Modeling Systems Using Side Channel Information

By

Bogdan Copