multiple attacking computers work against a common target. The same can be done for the malignant battery exhaustion attacks identified by Martin et. al. and discussed in Chapter 2. Multiple programs, each causing small power increases on the device, can operate together to cause a much larger effect.

There are two primary scenarios for such an attack with other variants using combinations of the two. The first scenario has a controlling program that generates child programs which operate for short intervals and then die or go to sleep. The controlling program could also contribute directly to the attack, but does not have to. The second scenario has one program that attacks for a short while before spawning a child process and dieing or going to sleep. The child process then does the same thing. Again, the effect of the individual processes is minimal while their overall effect is large. Both variations of this distributed malignant attack were tested and the results are shown in Chapter 5.

A formal taxonomy of all the categories identified in their work was generated and is shown in Figure 3.1. The two main categories of attack identified were malignant and benign attacks. Network service attacks were described as special form of benign attack and so has been made a descendant of benign attacks. Most malignant attacks could conceivably be transformed into a distributed attack. Therefore, the malignant attack category also encompasses this form of attack.

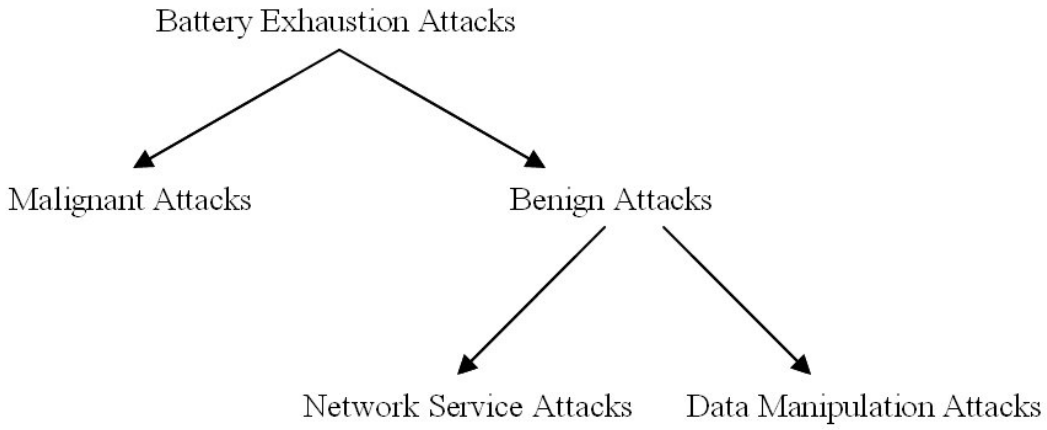

Figure 3.1: This figure shows a formal taxonomy for battery exhaustion attacks.

3.2 Development of a New Attack

All of the attacks previously developed and discussed in Chapter 2 operated with the underlying assumption that it would be the attacking program itself that consumed the most energy, thereby draining the battery of the device more quickly. In the case of benign attacks, the assumption still applies as the benign program is turned into an attacking program through its processing of cleverly manipulated data. What I have since discovered is that this assumption is not always true.

During the course of developing the intrusion detection system, an attack was discovered that raised the power usage of the system significantly, but the attacking program managed to do so without itself drawing significant power. This occurred when one program's window was dragged across the screen to reposition it for better viewing. While the first window was being repositioned, it crossed over another program's window. During that time period, the power usage of the second program was observed to have risen considerably. This occurred because the second program had to perform calculations in order to redraw the revealed portion of its window. It is not clear in that context that there is an "attacking" program, but the observed behavior led to the development of an attack. One possible way of implementing the attack would have been to have an attacking program with its own window move itself back and forth across the screen over another program's window. While effective, this form of the attack would have been readily apparent to a user and also have made finding the attacking program a trivial task. There was a better and must less obvious way to implement the attack.

Normally, most program windows operate independently of all other programs and as far as they are concerned, are the sole windows open and visible. To handle circumstances where one program window obscures the contents of another and later the obscured window becomes visible again, programs respond to events sent by the main window manager alerting them to the fact that part or all of their window has been uncovered or revealed. Under normal circumstances, only the main window manager will send these events to other programs.

What I found, however, is that there usually is nothing preventing any application from sending these same events. Under both Microsoft Windows operating systems and the X Windows environment in Linux, there are facilities that allow events/messages to be sent between processes without intervention of the windowing system.

The better method of implementing the attack, then, was to have the attacking program send events to a victim program indicating to the victim program that its window had been revealed. Figure 3.2 shows an example of invalidating a region of a program's window. The program, in this case Microsoft's Notepad application, receives a message indicating that a portion of its window, encompassed by the rectangle in the figure, has been invalidated. In most cases the program will then respond to this message by performing any necessary calculations to redraw its window's contents. Some programs, with more coding effort on the part of the developer, will work to redraw only those portions of their window marked as invalid while others will more simply redaw their entire window. In this form of the attack, the attacking program does not even need a visible window of its own to operate.

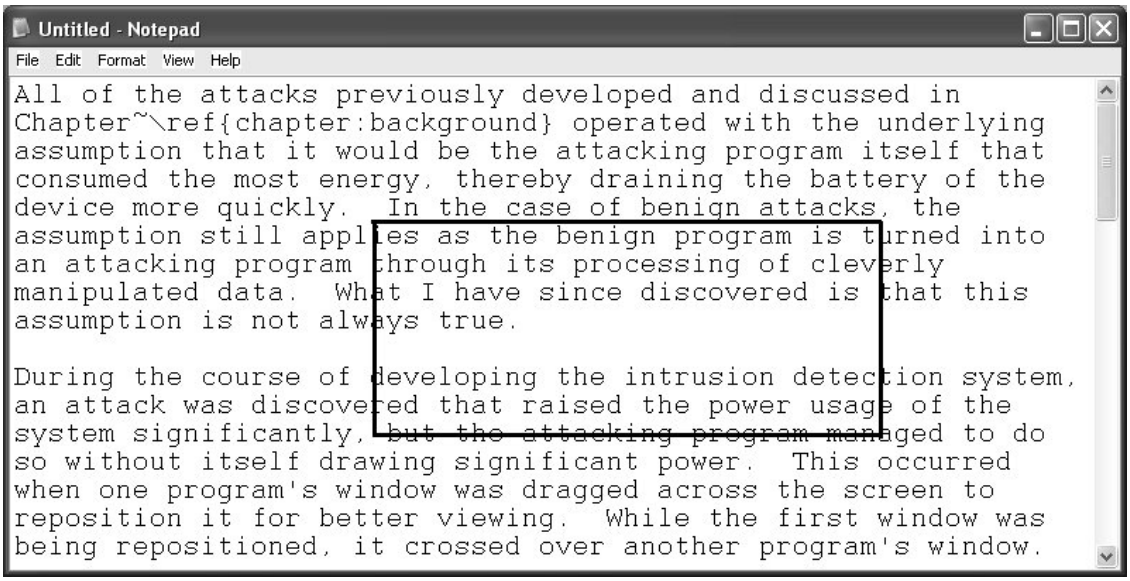

Figure 3.2: The region encompassed by the rectangle has been mark invalid and needs to be redrawn.

The flow of this attack is shown in Figure 3.3. In a Windows operating system en-

vironment, the attacking program sends redraw messages using the PostMessage function specifying the benign program window that is to receive those messages. Those messages will be placed in the message queue of the window by the operating system to be retrieved later in the window's message loop. Upon receiving those messages, the window will perform any necessary calculations to redraw its window. Under the X Windows environment, the attacking program can send expose events to the window manager using the XSendEvent function, specifying the program to receive those events. The window manager will then place those events in the program's event queue to be retrieved and processed by the program. Upon processing the expose events, the program will perform any calculations and redraw its window.

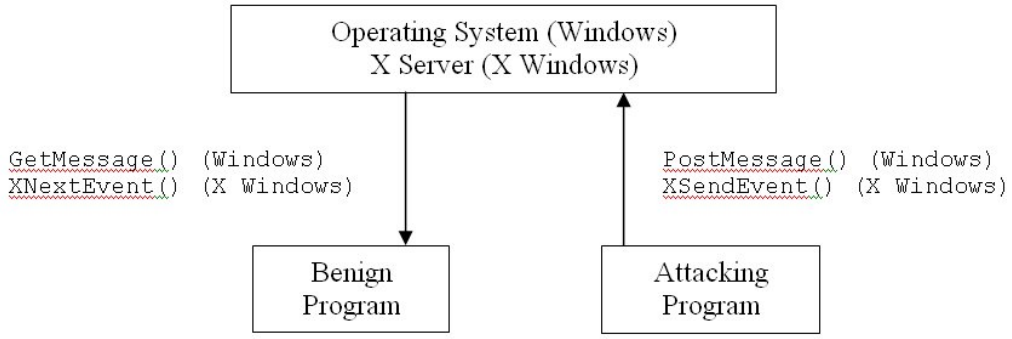

Figure 3.3: The attack sends messages through the window manager to the benign program.

The effect of the redrawing can vary significantly from program to program, depending greatly on the amount of computation and possibly I/O requests that must be performed to successfully redraw the contents of the window. In the case of a program such as Microsoft's Paint program, the image being displayed in its main window is usually readily available in video memory and can be redisplayed with very little computation. On the other hand, a text editing application such as Microsoft's Notepad may have to reread a file from hard disk, or at least memory, and perform calculations regarding text size, character spacing, etc. in order to update its window contents.

Such a scenario with the Notepad program was carried out with the results shown below in Figure 3.4. The dialog shown lists the programs that were running on the system at the time of the attack. The program notepad.exe was previously observed to cause power consumption on the order of less than 1 W. As shown in the figure, at the time of the attack, notepad.exe caused power consumption significantly higher than the power before executing the attack. Additionally, the program causing Notepad to consume the additional power, ServiceAttackApp.exe, consumes very little power. This same attack was used against the Microsoft Paint application and caused almost not additional power consumption for the reasons discussed above, registering only 0.14 W at most.

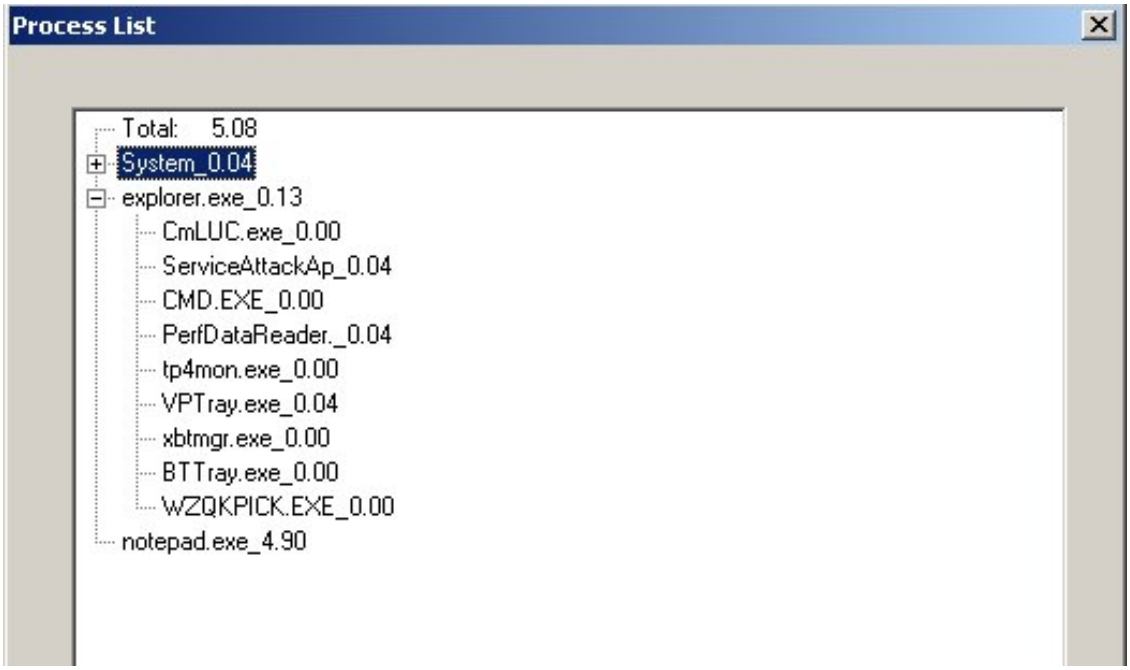

Figure 3.4: Notepad's power usage under attack is significantly higher than normal.

Such an attack has successfully been created both under Microsoft Windows operating systems and a Linux operating system in the X Windows environment. The attack was even less apparent under the X Windows environment, with no visible flashing occurring while the attack was in progress. However, monitoring the processor usage of all processes showed that the attack was indeed successful.

3.3 Attack's Place in Taxonomy

At first glance, the attack appears to fit both within the benign and service branches of the existing attack taxonomy. The attacking program causes a benign program to increase its power usage by sending, indirectly, bogus event data to the benign program. On the other hand, while the current taxonomy's concept of a service attack encompasses only network services, the reception and handling of these events can be viewed as a service of the benign program, only locally instead of remotely.

This idea of a local service attack opens up the possibility of exploiting many kinds of local services. Any database service that is running on the machine can be made to run long, computationally intensive queries and the results ignored. The desktop searches introduced by Google and recently Yahoo and Microsoft have possibility for exploitation. The windowing environment itself may further be vulnerable.

The taxonomy discussed in section 3.1 does not allow for the concept of a non-network service attack and needs modification. Generalizing from network service attacks to generic service attacks, of which network service attacks are a descendant, allows for the taxonomy to encompass this new form of service attack. Figure 3.5 illustrates this modification. Attacks that take advantage of local services on a computing device as well as attacks that exploit network services are descendants of the new service attack category.

This attack is particulary bad because it operates in a manner that is not easily tracked under the operating systems and windowing systems used. Chapter 6 will describe how this attack cannot be caught by the intrusion detection system that was developed without additional resources from the operating system or windowing system on the mobile device it is deployed on.

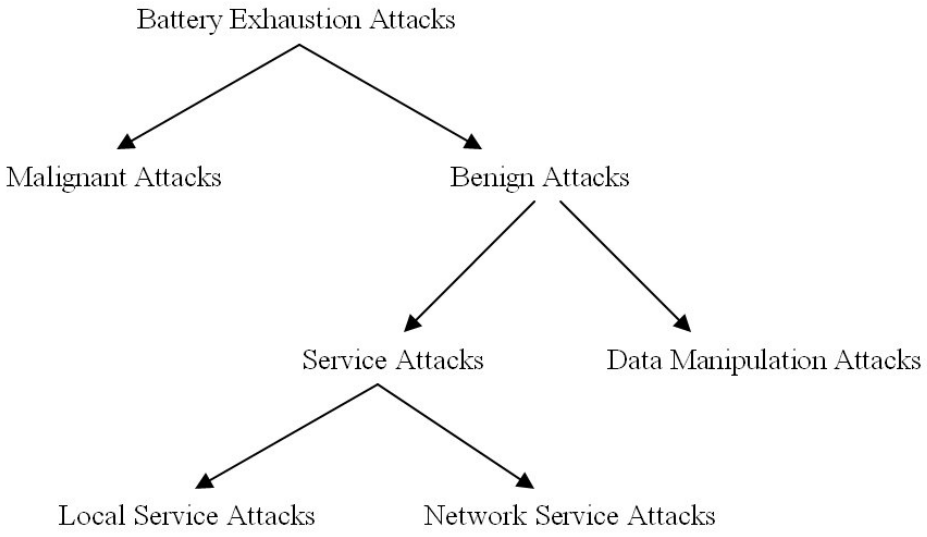

Figure 3.5: The new taxonomy includes a generic service attack category and its descendant local service attack category.

Chapter 4

Linear Regression Model

This chapter focuses on the development of a linear regression model for estimating the power consumption of a system. First, other methods of determining the system power are discussed. Then, a methodology for selecting system parameters to use in the regression model is discussed. A description of the two test systems and power measurement setup used for experimentation is given. Finally, models of two test systems are developed and analyzed.

4.1 Determining System Power

The most straightforward way to detect a battery exhaustion attack would be to measure the power on a process-by-process basis, thus determining which processes were responsible for consuming large amounts of energy. Unfortunately, most, if not all, battery powered devices lack a high fidelity power measurement system. If "smart battery" technology is used, rough measurements of power consumption and remaining battery capacity can be obtained, but these power measurements are too coarse to provide energy consumption on a process-by-process basis. The smart battery chips have low sampling rates, on the order

of 1 Hz. To increase accuracy in estimating the remaining capacity, some of these chips also only report a value for the current power dissipation rate averaged over tens of seconds. The graph of Figure 4.1 shows the recorded power of the Latitude test system, marked as blue circles, and the reported discharge rate of its main battery, marked as green diamonds. Due to the measurement setup, these measurements were taken separately, but the timing of the events they record are the same. The reported power dissipation of the smart battery is both lower and delayed from those recorded with a high fidelity multimeter. This delay makes accurate power measurements on a process-by-process basis difficult if not impossible.

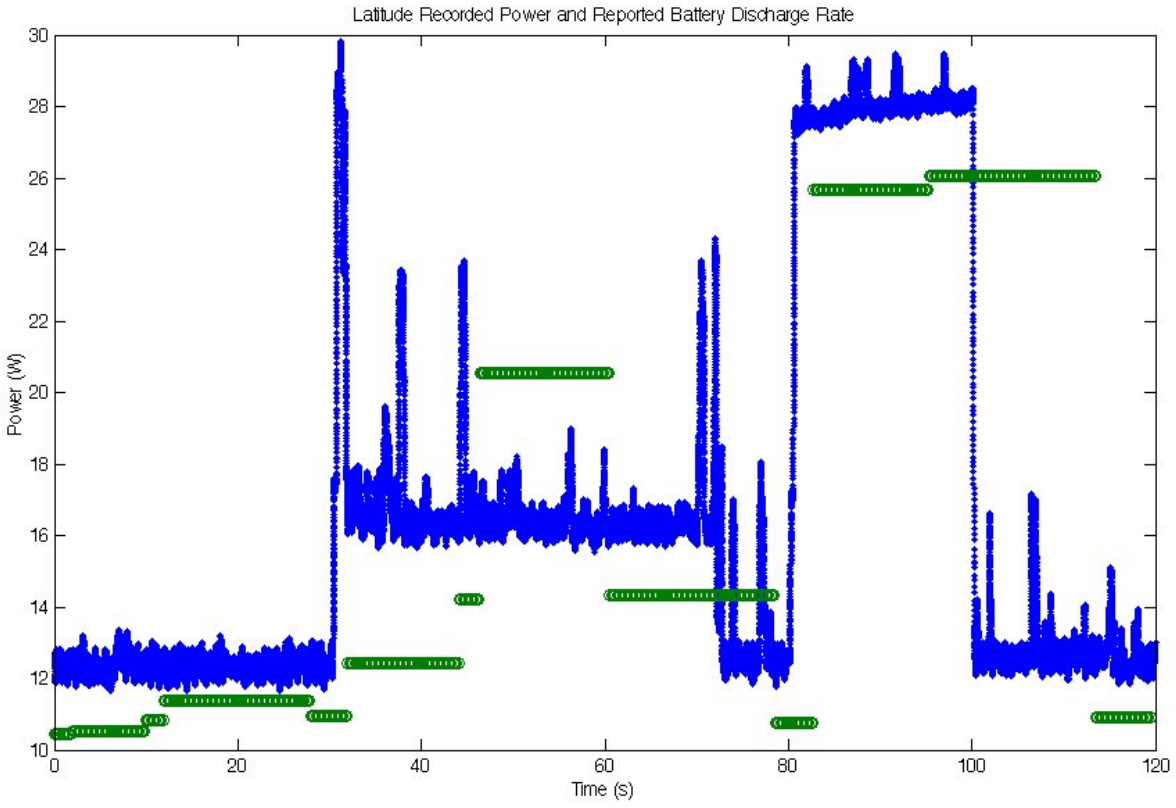

Figure 4.1: The smart battery discharge rate (green circles) is averaged over long intervals and does not follow recorded power (blue diamonds) well.

Additionally, the computing environment found on most mobile computing devices would hinder determining power consumption of individual processes solely on a time division

basis. Programs that make I/O requests are typically removed from the processor while the program's I/O request is being serviced. In the meantime, another program is executed. Measuring the total system power and assigning it to the process that is currently being executed would be insufficient. It would be necessary to have a complex power measurement system that maintains links between the power usage of individual components of the system and those processes that are using those components.

On the other hand, even if the battery operated devices of interest had high frequency power measurement systems available, use of such a system could leave a user susceptible to a cyptographic key cracking technique known as power analysis [10]. For devices that perform encryption, the energy consumption can be correlated to bits of the encryption key, allowing an attacker to very rapidly find large portions of an encryption key. If detecting battery exhaustion attacks requires a high resolution power measurement system, it could make this attack possible without requiring physical access to the device. Thus we desire a method of estimating power on a process-by-process basis that does not have a high enough fidelity to make power analysis attacks feasible.

Barring direct power measurement, it is necessary to measure other indirect indicators of power usage on the system. In the work on battery exhaustion attacks discussed in Chapter 2, several components were identified that caused significant elevated power consumption on mobile systems. Extended processor usage and repeated wireless transmission both caused elevated power levels in the devices tested. It was thought that repeated hard disk access or causing the hard disks to spin down and up repeatedly could also cause elevated power levels.

From that work, it was believed that there would be a high correlation between system parameters such as processor usage and hard disk access rates and the total power usage of the system. While monitoring processor usage and power usage on one test system, the graphs shown in Figure 4.2 were obtained. The top graph shows the measured power of the system in Watts, while the bottom graph, with measurements taken concurrently, shows

the measured processor load as a percentage of time the processor was processing non-idle threads in the measurement interval as reported by the operating system, Windows 2000 Professional. The two graphs show a very high correlation between the processor behavior and the total power usage of the system. Many of the local peaks shown in the power graph have a counterpart in the processor load graph. Additionally, a scattergram of the system power versus processor load shown in Figure 4.3 shows a linear trend.

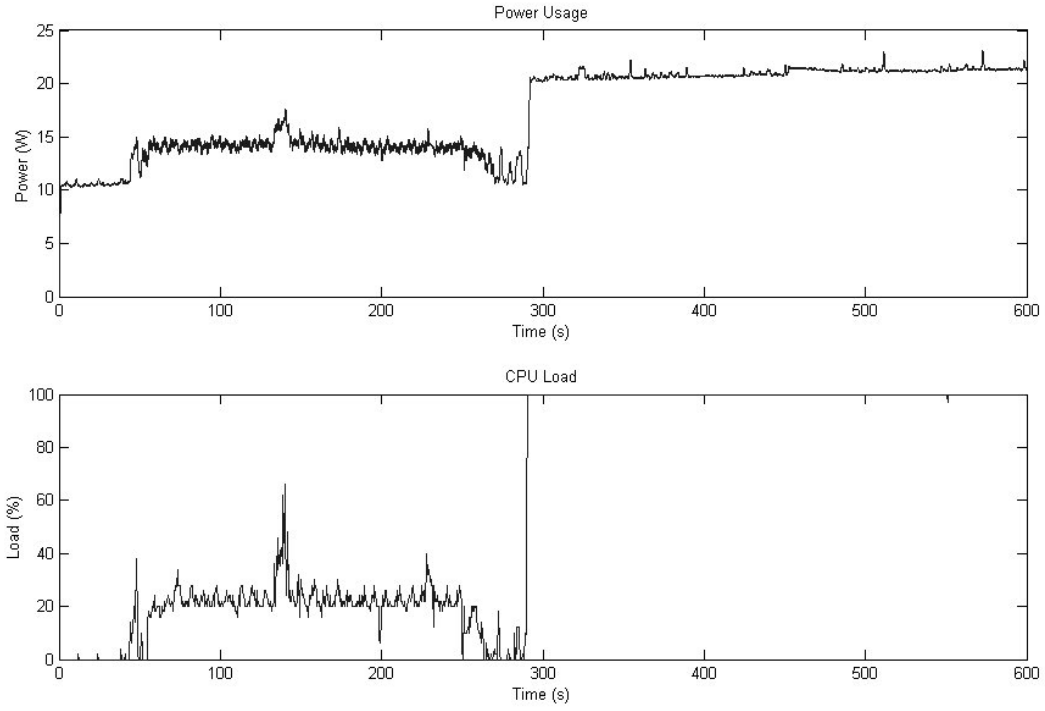

Figure 4.2: CPU load (bottom) has a direct affect on system power consumption (top).

Because of this apparent correlation, a simple linear regression model was developed with processor usage as the predictor variable and system power as the dependent variable [17]. A series of observational studies was done to develop a random sample set of processor load and system power. The least squares estimation method, in which the sum of squares of the errors about the regression line are minimized, was used to obtain equation 4.1 for estimating system power. While this model performed fairly well, with a coefficient of determination of

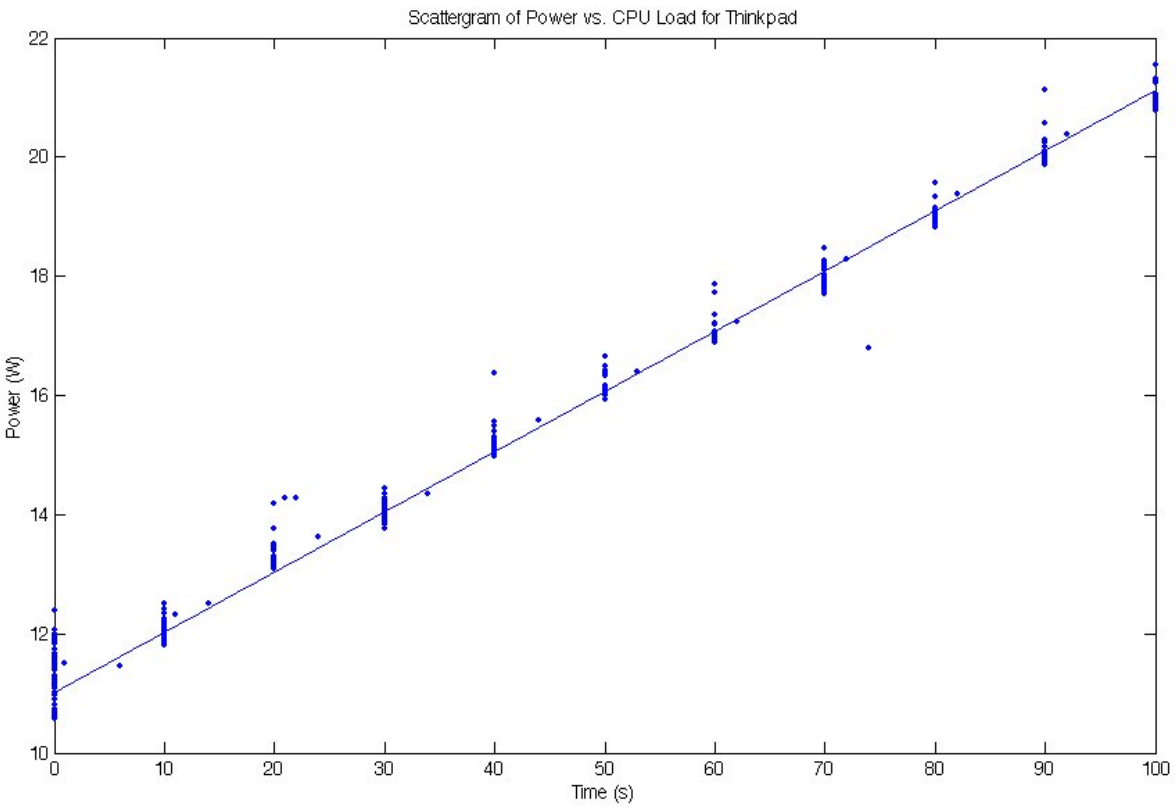

Figure 4.3: Scattergram of CPU load and system power shows a linear trend.

0.96, meaning it explained 96% of the variance of the sample data, and a root mean squared error (RMSE) of .719 W, it was hoped a multiple linear regression model taking into account more system parameters would be even more accurate [17].

$$\hat{\mu}_{Y|x} = \hat{y} = 11.0105 + 0.1011x \tag{4.1}$$

4.2 Selecting System Parameters

A first cut at selecting system parameters to monitor is to select those parameters that are most directly correlated to physical system devices. Ideally, these would include metrics

such as the number of bytes per second (Bps) processed by network interfaces, hard drives, memory accesses, and the like as well as the amount time the processor is being utilized. Other possible parameters include more advanced processor information such as the number of floating point operations preformed and cache information such as the number of hits and misses in a given time period. Unfortunately, those more advanced parameters were not readily available to collect on the systems tested.

The accuracy of a model's predictions from values that fall outside the data used to generate it is questionable. Therefore, it makes sense to have each device consume as much power as possible while testing by causing it to operate as fast as it can. To test the systems, various programs and benchmarks were employed. A program that caused increasing CPU load in 10% intervals by performing addition operations over longer and longer time intervals was written and used to map the processor behavior. The program used one controlling thread to control and increase the duty cycle of another thread performing the addition operations by alternatively making it active and pausing its execution on a 400 ms period. For example, to make the CPU usage 20%, the controlling thread would make the addition thread active for 80 ms and pause it for 320 ms. The program is included in Appendix A. FreshDevices' FreshDiagnose benchmarking program was used to tax the hard drive [7]. The IPerf bandwidth measurement tool was used to generate traffic on each system's wireless network interface [19]. It was also employed in testing the effects of the Bluetooth interface on the Latitude by establishing a network over that interface.

For both test systems, Microsoft Performance Data Counters were used to capture the system metrics [16]. That library allowed snapshots of system information to be collected, capturing all of the counter information at the same time. Many of the possible counters that could be gathered were more operating system centric and did not correlate with any particular system device. Those counters included such things as information about the file system cache, information about the paging file, and the number of jobs and threads in the system. Amongst the many counters offered, six of them had clear connections to physical system devices and were initially selected for the model:

- percentage of time the processor was busy with non-idle threads
- the number of Bps for physical disk reads
- the number of Bps for physical disk writes
- the number of Bps doing wireless network receives
- the number of Bps doing wireless network writes
- the number of memory pages faults per sec

The memory page faults proved to be ineffective in predicting the power of the test systems. Matlab was used to perform the multiple linear regression and frequently failed to complete the regression as the data collected during test runs led to singular matrices if the page fault metric was included. This indicated that the metric was not indicative of the power values recorded and had a regression coefficient of zero as far as Matlab's precision allowed. For the measurements where Matlab could perform the regression with the page fault metric, its contribution to the power model was negligible.

The general equation for the multiple linear regression model is shown in the following equation:

$$\hat{\mu}_{Y|x_1,x_2,...,x_k} = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_k x_k \tag{4.2}$$

β_0 represents the system's idle power level. The rest of the coefficients symbolize the contribution of each predictor variable to the total system power.

4.3 Test Systems and Power Measurement Setup

Two systems were used for model generation and evaluation, a IBM Thinkpad T23 and a Dell Latitude D600. The Thinkpad has a Pentium III Mobile processor operating at 866 MHz, 256 MB of main memory, a 5400 RPM 8 GB hard disk drive, a Lucent Technologies Orinoco 802.11b wireless card, a main battery with 39 Wh of capacity, and is running the Microsoft Windows 2000 Professional operating system. The Thinkpad was used in [14] to evaluate the effectiveness of different forums of battery exhaustion attack. The Latitude

has a Pentium M processor operating at 1.6 GHz, 576 MB of main memory, a 7200 RPM 40 GB hard disk drive, an internal Intel Pro/Wireless 2200BG 802.11g wireless network interface, a main battery with 53 Wh of capacity, and is running the Microsoft Windows XP Professional operating system. The latest Pentium M processors contain an Intel technology name Enchanced SpeedStep that allows it to change its operating frequency and voltage based on its current load [8]. This affected the model of the Latitude as will be shown below.

The power measurement setup used was the same as that used in [14], using an Agilent 3458A Digital Multimeter capable of sampling system current at 10,000 samples per second [1]. The sampling rate was reduced to 1000 Hz to help eliminate noise from the data, but that data rate proved to be more than sufficient. An Agilent 66319B Mobile communications DC source[2] was used to power the Thinkpad as a constant voltage source. This eliminated the variability of the main battery's output voltage over time. Unfortunately, the same could not be done with the Latitude due to an inability to find a suitable connector to match its main battery connection. Instead, the Latitude's AC power supply was employed with a sense resistor soldered into the supply's output cord. The main battery was removed to avoid effects of charging during the experiments.

The clocks of the two test systems were synchronized with that of the system recording the power of the test systems using Microsoft's implementation of the net time protocol (NTP). This allowed the metrics recorded on the test systems to be synchronized with the power values captured on the recording system. Not performing this step prior to testing caused significant variation in the regression coefficients for the power models.

4.4 Model Generation and Evaluation

The programs discussed above were run on the Thinkpad while recording the listed parameters and the system's power usage. Performing multiple linear regression on the data set arrived at the coefficients for the regression equation shown in Table 4.1. Stepwise regres-

Coefficient	System Component	Value
β_0	system	11.0105 W
β_1	processor	0.1011 W per % CPU
β_2	hard disk	6.4920e-7 W per Bps Read
β_3	hard disk	4.5541e-7 W per Bps Written
β_4	wireless network interface	-1.7654e-6 W per Bps Read
β_5	wireless network interface	-1.0992e-6 W per Bps Written

Table 4.1: Regression coefficients for Thinkpad model.

sion was used to determine the affects of the various parameters on the overall model. The processor usage was the most significant factor, as was expected from the simple regression model, with the hard disk read time and write time percentages contributing a small amount as well. These three parameters alone have a coefficient of determination of 0.9837 and a RMSE of 0.444 W. The remaining parameters have almost negligible improvements to the model, increasing the coefficient of determination to 0.9876 and decreasing the RMSE to 0.388 W.

The graph of Figure 4.4 shows the recorded and predicted power values over a benchmark run of the Thinkpad. The predicted power values, marked by diamonds, are overlaid onto the recorded power values, marked by circles. Figure 4.5 shows the percentage error between the actual and predicted readings over time. The average error over the data trace is 2.12%. Some of the significant errors appear to occur at large power transitions when the processor went from a highly active state to idle or vice versa.

For the Dell Latitude, the range of power values observed was much higher, with an idle power of approximately 12 W and a peak power of approximately 30 W observed during testing. The realtime frequency and voltage scaling technology changed the response of the power to processor usage from a linear model into one that looks more a polynomial pattern, as shown in Figure 4.6. It is interesting to note that disabling that technology causes the

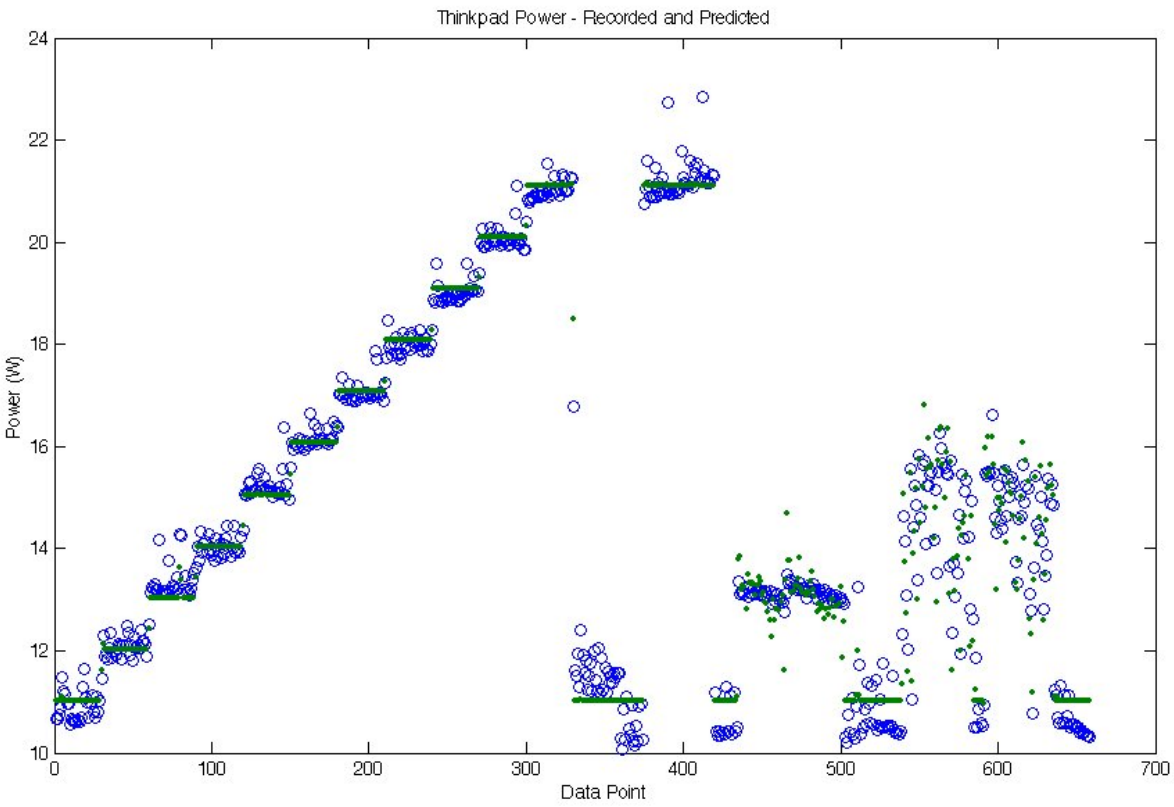

Figure 4.4: Comparison of power estimation (green diamonds) with actual power usage (green circles) for Thinkpad.

Pentium M to behave in the same linear manner that the Pentium III exibited. This is shown in Figure 4.7.

It is desirable to model the Latitude with the SpeedStep technology enabled as this is the most power efficient way for it to operate. This changed the regression equation to one of the form shown in Equation 4.3.

$$\hat{\mu}_{Y|x_1,x_2,...,x_k} = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + ... + \beta_k x_1^k + \beta_{k+1} x_2 + ... + \beta_{k+n} x_n \tag{4.3}$$

It was found that, for the data in Figure 4.6, the power to processor usage was best modeled by a polynomial of degree three. Using that data as well as the data generated from running the other benchmark programs, regression coefficents were generated and are shown in

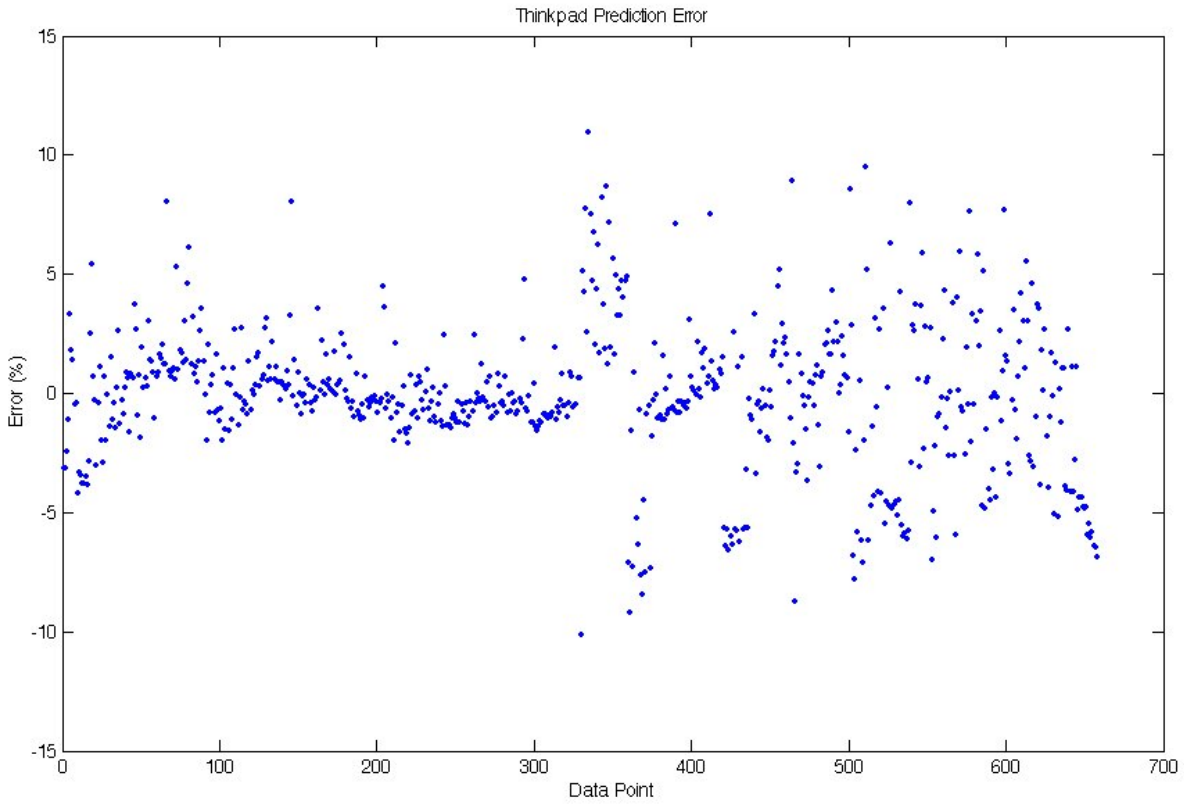

Figure 4.5: Percent error of power prediction for Thinkpad model.

Table 4.2.

The benchmark run for the Latitude with the recorded power values, marked with blue circles, and predicted power values, marked with green diamonds, is shown in Figure 4.8. The error of the Latitude's regression model is higher than the Thinkpad's with the average error for the benchmark run at 4.46%. The error over the entire benchmark run is shown in Figure 4.9.

As shown in Figure 4.8, during the peak power consumption from around data points 290 to 300 seconds and 340 to 350, there was a ramping of the power that the regression model did not predict. As the only program running at that point was focusing on processor usage, it was believed the increased power consumption might be due to thermal influences on the processor. In [18], the effects of temperature on transistor leakage current is discussed.

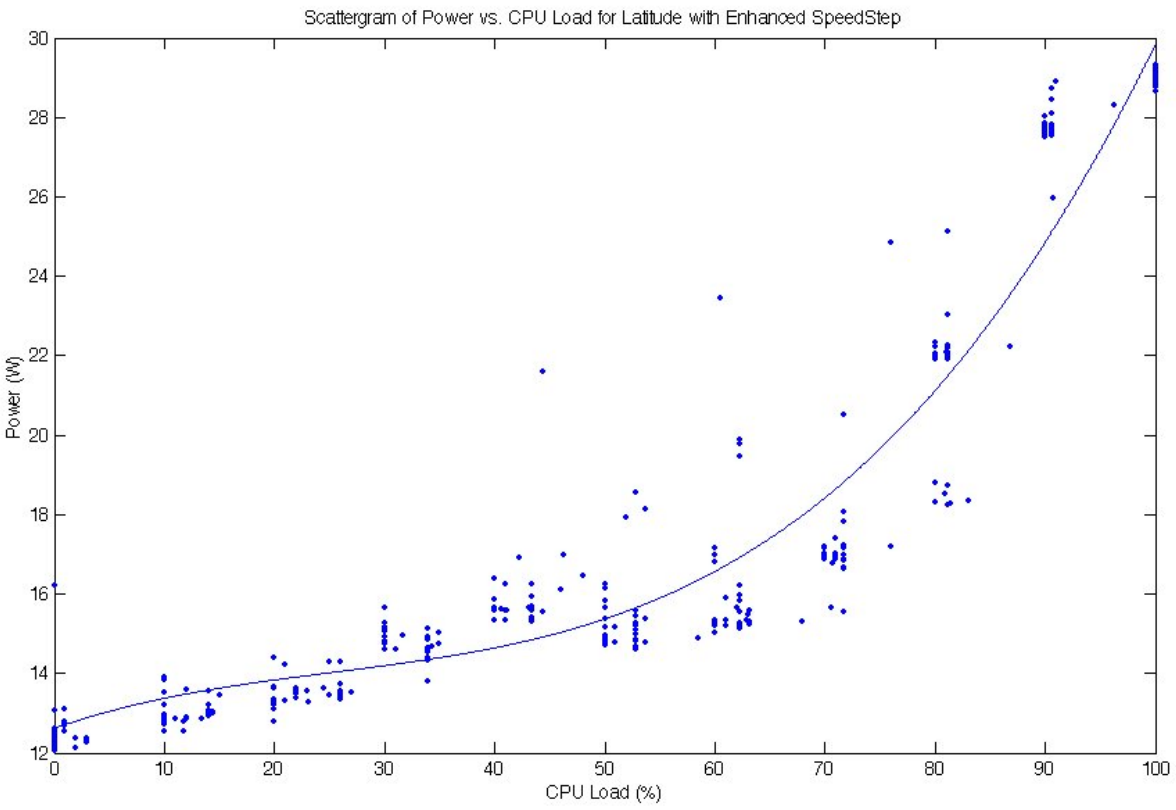

Figure 4.6: Scattergram of power versus CPU load for Latitude with SpeedStep technology enabled.

The high current requirement when the CPU is active 100% of the time causes an increase in chip temperature. This in turn causes an increase in thermal voltage which causes an increase in leakage current. This cycle continues until the temperature reaches a steady state value. Using an open source program named I8kfanGUI, designed to monitor and control the thermal management of Dell notebooks, it was observed that the power ramping coincided with an increasing processor temperature. Using portions of code from I8kfanGUI, temperature was added as another metric to the regression model [5]. Initial results appeared promising with the model more closely following the power increase at peak processor usage. The recorded power values and predicted values for one data set are shown together in Figure 4.10.

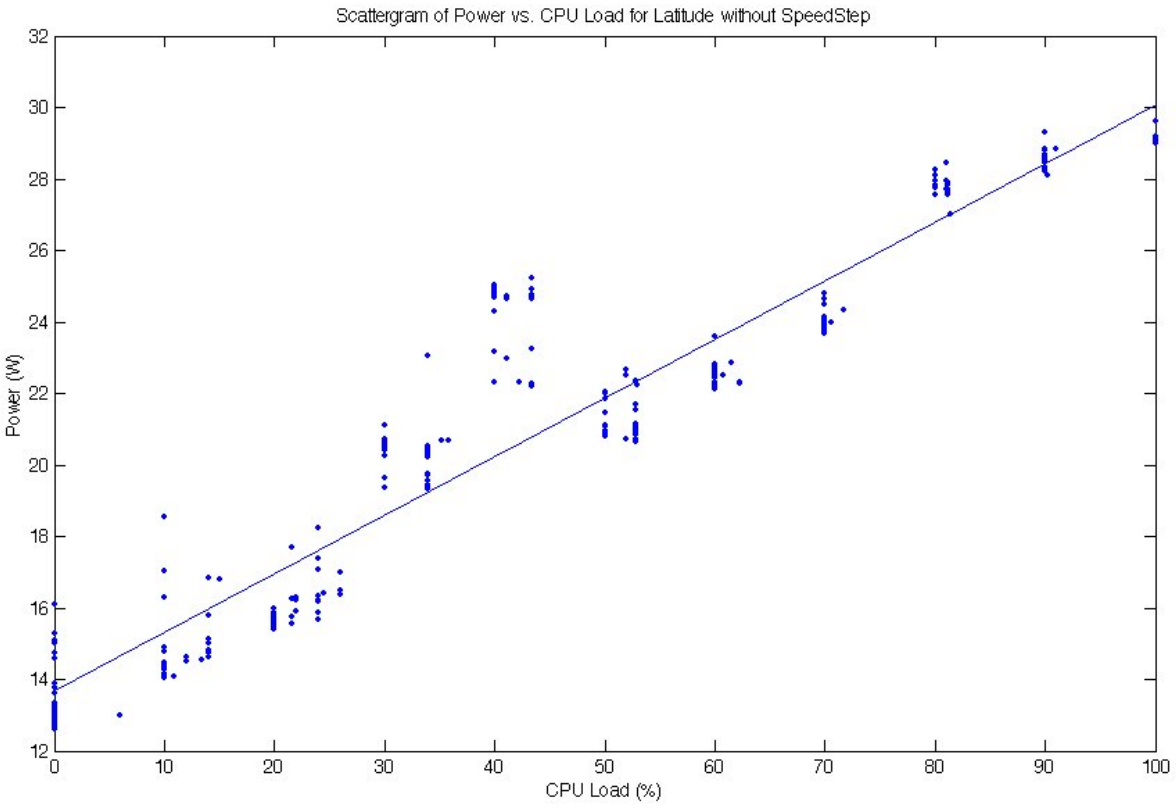

Figure 4.7: Scattergram of power versus CPU load for Latitude with SpeedStep technology disabled.

It was found, though, that the regression coefficients for these models varied greatly between data sets. Also, the temperature compensated models overpredicted the power of the notebook for low processor activity that occurred immediately after periods of high processor activity. This phenomenon can be seen in Figure 4.10 around 180 to 210 seconds. Further review of the processor temperature's effects on the notebook's power consumption indicate that, while having a noticeable effect, it was not linear or polynomial in nature but mostly likely better modeled by a more complex regression model, perhaps as some product with the CPU load. Models generated without temperature information had more repeatable accuracy and so temperature information was not selected to be included in future models.

Like the memory page faults, the effect of the Bluetooth interface on the Latitude's power

Coefficient	System Component	Value
β_0	system	12.6199 W
β_1	processor	0.0965 W per % CPU
β_2	processor	-0.0024 W per % CPU^2
β_3	processor	3.1715e-5 W per % CPU^3
β_4	hard disk	1.9846e-7 W per Bps Read
β_5	hard disk	2.8964e-7 W per Bps Written
β_6	wireless network interface	1.2630e-6 W per Bps Read
β_7	wireless network interface	1.8311e-6 W per Bps Written

Table 4.2: Regression coefficients for Latitude model.

consumption was negligible. If the radio was in an active power state, it increased the idle power level by approximately 3 W, but the amount of data transmitted or received over the interface had no noticeable power increase. Aside from noting that it should be determined whether the radio was in an active or passive power state, its influence on the regression model was not considered further.

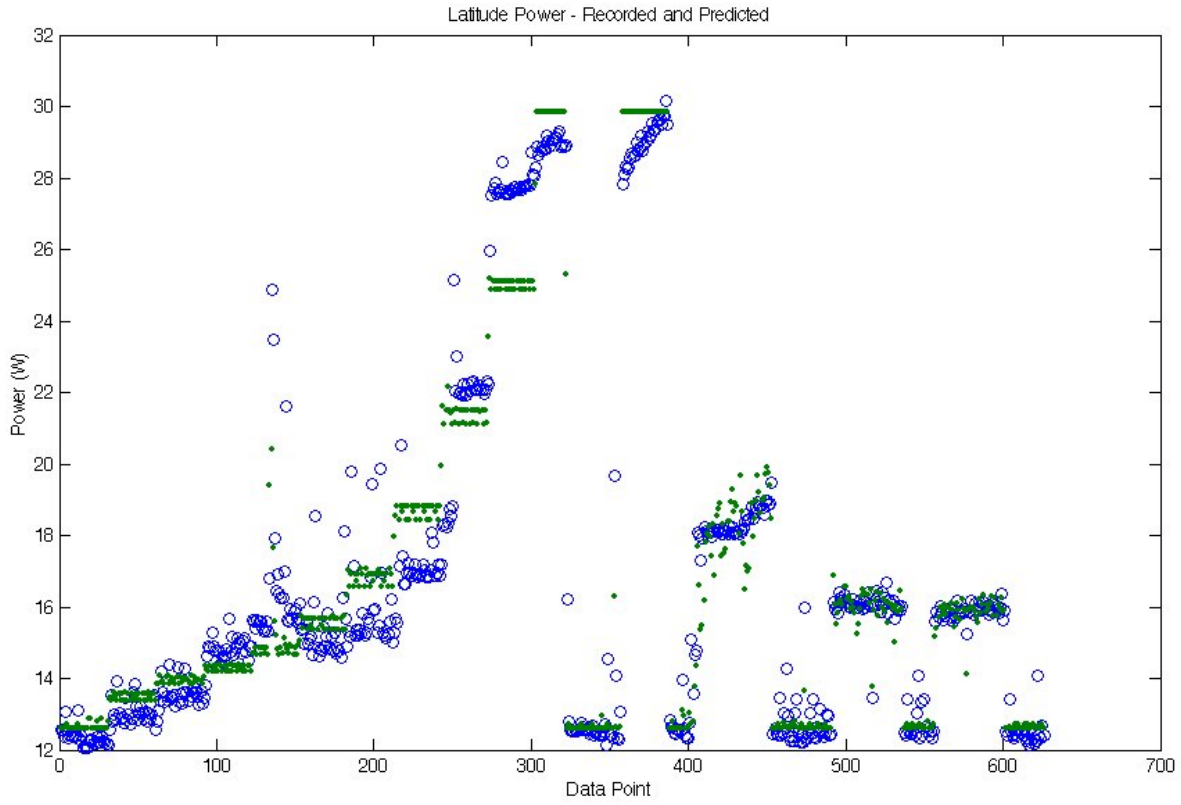

Figure 4.8: Comparison of power estimation (green diamonds) with actual power usage (blue circles) for Latitude.

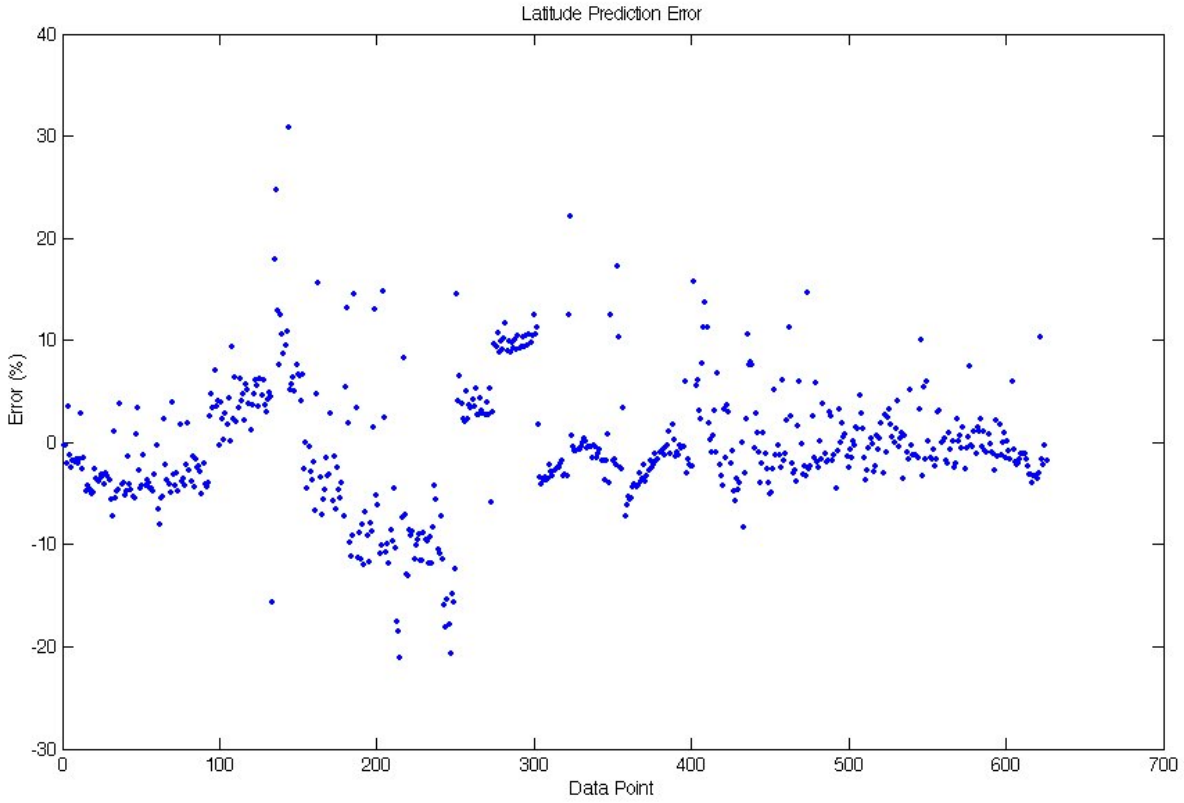

Figure 4.9: Percent error of power prediction for Latitude model.

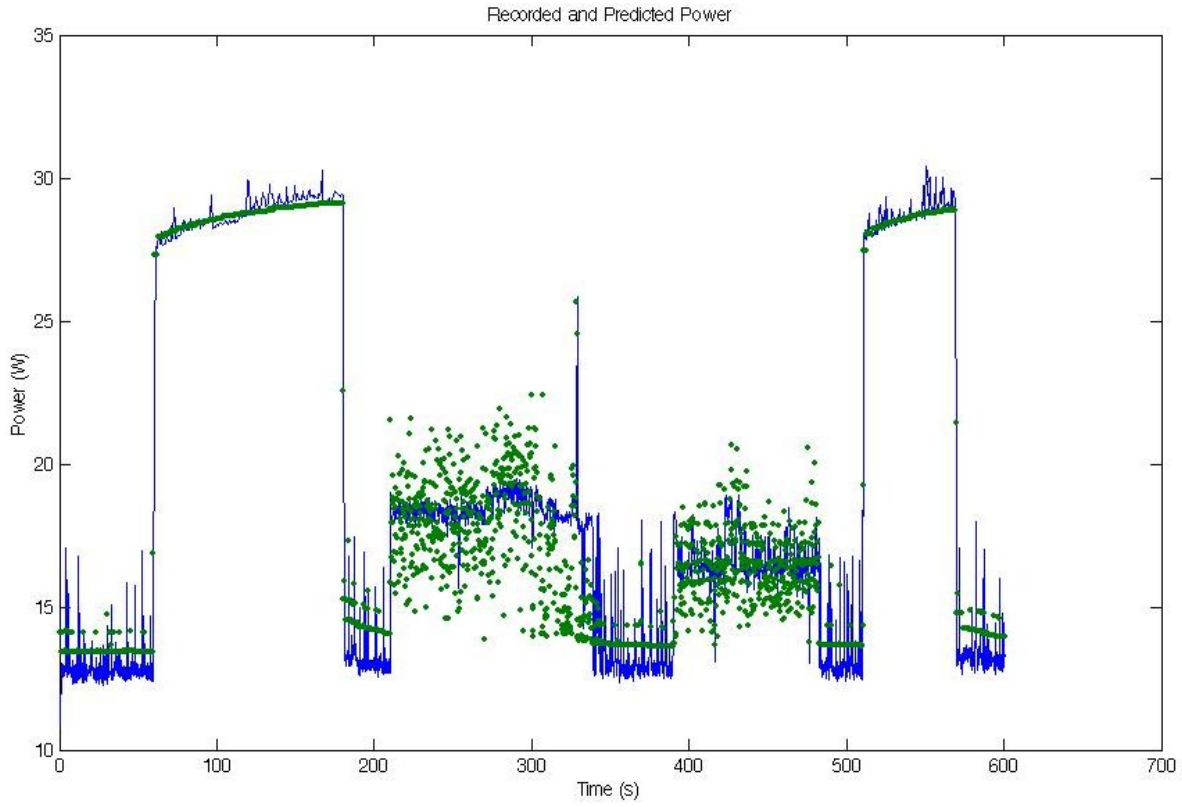

Figure 4.10: Temperature compensated model with recorded power (blue line) and predicted power (green diamonds).

Chapter 5

Intrusion Detection System

This chapter focuses on the development of an intrusion detection system (IDS) against battery exhaustion attacks for mobile computing devices. First, potential limitations on the system are discussed. Then, the affects of inaccuracies of the linear regression model on the performance of the intrusion detection system are discussed as well the affects of using a moving window to average power predictions and tune the system. Finally, an implemented detection system is analyzed and critiqued.

5.1 IDS Constraints

As compared to desktop or enterprise computer systems on which traditionally studied and commercially available intrusion detection systems are deployed, there are a large number of restrictions imposed on any IDS that can be deployed on most mobile devices. These restrictions include limited processor performance, memory, and power usage. Whereas a typical desktop system today may have upwards of one to two gigabytes of memory, a typical IPAQ has 64 megabytes of main memory available. Likewise, while desktop systems have a large amount of processing power available with processors operating upwards into the 3

GHz range, most mobile devices have processors operating in just a few hundred megahertz. While it is to be expected that these values will improve in the future, mobile devices will always be more resource-constrained relative to desktop systems. Any system of detection implemented on these mobile devices must have a small footprint and limit the amount of power it consumes. Any solution to the problem of intrusion detection and specifically to the problem of detecting battery exhaustion attacks cannot itself incur a large power and performance requirement, or else it could be used as an avenue of attack.

Methods used in commercial IDSs may be difficult if not impossible to implement on mobile devices. For instance, the extensive audit data, in the form of system logging, collected and analyzed by an IDS may simply not be present or consume too many device resources to collect. Also, extensive analysis of this data may consume too much power or make a system too unresponsive for a user for it to be worthwhile. Most network IDSs also rely upon the cooperation of several detectors to gather enough information to cover the entire network. Most mobile devices operate independently.

The goal of an IDS that detects battery exhaustion attacks must be to identify attacks that cause the system to consume too much energy. The attacks can not be prevented from using any energy, but the amount of energy they consume can be mitigated. Consequently, the goal for a battery exhaustion IDS should be to guarantee a specific percentage of the overall battery life of the system. If a device could operate at idle power for 3 hours under normal usage, the goal might be to guarantee 2 hours of operational life in the face of repeated attacks. Given this goal, the IDS must first know when the system has high power consumption over a long period of time, such that the system is in danger of not meeting the guaranteed battery life. When the time threshold has been exceeded, the IDS should then identify which process or processes are responsible for using the most energy over that period of time. Such an IDS is unique in that it can still be successful even if it does not detect all attacks against the system. It will allow attacks through that do not cause the system to exceed its energy consumption threshold that is discussed below. In that case, even though the attacks are successful, the goal of guaranteeing a specific battery life can

still be achieved.

5.2 Accuracy of the Model

This section discusses how the inaccuracies of a power prediction model affect its use in an intrusion detection system and what compensations must be made. Firstly, some common terms used to described the efficiency of an intrusion detection system should be introduced. In an intrusion detection system, the *false positive rate* of the system is the percentage of time the system identifies an attack as occurring when in fact one is not. The complementary measurement, known as the *false negative rate*, is the percentage of time the system does not identify an attack as occurring when in fact one is. For an intrusion detection system based on a power model, there are also some important measurements that should be identified. The term *guaranteed battery life* is used to express the desired operating time the computing device should achieve given its main battery's capacity. The *power threshold* is the average power level that a computing device cannot exceed if it is to meet its guaranteed battery life. The following example illustrates how the threshold can be calculated:

> A computing device has a battery rated at 53 Wh. Assume the battery is capable of delivering 100% of that rating. A consumer would like the device to have a guaranteed battery life of 2 hours. The power threshold for that device is then 53 Wh / 2 hours = 26.5 W.

26.5 W, then, is the highest average power the device can operate at and still meet the guaranteed battery life of 2 hours. Most likely, the threshold will have to be adjusted as the battery will not deliver 100% of its capacity, but those effects are not considered here.

If the model where 100% accurate, the false positive rate of the intrusion detection system would only be the percentage of time non-malicious programs caused the computing device to exceed the power threshold. Likewise, its false negative rate would be those attacking

programs that fail to exceed the power threshold. It turns out that, if the goal of the intrusion detection system is only to have the computing device meet its guaranteed battery life and the power can be accurately determined, a false negative is not of great importance.

Unfortunately, barring a high speed power measurement system built into the system itself, which would introduce concerns about power analysis attacks as described in section 4.1, 100% accuracy cannot be achieved in the power model. One way of compensating for the error of each prediction is the use of statistical confidence intervals. A 90% confidence interval for a given prediction is a range of values within which it is 90% likely the true value lies. Higher and lower percentage intervals can be used, but 90% is a typical value used in evaluating statistical models. For a simple one variable linear regression model, finding and plotting the intervals over the range of predictions which are of interest will give confidence bands. Shown in Figure 5.1 is a close-up of a regression line for power versus processor usage around the ideal threshold value, marked on the Y axis as T_I. The confidence bands for the regression line are shown as dashed lines above and below the regression line. With the 90% confidence bands, for a given point on the regression line, it is 90% likely the true value of the power lies on a line segment that runs through the point on the regression line parallel to the Y axis between the lower confidence band and upper confidence band. An example, marked as point (Ce, Pe), is also shown in Figure 5.1.

As can be seen at the example point, part of its interval lies above the ideal threshold and part below it. Due to this, it is not possible to state whether or not the actual value of the power is above or below the threshold at this prediction point. There are a range of predicted points, then, whose confidence intervals include the threshold that will be indicative of an attack even if the actual power values they predict lie below the threshold. These values are shown as the shaded region in Figure 5.2. The only way to reduce this region is to decrease the residual error and variance of the regression model. Models that more accurately account for the variance of the power measurements will have tighter confidence bands and their area of uncertainty will be smaller.

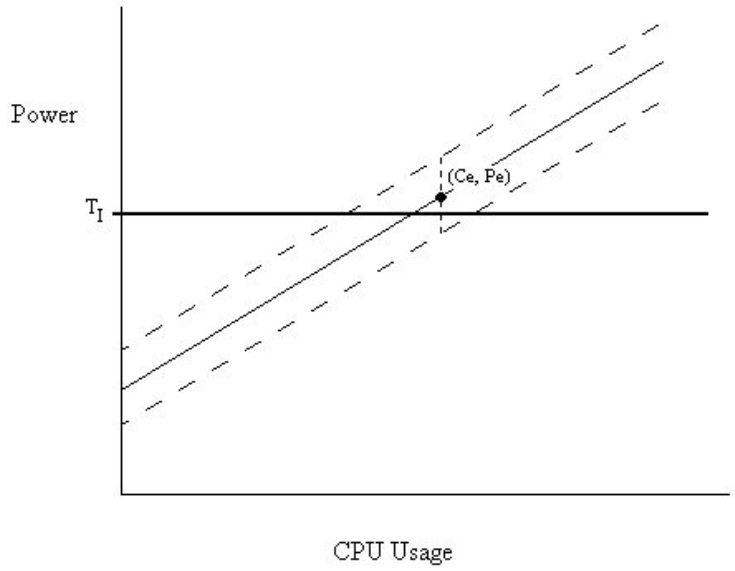

Figure 5.1: Regression line with confidence bands.

For multiple linear regression models, a confidence interval must be calculated for each combination of predictor variables. Even with just a few predictor variables with a moderate range of possible values, it may not be feasible in terms of memory requirements to calculate and store the confidence intervals beforehand. If processor power allows, the best choice then is to calculate the confidence interval for each power prediction while the intrusion detection system is running. For the data obtained for the two test systems used for this research, the confidence intervals remained relatively constant across all observed values, varying by less than 0.1 W and could most likely be set as a constant value to reduce the computation costs of the IDS. The average Thinkpad confidence interval was $\pm$ 0.64 W. The average Latitude confidence interval was $\pm$ 1.95 W.

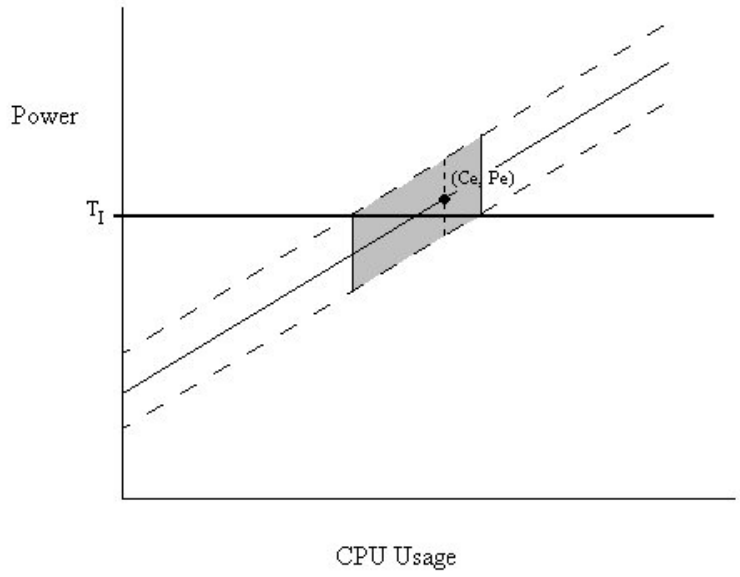

Figure 5.2: The shaded region represents all values that would indicate an attack.

5.3 Window Size

As was shown in the power figures of Chapter 4, the power usage of the test systems had large but brief power increases even when system power was relatively constant. Some of those increases temporarily exceeded the power threshold of their systems. Rather than work with the instantaneous power predictions of the linear regression model, the intrusion detection system can work with the average of a moving window of power predictions, helping to smooth those brief power increases. Figure 5.3 illustrates this smoothing. The top graph shows the instantaneous power predictions generated by the regression model. The bottom graph shows the average values for a moving window of 1 second.

The size of this moving window directly affects the responsiveness of the intrusion detection system to large transitional power changes. The following example illustrates this affect:

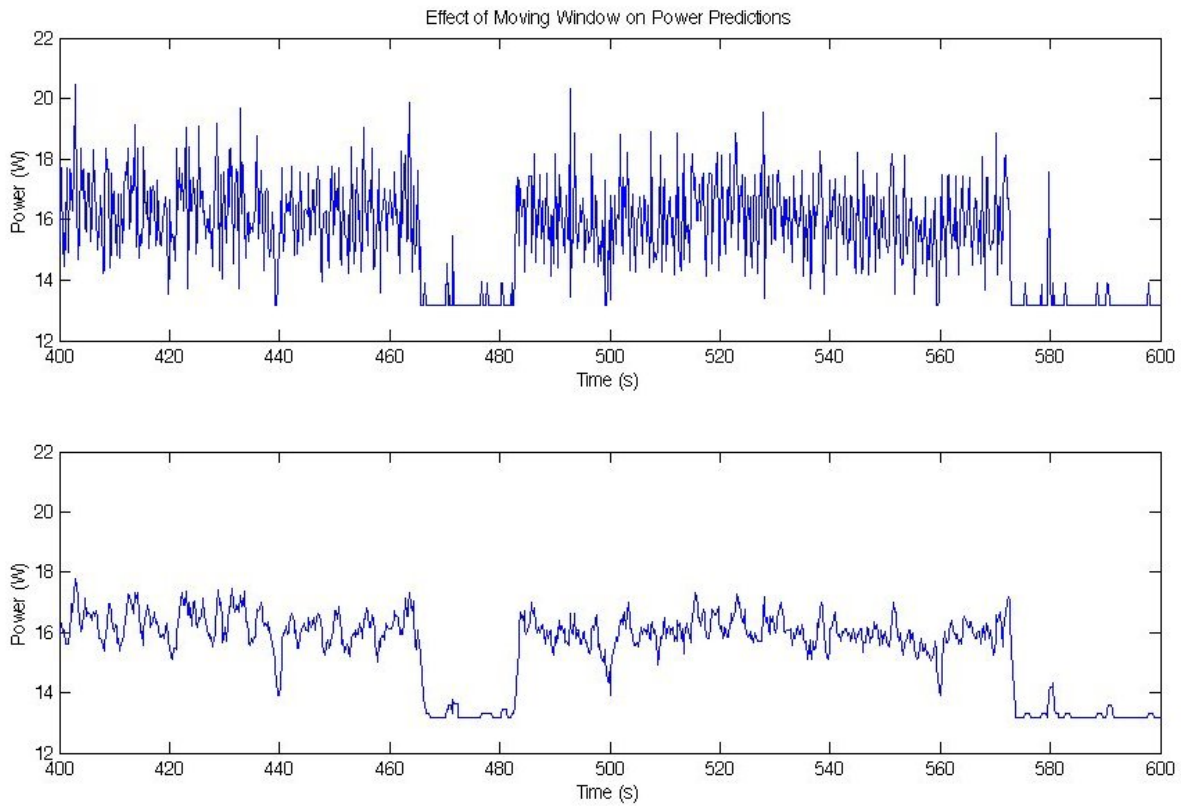

Figure 5.3: Comparison of power estimation (top) to the average power with a moving window of size 5 (bottom).

Assume the power usage of a system between predictions is constant and the predictions are made every 1 second. That is, if a prediction of 5 W is made, the system is operating at 5 W for the full second until the next prediction is made. The power threshold of the system is 14 W and its battery capacity is 28 Wh. Take the following data set in Watts: 0.00, 0.00, 5.00, 10.00, 20.00, 5.00, 0.00, 5.00, 5.00, 0.00. Starting with the fourth element, with a moving window of size 2, the average power values in Watts would be 7.50, 15.00, 12.50, 2.50, 2.50, 5.00, and 2.50. Again starting with the fourth element, if the moving window is of size 4, the average power values in Watts would be 3.75, 8.75, 10.00, 8.75, 7.50, 3.75, and 2.50.

With a window size of two, the average power values exceed the power threshold while the average values for a window size of four do not. For a given time interval, the larger window size allows for higher peak power values than the smaller window size. This fact can be useful in tuning the intrusion detection system. If there exist non-malicious programs that tend to exceed the power threshold of the system in small intervals but overall consume less power, the window size can be increased to allow for this, reducing the false positive rate of the intrusion detection system. Studying the power usage of a mobile device while running the applications most likely to be used on it will help to select the proper window size. If P_T is the threshold power, n_W is the size of the moving window, and p_i is an instantaneous power prediction, then Equation 5.1 shows the relationship of the power over the window and the threshold power. When the average power over the window exceeds the threshold value, the detection system will detect an attack.

$$\frac{\sum_{i=0}^{n_W} p_i}{n_W} \leq P_T \tag{5.1}$$

Rewriting Equation 5.1 to find the relationship to the window size yields Equation 5.2.

$$\frac{\sum_{i=0}^{n_W} p_i}{P_T} \leq n_W \tag{5.2}$$

By maximizing the numerator of the left-hand side of the equation, the maximum necessary window size can be found.

There is a tradeoff for selecting larger window sizes, however. Since the moving window allows for power values above the threshold, it is possible for an attacking program to consume more energy than should be allowed to meet the guaranteed battery life. In the worst case, if an attacking program causes the mobile device to consume power at just above the power threshold, it can take almost the full duration of the moving window before the attack is detected. For very large window sizes, on the order of a few minutes, this effect could be noticeable on the device's battery life. Fortunately, the window sizes that will typically be used are on the order of a few seconds and the capacity lost due to this will have a minimal impact on the battery life. Likewise, the false positive and false negative rates of the IDS will be negatively impacted by larger window sizes.

5.4 IDS Implementation and Evaluation

With the power estimation, it can be determined when the system has exceeded its power threshold and the goal of reaching its guaranteed battery life cannot be guaranteed. The task is then to determine what processes are causing the increased power consumption. While all the metrics used in estimating the power cannot be easily determined on a per process basis, the amount of processor usage can be. From the regression models, the processor usage proved to be the largest factor in power consumption on both test systems, modeling most of the variability of the power on its own. While not ideal in that all metrics cannot be used, using the processor usage of each process as a means of determining its effect on overall system power usage is a good starting point.

Selecting the models that performed well for several data sets on each test system, an IDS was written that gathered the performance metrics and calculated power values in real time. When the IDS calculates that the average power has exceeded the power threshold, it attempts to determine which process or processes are responsible for consuming the most energy. Using the percentage of time a process was reported to have used the processor, the calculated total device power was divided amongst the running processes for the measurement time interval.

Additionally, to combat against the distributed attacks discussed in section 3.1, a hierarchy tree of processes was created. Under the Microsoft operating systems, a unique process identifier is assigned to each process, and the identifier for each process' parent process, the process that created it, can also be determined. These two values allowed the hierarchy to be created. For each process, the cumulative power consumption of the process' children was computed and added to the consumption of the process itself. This allowed the IDS to detect any group of processes that behave in either of the scenarios that were outlined. Figure 5.4 shows the hierarchy tree during a distributed attack following the first scenario. Figure 5.5 shows the tree for an attack following the second scenario.

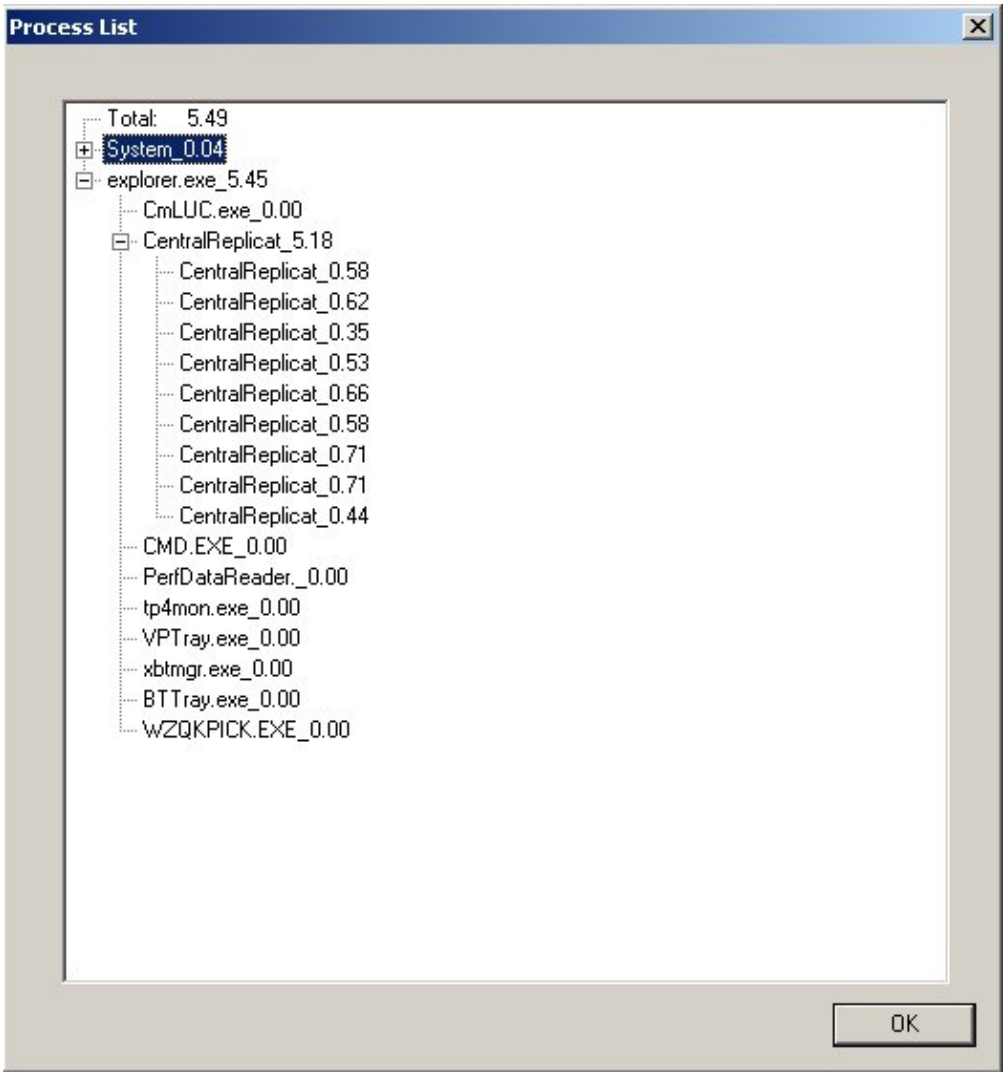

Figure 5.4: Distributed attack following first scenario.

Seven attacks were run on both test systems and the results recorded. These attacks included the cache, animated GIF, SSH, and two distributed malignant attacks described in section 3.1 and a form of the service attack described in section 3.2. The child programs of the distributed attacks performed repeated addition to tax the processor while they were active. While the power threshold is normally set based on a desired guaranteed battery life, for testing the IDSs, thresholds were selected that fell in the middle range of power values for their respective systems. The Thinkpad threshold was set at 15.60 W, which

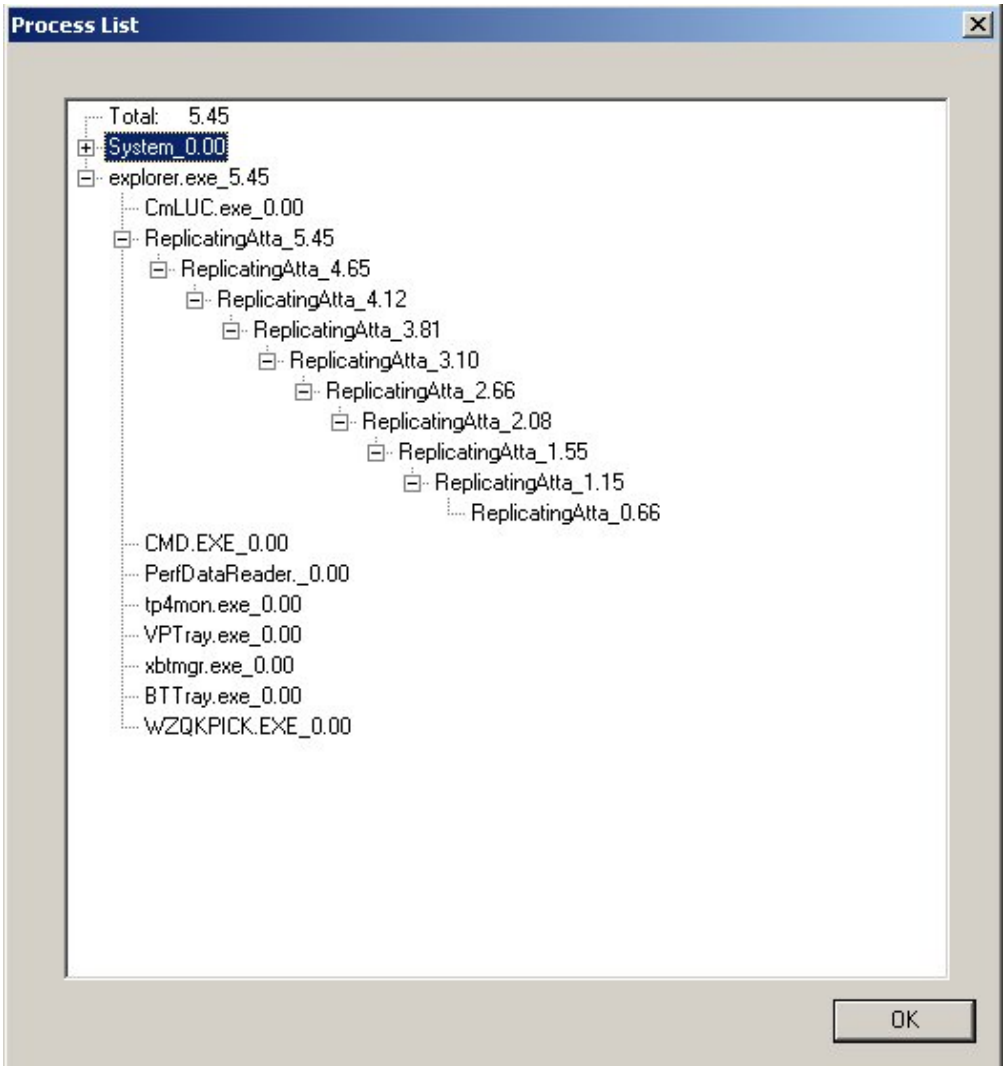

Figure 5.5: Distributed attack following second scenario.

would correspond to a guaranteed battery life of 2.5 hours. The Latitude threshold was set at 17.67 W, which would correspond to a guaranteed battery life of 3 hours. A window size of 1 second was used for both systems.

Figure 5.6 shows the averaged predicted power values, marked by green diamonds, overlayed on the recorded power values, marked by blue circles, for the Thinkpad during the cache, animated GIF, and SSH attacks. The periods during which the cache, animated GIF, and SSH attacks were active are marked on the graph as areas 1, 2, and 3, respectively. The

average error of the predictions for the whole time interval was 4.91%. Without taking into account the confidence intervals of each prediction, the false positive and false negative rates were 0.00% and 1.11%, respectively. If the confidence intervals are added to the predictions, as seen in Figure 5.7, the false positive and false negative rates remain the same. During the cache attack, the IDS correctly identified the attacking program. The animated GIF attack caused the web browser viewing the image, Internet Explorer, to consume more power and it was also correctly identified. During the SSH attack, the OpenSSH server running on the system was identified as the attacking process [15].

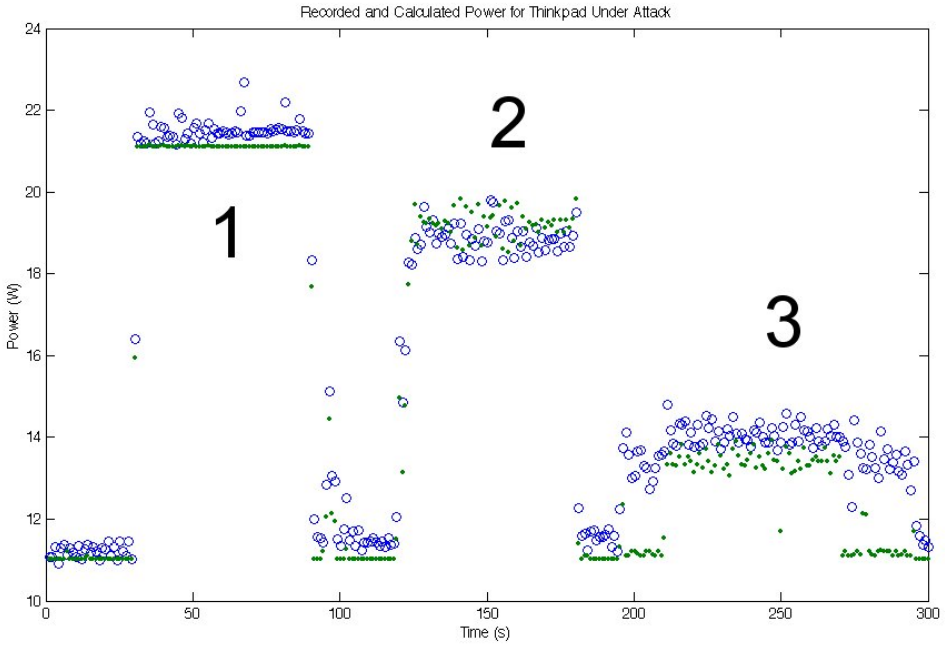

Figure 5.6: Averaged predicted power (green diamonds) and recorded power (blue circles) for Thinkpad during cache, animated GIF, and SSH attacks.

Figure 5.8 shows the averaged predicted power values, marked by green diamonds, overlayed on the recorded power values, marked by blue circles, for the Thinkpad during the service attack, the distributed attack following scenario one, and the distributed attack following scenario two, marked on the graph as areas 4, 5, and 6, respectively. The average error of the predictions was 4.53%. Without the confidence intervals, the false positive and

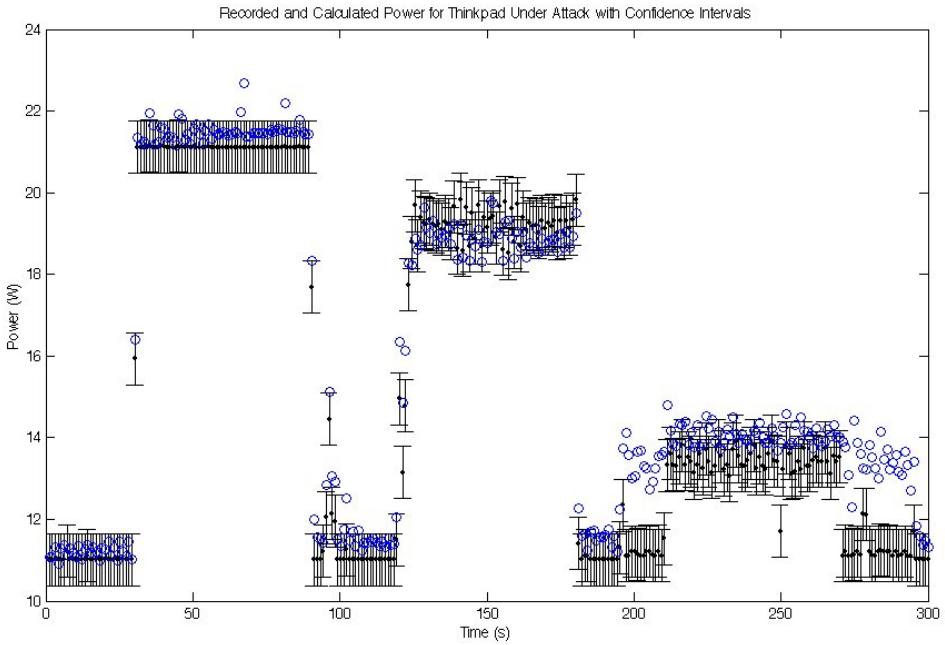

Figure 5.7: Averaged predicted power (black diamonds) with confidence intervals and recorded power (blue circles) for Thinkpad during cache, animated GIF, and SSH attacks.

false negative rates were 0.00% and 15.5%, respectively. Taking the confidence intervals into account, both rates become 0.00%. The predicted values with confidence intervals are shown in Figure 5.9. The distributed attacks were correctly identified as the group of processes causing the increased power consumption by the IDS. The service attack caused the Notepad program to be detected as the guilty program.

During the service and distributed attacks, the Thinkpad model failed to account for part of the power consumption. The nature of these attacks is such that they cause a great deal of context switching in the operating system. This subsequently causes the cache of the processor to be flushed at each context switch. The work done in [11] has shown that cache misses can cause elevated power consumption above what is normally seen for a given CPU load. This appears to be the case for these three attacks. For the two distributed attacks, the confidence intervals of the predictions cover most of the recorded power values and they would still be flagged as attacks if the threshold were at their power levels. The overall false

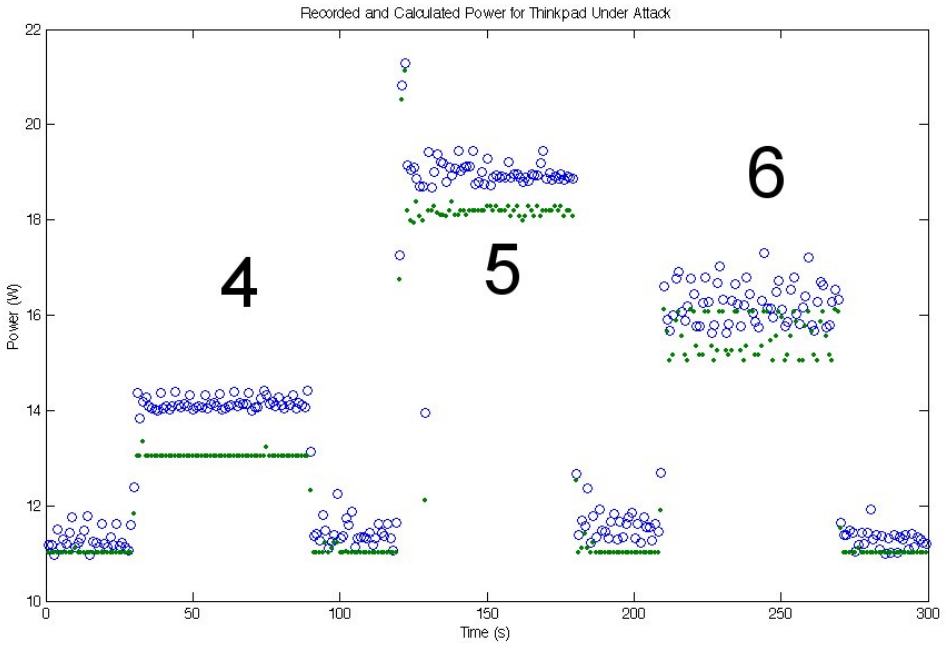

Figure 5.8: Averaged predicted power (green diamonds) and recorded power (blue circles) for Thinkpad during service attack and distributed attacks.

positive and false negative rates for the Thinkpad IDS over the two data sets were 0.00% and 0.55%, respectively.

Figure 5.10 shows the results for the Latitude during cache, animated GIF, and SSH attacks, marked in the same order and manner as the Thinkpad results. The average error over the entire time interval was 6.75%. Without the confidence intervals, the false positive and false negative rates were 0.00% and 4.22%, respectively. Including the confidence intervals, as shown in Figure 5.11, the false negative rate decreased to 3.28%. As with the Thinkpad, all of the attacking processes were correctly identified. If the threshold were instead set at 16.00 W, without the confidence intervals, the false negative rate becomes much worse at 50.00% while the false positive rate remains 0.00%. With the confidence intervals, the false negative rate falls to 0.89%. Though the Latitude model suffers from more inaccuracy than the Thinkpad and the large confidence intervals will affect its ability to properly detect or ignore power levels close to the threshold, they help to compensate for the error of the

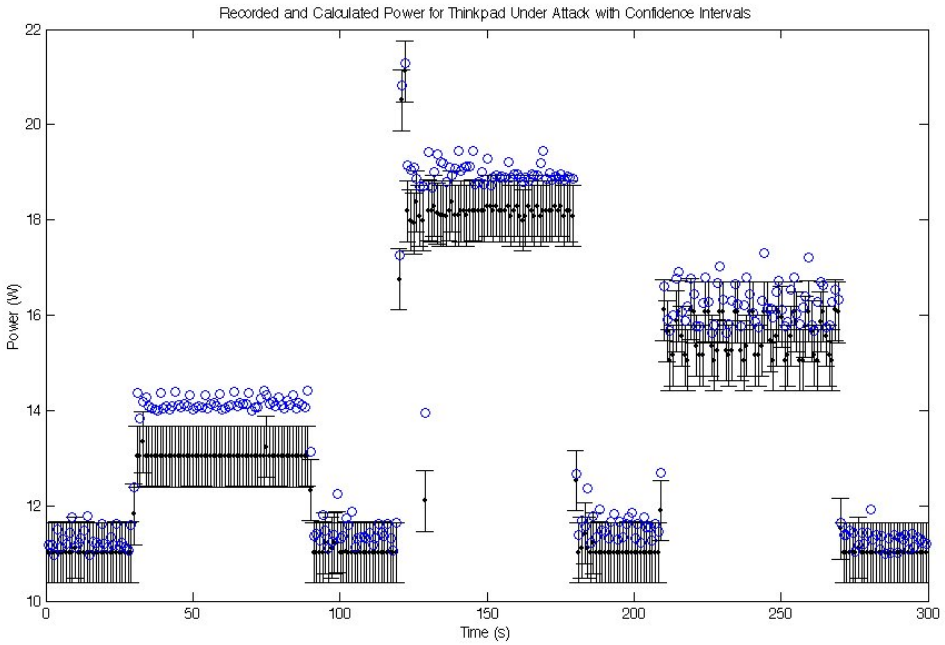

Figure 5.9: Averaged predicted power (black diamonds) with confidence intervals and recorded power (blue circles) for Thinkpad during service attack and distributed attacks.

predictions as shown in these two cases.

Figure 5.12 shows the results for the Latitude during the service attack, distributed attack following scenario one, and the distributed attack following scenario two, again marked in the same order and manner as the Thinkpad results. The average error for the predictions was 7.95%. The false positive and false negative rates without confidence intervals were 0.00% and 21.70%, respectively. Including the confidence intervals, as shown in Figure 5.13, the false positive rate remains the same, while the false negative rate increases slightly to 21.90%. The distributed attacks were correctly identified as the group of processes causing the increased power consumption by the IDS. Again, Notepad was shown as the guilty program during the service attack.

The overall false positive and false negative rates for the Latitude IDS over both data sets were 0.00% and 12.5%. Most of these false negatives occurred during the last distributed

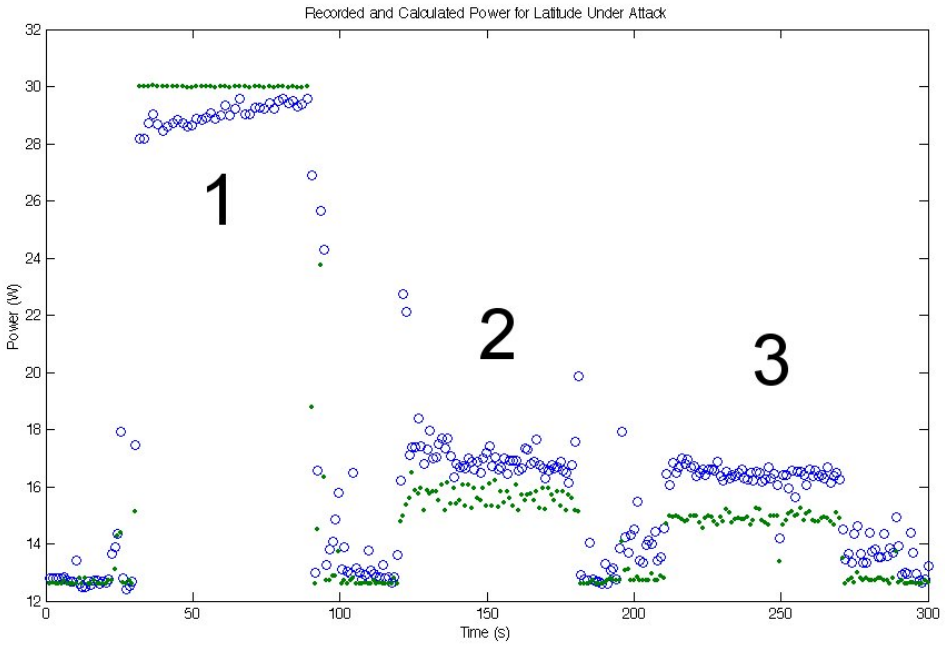

Figure 5.10: Averaged predicted power (green diamonds) and recorded power (blue circles) for Latitude during cache, animated GIF, and SSH attacks.

attack. In the range of 35% to 45% average CPU load that this attack caused on the Latitude, the model underpredicts the power, as shown in the scattergram in Figure 4.6. This, combined with the same effects seen in the Thinkpad results believed to be caused by cache misses, caused the large error during this attack. Further improving the Latitude model is desirable as the SpeedStep function already helps to reduce the power consumption and extend battery life.

Figures 5.14 and 5.15 show the recorded and calculated power with confidence intervals for the Thinkpad over a five minute period of time during which normal user activities were conducted with no malicious attacks. Figures 5.14 shows the IDS values for a one second moving window while Figure 5.15 shows the IDS values for a five second moving window. Microsoft Word and the Mozilla Firefox browser were used during this period of time for typing a document and viewing several web pages including CNN.com, Google.com, and randomly returned pages from Google searches. As shown in those figures, at approximately

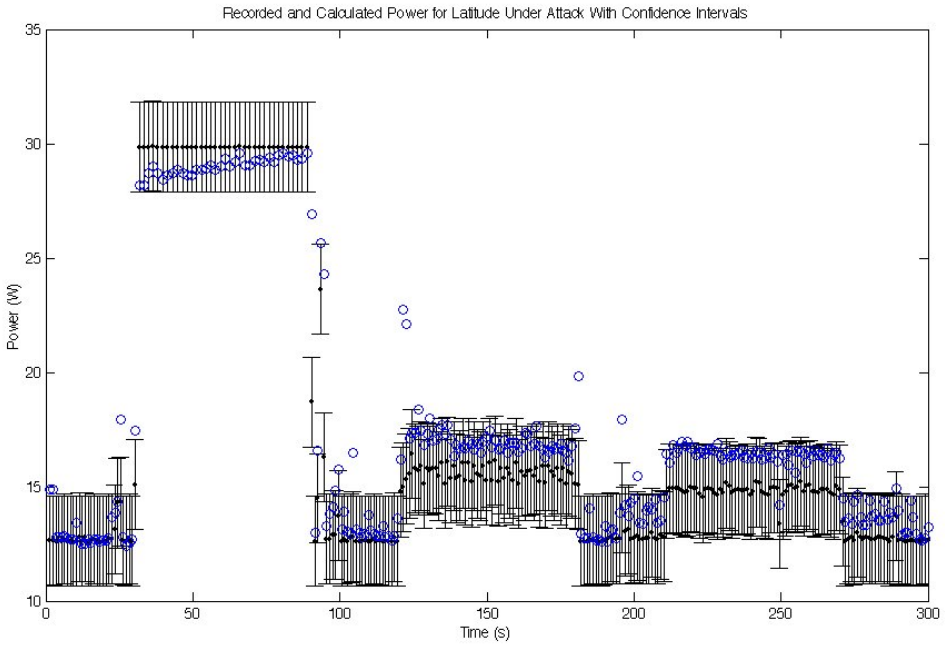

Figure 5.11: Averaged predicted power (black diamonds) with confidence intervals and recorded power (blue circles) for Latitude during cache, animated GIF, and SSH attacks.

140 seconds, a large power spike occurred, which corresponded to the loading of the Firefox browser. As shown in Figure 5.14, the IDS detected this as an attack, increasing its false positive rate. It is necessary to increase the moving window size to five seconds before the IDS detects no attacks over the data set, as shown in Figure 5.15. This illustrates the sensitivity of the IDS to the moving window size and the necessity of characterizing the highest sustained power consumption of legitimate programs to reduce the false positive rate of the IDS.

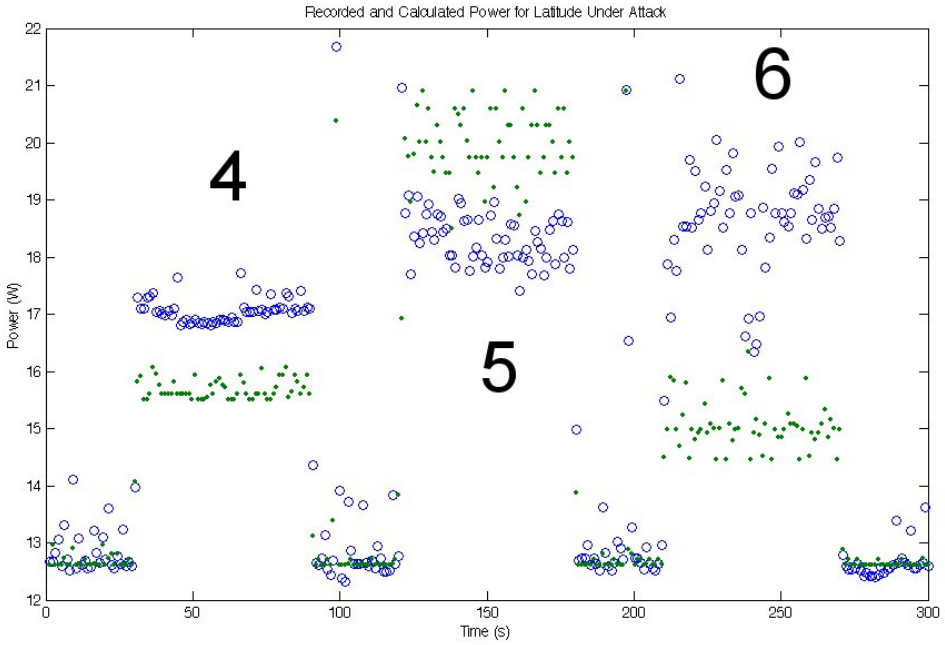

Figure 5.12: Averaged predicted power (green diamonds) and recorded power (blue circles) for Latitude during service attach and distributed attacks.

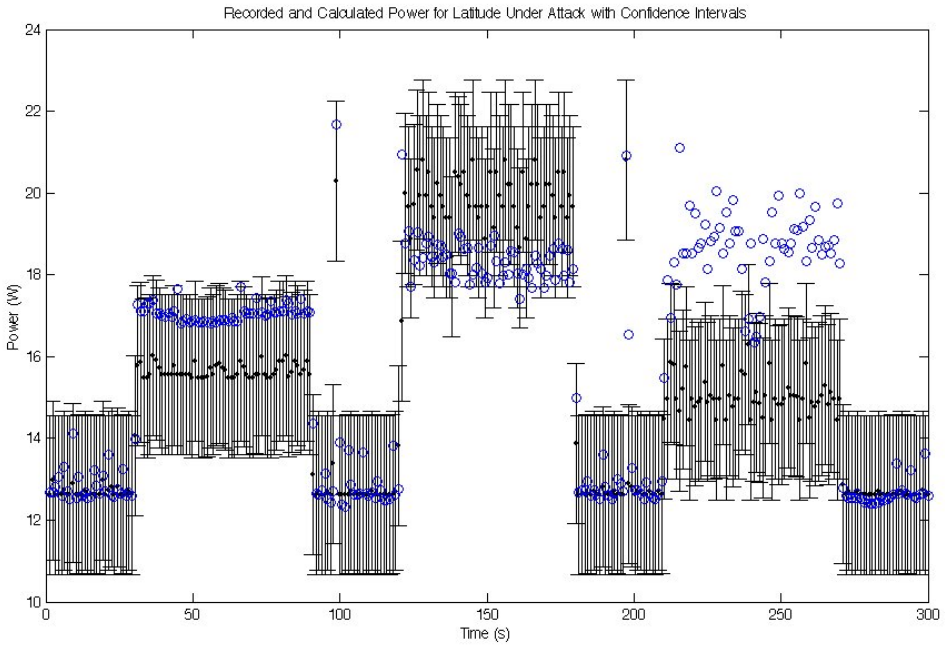

Figure 5.13: Averaged predicted power (black diamonds) with confidence intervals and recorded power (blue circles) for Latitude during service attach and distributed attacks.

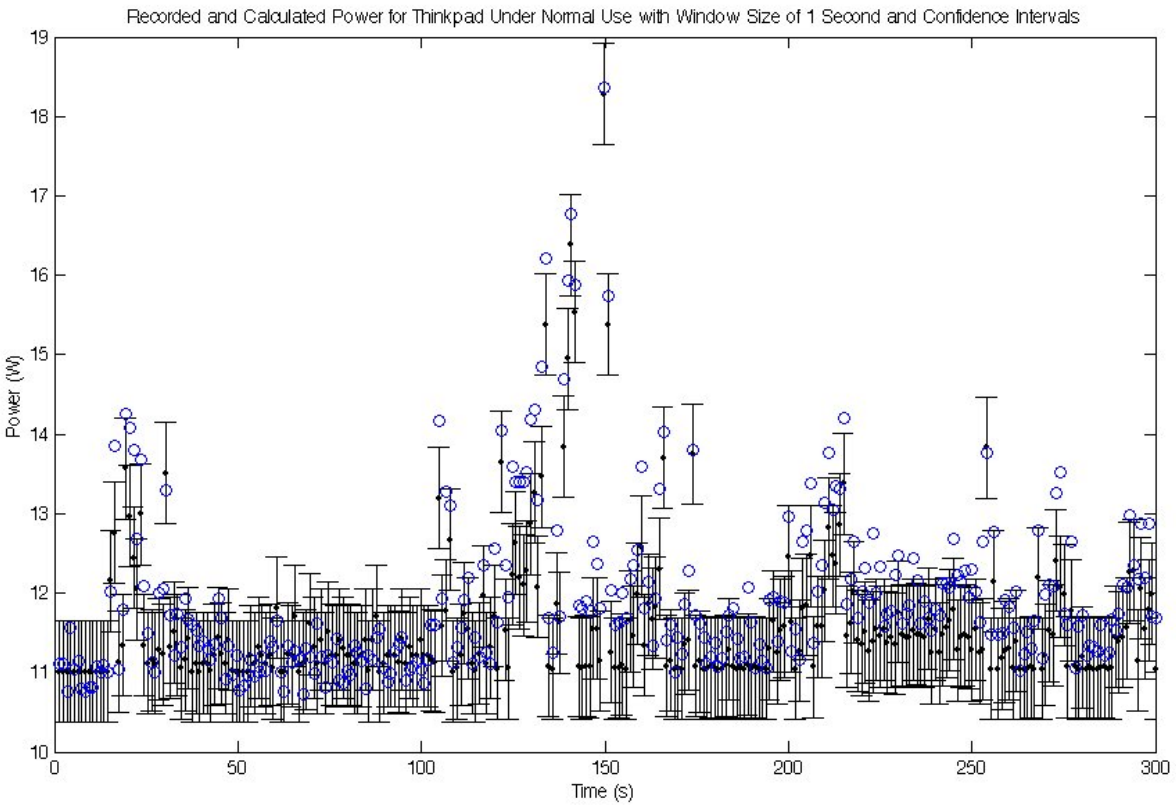

Figure 5.14: Recorded (blue circles) and predicted (black diamonds) power with confidence intervals for Thinkpad under normal use with window size of one second.

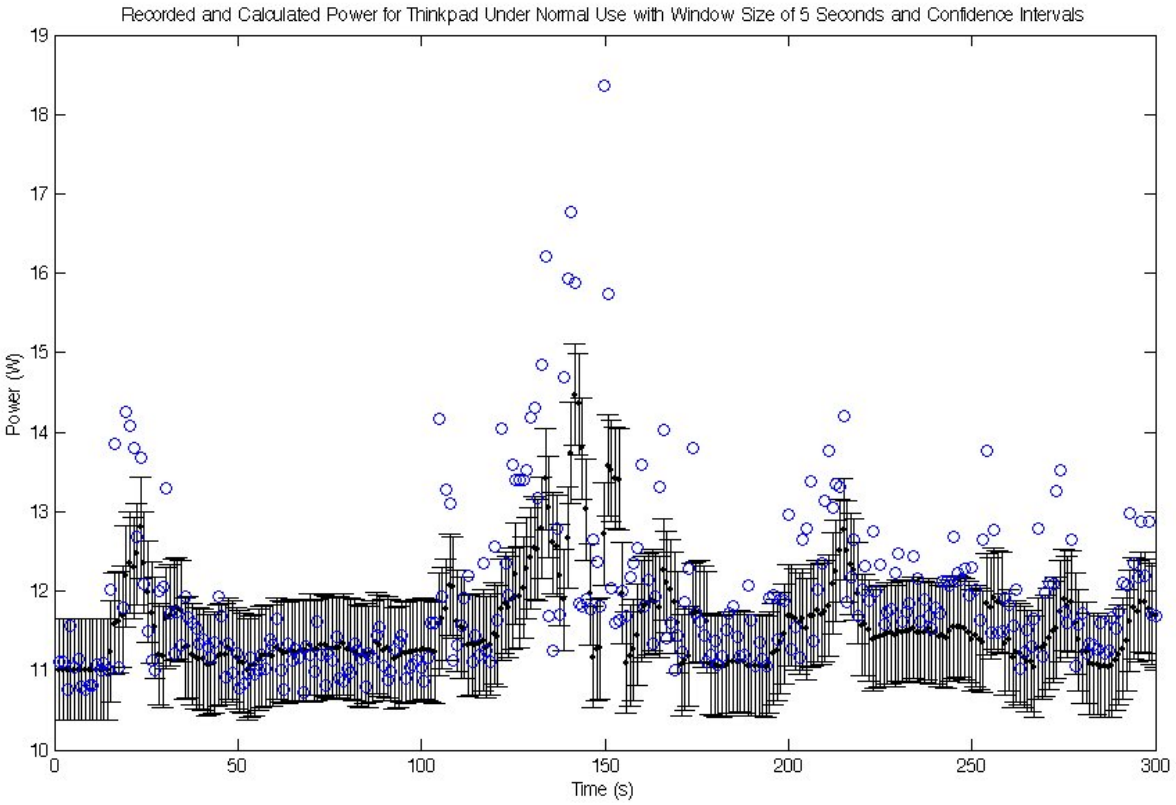

Figure 5.15: Recorded (blue circles) and predicted (black diamonds) power with confidence intervals for Thinkpad under normal use with window size of five seconds.

Chapter 6

Future Work

This chapter discusses future work that could be done to improve the accuracy and usefulness of the intrusion detection systems.

6.1 Improvement of IDS

Unfortunately, the Latitude's regression model did not perform as well as the Thinkpad's regression model. Though the model performed well under certain situations such as when using the benchmark programs, the unexplained variation of the model lead to large confidence intervals for the predicted power. The power saving technology built into the Pentium M helps to eliminate the effectiveness of battery exhaustion attacks. The combined effort of that technology with an IDS would work well to eliminating the threat of battery exhaustion attacks. The brief attempts at including temperature data showed that it would be a useful parameter to include, but its behavior was not linear. Using cross product terms in the regression model might better model its effects. Experimenting with using quadratic terms for only some of the parameters and possibly using cross product terms could prove beneficial.

Even operating at the highest priority, during periods of very high and extended processor

load, the IDS was sometimes not scheduled at the sampling interval it was supposed to be. If an operating system with realtime capabilities were utilized, the IDS could be scheduled more reliably and closer to its exact sampling interval. Generating an IDS under the Linux operating would be a worthwhile endeavor. There are embedded and mobile devices that use Linux and realtime versions of Linux have been developed. There are also drivers under Linux that provide access to further metrics such as the processor's cache performance.

If a non-malicious process normally operates at a sustained power level above the detection threshold, using a window size sufficiently large enough to cause it not to be detected as an attack could increase the false negative rate to an unacceptable level. To decrease the false positive rate of the power IDS while keeping the window size reasonably small, identifying and characterizing the normal power behavior of non-malicious processes could be done. Then, non-malicious programs that cause the system to exceed its threshold will not be flagged as attacks. This same method could be used to characterize the normal power behavior of programs that could be vulnerable to benign attacks. Then, when the program is manipulated into attacking the system, its behavior can be distinguished from its normal behavior and properly identified as an attack.

6.2 Inability of IDS to Identify Attacking Process

The intrusion detection system discussed in Chapter 5 cannot detect the new form of battery exhaustion attack identified in Chapter 3, or rather, it can detect the attack, but assigns blame on the wrong program. The power consumption of the non-malicious program will cause it to be marked by the intrusion detection system as an attack, while the attacking program will not be seen as a threat. A deviously written attack can methodically attack all vulnerable programs on a system, all the while remaining out of suspicion.

Under the windowing environments where a form of the attack was developed, a method of easily tracing the flow of events from sender to receiver was not readily available. As far as the

receiving program was concerned, all of the events received came from the window manager. A collaboration between the window manager and the IDS would allow for accountability amongst the other programs for the messages they generate. A high rate of event generation from one program to another with subsequent elevated power usage by the receiving program would follow the pattern of attack. Care must be taken to make sure cooperative non-attacking programs that communicate through such events are not immediately marked as bad. Further analysis would most likely be needed to determine if this behavior is anomalous.

While the above mentioned modifications to the window manager and IDS would detect that form of service attacks, it is not general enough to detect other possible local service attacks such as the database or desktop searches mentioned in Chapter 3. A new system of program accountability is needed to aid in tracking the services offered by programs on a system and those programs that use them. One possible solution is to use a trusted middleware to broker the services. Applications with services to offer would register with the middleware and applications wishing to use those services would send requests through the middleware. Records of service usage could then be kept. If the overhead of this tracking mechanism is large, it could be selectively employed when the behavior of the system becomes suspicious.

6.3 IDS Reaction

While the current IDS implementation can detect when an attack is occurring and identify the offending program, it currently has no offensive mechanism to stop the program's behavior. Implementing a reaction mechanism similar to that developed by Forrest et. al. [21] and discussed in section 2.3 would allow the IDS to reduce the impact of a program on the power consumption of the computing device, allowing the program to continue to operate at a reduced capacity. As no IDS has a zero false positive rate, this seems a fairer reaction than simply ending a process, perhaps wrongly. Use of the Linux operating system would allow

this scheme to be implemented by altering the operating systems scheduling mechanism.

Bibliography

[1] "Agilent 3458 multimeter data sheet," Agilent. [Online]. Available: http://cp.literature.agilent.com/litweb/pdf/5965-4971E.pdf

[2] "Agilent 66300 series data sheet," Agilent. [Online]. Available: http://www.home.agilent.com/upload/cmc_upload/All/66300series_datasheet_May04.pdf

[3] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," Chalmers University, Tech. Rep. 99-15, March 2000.

[4] J. Balthrop and S. a. Forrest, "Revisiting lisys: Parameters and normal behavior," in *Proceedings of the Congress on Evolutionary Computing*, 2002.

[5] "I8kfangui," Christian Diefer. [Online]. Available: http://www.diefer.de/i8kfan/

[6] "Symbos.cabir," Corporation, Symantec. [Online]. Available: http://securityresponse.symantec.com/avcenter/venc/data/epoc.cabir.html

[7] "Freshdiagnose," FreshDevices. [Online]. Available: http://www.freshdevices.com/freshdiag.html

[8] "The intel pentium m processor: Microarchitecture and performance," Intel. [Online]. Available: http://www.intel.com/technology/itj/2003/volume07issue02/art03_pentiumm/p10_speedstep.htm

[9] A. Jones and R. Sielken, "Computer intrusion detection: A survey," University of Virginia, Computer Science, Tech. Rep., 2000.

[10] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," *Advances in Cryptography*, pp. 388–397, 1999.

[11] J. Krishnaswami, "Denial-of-service attacks on battery-powered mobile computers," Master's thesis, Virginia Tech, 2004.

[12] W. Lee and S. Stolfo, "Data mining approaches to intrusion detection," in *Proceedings of the 7th USENIX Security Symposium*, January 1998, pp. 77–94.

[13] E. Lundin and E. Jonsson, "Survey of intrusion detection research," Chalmers University, Tech. Rep., February 2002.

[14] T. Martin, M. Hsiao, D. S. Ha, and J. Krishnaswami, "Denial-of-service attacks on batter-powered mobile computers," in *Second IEEE International Conference on Pervasic Computing and Communications*, vol. 8, no. 4, March 2004.

[15] "Openssh for windows," Michael Johnson. [Online]. Available: http://sshwindows.sourceforge.net/

[16] "Microsoft performance data counters," Microsoft. [Online]. Available: http://msdn.microsoft.com/library/default.asp?url=/library/en-us/perfmon/base/performance_data.asp

[17] J. Milton and J. C. Arnold, *Introduction to Probability and Statistics*, 3rd ed. Irwin McGraw-Hill, 1995, pp. 382–491.

[18] T. M. Mudge, "Power: A first-class architectural design constraint," *IEEE Computer*, vol. 34, no. 4, pp. 52–58, 2001.

[19] "Iperf bandwidth measurement tool," NLANR Distributed Application Support Team. [Online]. Available: http://dast.nlanr.net/Projects/Iperf/

[20] M. Satyanarayanan, "Pervasive computing: vision and challenges," *IEEE Personal Communications*, vol. 8, no. 4, pp. 10–17, August 2001.

[21] A. Somayaji and S. Forrest, "Automated response using system-call delays," in *Proceedings of Usenix Security Symposium*, 2000, pp. 185–198.

[22] F. Stajano and R. Anderson, "The resurrecting duckling: Security issues for adhoc wireless networks," in *Proceedings of the 7th International Workshop on Security Protocols, Lecture Notes in Computer Science volume 1796*, April 1999, pp. 172–194.

Appendix A

Source Code of CPU Load Program

The TestCPULoad program was writting using Microft Visual C++ 6.0, Serivce Pack 6 and uses the Microsoft Foundation Classes. The framework for the program was generated using the Project Wizard, selecting a MFC dialog based application. This appendix includes the source files that were modified from the automatically generated code.

CPULoadTestDlg.h

```
// CPULoadTestDlg.h : header file
//

#if !defined(AFX_CPULOADTESTDLG_H__D224BBDB_CB5C_4573_B15F_70010B684D11__INCLUDED_)
#define AFX_CPULOADTESTDLG_H__D224BBDB_CB5C_4573_B15F_70010B684D11__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

/////////////////////////////////////////////////////////////////////////////
// CTimeLimitCacheAttackDlg dialog

class CCPULoadTestDlg : public CDialog
{
// Construction
```

```
public:
CCPULoadTestDlg(CWnd* pParent = NULL); // standard constructor

// Dialog Data
//{{AFX_DATA(CCPULoadTestDlg)
enum { IDD = IDD_CPULOADTEST_DIALOG };
CButton c_bRun;
CEdit c_eSleepTime;
CEdit c_eRunTime;
//}}AFX_DATA

// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CCPULoadTestDlg)
protected:
virtual void DoDataExchange(CDataExchange* pDX); // DDX/DDV support
//}}AFX_VIRTUAL

private:
    int m_iCurrInterval;

// Implementation
protected:
HICON m_hIcon;

// Generated message map functions
//{{AFX_MSG(CCPULoadTestDlg)
virtual BOOL OnInitDialog();
afx_msg void OnSysCommand(UINT nID, LPARAM lParam);
afx_msg void OnPaint();
afx_msg HCURSOR OnQueryDragIcon();
afx_msg void OnRunButton();
afx_msg void OnTimer(UINT nIDEvent);
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
private:
CWinThread* m_ptAttack;
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_CPULOADTESTDLG_H__D224BBDB_CB5C_4573_B15F_70010B684D11__INCLUDED_)
```

CPULoadTestDlg.cpp

```
// CPULoadTestDlg.cpp : implementation file
//

#include "stdafx.h"
#include "CPULoadTest.h"
#include "CPULoadTestDlg.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif



#define K  * 1024
#define M  * 1024 * 1024
/* Tune TUNE for machine speed;  larger number runs slower */
#define TUNE  10 M
//#define TUNE 1 M
//#define MAXSIZE ( 1 M)
#define MAXSIZE ( 100 K)

#define INTERVAL 400
const int timeIntervals[] = {40, 80, 120, 160, 200, 240, 280, 320, 360, 400};

//#define INTERVAL 200
//const int timeIntervals[] = {20, 40, 60, 80, 100, 120, 140, 160, 180, 200};

/////////////////////////////////////////////////////////////////////////////
// Function prototypes
UINT RunCacheAttack(LPVOID pParam);
int touch_array(int *array, int num_ints, int iterations);



/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{
public:
```

```
CAboutDlg();

// Dialog Data
//{{AFX_DATA(CAboutDlg)
enum { IDD = IDD_ABOUTBOX };
//}}AFX_DATA

// ClassWizard generated virtual function overrides
//{{AFX_VIRTUAL(CAboutDlg)
protected:
virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
//}}AFX_VIRTUAL

// Implementation
protected:
//{{AFX_MSG(CAboutDlg)
//}}AFX_MSG
DECLARE_MESSAGE_MAP()
};

CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
//{{AFX_DATA_INIT(CAboutDlg)
//}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CAboutDlg)
//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
//{{AFX_MSG_MAP(CAboutDlg)
// No message handlers
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CCPULoadTestDlg dialog

CCPULoadTestDlg::CCPULoadTestDlg(CWnd* pParent /*=NULL*/)
```

```
: CDialog(CCPULoadTestDlg::IDD, pParent)
{
//{{AFX_DATA_INIT(CCPULoadTestDlg)
// NOTE: the ClassWizard will add member initialization here
//}}AFX_DATA_INIT
// Note that LoadIcon does not require a subsequent DestroyIcon in Win32
m_hIcon = AfxGetApp()->LoadIcon(IDR_MAINFRAME);

    m_ptAttack = NULL;
}

void CCPULoadTestDlg::DoDataExchange(CDataExchange* pDX)
{
CDialog::DoDataExchange(pDX);
//{{AFX_DATA_MAP(CCPULoadTestDlg)
DDX_Control(pDX, IDC_RUN_BUTTON, c_bRun);
DDX_Control(pDX, IDC_SLEEPTIME_EDIT, c_eSleepTime);
DDX_Control(pDX, IDC_RUNTIME_EDIT, c_eRunTime);
//}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CCPULoadTestDlg, CDialog)
//{{AFX_MSG_MAP(CCPULoadTestDlg)
ON_WM_SYSCOMMAND()
ON_WM_PAINT()
ON_WM_QUERYDRAGICON()
ON_BN_CLICKED(IDC_RUN_BUTTON, OnRunButton)
ON_WM_TIMER()
//}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CCPULoadTestDlg message handlers

BOOL CCPULoadTestDlg::OnInitDialog()
{
CDialog::OnInitDialog();

// Add "About..." menu item to system menu.

// IDM_ABOUTBOX must be in the system command range.
ASSERT((IDM_ABOUTBOX & 0xFFF0) == IDM_ABOUTBOX);
ASSERT(IDM_ABOUTBOX < 0xF000);
```

```
CMenu* pSysMenu = GetSystemMenu(FALSE);
if (pSysMenu != NULL)
{
CString strAboutMenu;
strAboutMenu.LoadString(IDS_ABOUTBOX);
if (!strAboutMenu.IsEmpty())
{
pSysMenu->AppendMenu(MF_SEPARATOR);
pSysMenu->AppendMenu(MF_STRING, IDM_ABOUTBOX, strAboutMenu);
}
}

// Set the icon for this dialog.  The framework does this automatically
//  when the application's main window is not a dialog
SetIcon(m_hIcon, TRUE); // Set big icon
SetIcon(m_hIcon, FALSE); // Set small icon

// TODO: Add extra initialization here

return TRUE;  // return TRUE  unless you set the focus to a control
}

void CCPULoadTestDlg::OnSysCommand(UINT nID, LPARAM lParam)
{
if ((nID & 0xFFF0) == IDM_ABOUTBOX)
{
CAboutDlg dlgAbout;
dlgAbout.DoModal();
}
else
{
CDialog::OnSysCommand(nID, lParam);
}
}

// If you add a minimize button to your dialog, you will need the code below
//  to draw the icon.  For MFC applications using the document/view model,
//  this is automatically done for you by the framework.

void CCPULoadTestDlg::OnPaint()
{
if (IsIconic())
```

```
{
CPaintDC dc(this); // device context for painting

SendMessage(WM_ICONERASEBKGND, (WPARAM) dc.GetSafeHdc(), 0);

// Center icon in client rectangle
int cxIcon = GetSystemMetrics(SM_CXICON);
int cyIcon = GetSystemMetrics(SM_CYICON);
CRect rect;
GetClientRect(&rect);
int x = (rect.Width() - cxIcon + 1) / 2;
int y = (rect.Height() - cyIcon + 1) / 2;

// Draw the icon
dc.DrawIcon(x, y, m_hIcon);
}
else
{
CDialog::OnPaint();
}
}

// The system calls this to obtain the cursor to display while the user drags
//  the minimized window.
HCURSOR CCPULoadTestDlg::OnQueryDragIcon()
{
return (HCURSOR) m_hIcon;
}

void CCPULoadTestDlg::OnRunButton()
{
    int iTime = 0;
    CString csText;

    m_iCurrInterval = 0;
    SetTimer(1, timeIntervals[m_iCurrInterval]/*iTime*/, NULL);
    c_bRun.EnableWindow(FALSE);
m_ptAttack = AfxBeginThread(RunCacheAttack, 0);
}

UINT RunCacheAttack(LPVOID pParam)
{
    int iDummy = 0;
```

```
    while (1) {
        iDummy++;
        if (iDummy > 1000000) {
            iDummy = -1000000;
        }
    }


    return 0;
}


void CCPULoadTestDlg::OnTimer(UINT nIDEvent)
{
    CString csText;
    int iSleepTime = 0;
    int iRunTime = 0;
    static int iCurrCount = 0;

    if (nIDEvent == 1) {
        // stop running the attack
        m_ptAttack->SuspendThread();
        KillTimer(1);
        SetTimer(2, (INTERVAL - timeIntervals[m_iCurrInterval]), NULL);
    }
    else if (nIDEvent == 2) {
        if (iCurrCount == (30000 / INTERVAL)) {
            iCurrCount = 0;
            m_iCurrInterval++;
            if (m_iCurrInterval == 10) {
            }
        }

        KillTimer(2);
        m_ptAttack->ResumeThread();
        if (m_iCurrInterval < 9) {
            // start the attack again
            SetTimer(1, timeIntervals[m_iCurrInterval], NULL);
            iCurrCount++;
        }
        else {
            SetTimer(3, 30000, NULL);
```

```
        }

    }
    else if (nIDEvent == 3) {
        KillTimer(2);
        TerminateThread(m_ptAttack->m_hThread, 0);
        c_bRun.EnableWindow(TRUE);
        c_bRun.SetActiveWindow();

        return;
    }

CDialog::OnTimer(nIDEvent);
}
```

CPULoadTest.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32
```

```
#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "#define _AFX_NO_SPLITTER_RESOURCES\r\n"
    "#define _AFX_NO_OLE_RESOURCES\r\n"
    "#define _AFX_NO_TRACKER_RESOURCES\r\n"
    "#define _AFX_NO_PROPERTY_RESOURCES\r\n"
    "\r\n"
    "#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)\r\n"
    "#ifdef _WIN32\r\n"
    "LANGUAGE 9, 1\r\n"
    "#pragma code_page(1252)\r\n"
    "#endif //_WIN32\r\n"
    "#include ""res\\CPULoadTest.rc2""  // non-Microsoft Visual C++ edited resources\r\n"
    "#include ""afxres.rc""         // Standard components\r\n"
    "#endif\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


//////////////////////////////////////////////////////////////////////////////
//
// Icon
//
```

```
// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDR_MAINFRAME           ICON    DISCARDABLE     "res\\CPULoadTest.ico"

/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 235, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About CPULoadTest"
FONT 8, "MS Sans Serif"
BEGIN
    ICON            IDR_MAINFRAME,IDC_STATIC,11,17,20,20
    LTEXT           "CPULoadTest Version 1.0",IDC_STATIC,40,10,119,
                    8,SS_NOPREFIX
    LTEXT           "Copyright (C) 2004",IDC_STATIC,40,25,119,8
    DEFPUSHBUTTON   "OK",IDOK,178,7,50,14,WS_GROUP
END

IDD_CPULOADTEST_DIALOG DIALOGEX 0, 0, 208, 95
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
EXSTYLE WS_EX_APPWINDOW
CAPTION "CPULoadTest"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "Run",IDC_RUN_BUTTON,93,75,50,14
    PUSHBUTTON      "Cancel",IDCANCEL,151,75,50,14
    EDITTEXT        IDC_RUNTIME_EDIT,62,13,99,14,ES_AUTOHSCROLL
    LTEXT           "Run Time:",IDC_STATIC,21,16,34,8
    LTEXT           "Sleep Time:",IDC_STATIC,17,46,38,8
    EDITTEXT        IDC_SLEEPTIME_EDIT,62,43,99,14,ES_AUTOHSCROLL
    LTEXT           "s",IDC_STATIC,165,16,8,8
    LTEXT           "s",IDC_STATIC,166,45,8,8
END


#ifndef _MAC
/////////////////////////////////////////////////////////////////////////////
//
// Version
//
```

```
VS_VERSION_INFO VERSIONINFO
 FILEVERSION 1,0,0,1
 PRODUCTVERSION 1,0,0,1
 FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
 FILEFLAGS 0x1L
#else
 FILEFLAGS 0x0L
#endif
 FILEOS 0x4L
 FILETYPE 0x1L
 FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "040904B0"
        BEGIN
            VALUE "CompanyName", "\0"
            VALUE "FileDescription", "CPULoadTest MFC Application\0"
            VALUE "FileVersion", "1, 0, 0, 1\0"
            VALUE "InternalName", "CPULoadTest\0"
            VALUE "LegalCopyright", "Copyright (C) 2004\0"
            VALUE "LegalTrademarks", "\0"
            VALUE "OriginalFilename", "CPULoadTest.EXE\0"
            VALUE "ProductName", "CPULoadTest Application\0"
            VALUE "ProductVersion", "1, 0, 0, 1\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x409, 1200
    END
END

#endif    // !_MAC


/////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//
```

```
#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_ABOUTBOX, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 228
        TOPMARGIN, 7
        BOTTOMMARGIN, 48
    END

    IDD_CPULOADTEST_DIALOG, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 201
        TOPMARGIN, 7
        BOTTOMMARGIN, 88
    END
END
#endif    // APSTUDIO_INVOKED


/////////////////////////////////////////////////////////////////////////////
//
// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
    IDS_ABOUTBOX            "&About CPULoadTest..."
END

#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
#define _AFX_NO_SPLITTER_RESOURCES
```

```
#define _AFX_NO_OLE_RESOURCES
#define _AFX_NO_TRACKER_RESOURCES
#define _AFX_NO_PROPERTY_RESOURCES

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE 9, 1
#pragma code_page(1252)
#endif //_WIN32
#include "res\CPULoadTest.rc2"  // non-Microsoft Visual C++ edited resources
#include "afxres.rc"         // Standard components
#endif

/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Vita

Daniel C. Nash

Daniel Charles Nash was born in Fort Belvoir, Virginia on May 20, 1980. With his father in the Army, he was given the opportunity as a child to travel across the United States and live abroad in Germany with his family. He enrolled in the engineering program at Virginia Tech in the fall of 1998. During his undergraduate career, he participated in the cooperative education program with Lockheed Martin Global Telecommunications (COMSAT Laboratories), gaining valuable work experience in his field. He earned a Bachelor of Science in Computer Engineering in 2003 and continued his education at Virginia Tech working towards his Masters of Science in Computer Engineering. He was given the opportunity to publish a paper on his work in the Second IEEE International Workshop on Pervasive Computing and Communication Security titled Towards an Intrusion Detection System for Battery Exhaustion Attacks on Mobile Computing Devices. His interests include radio control aircraft and computer networking and security.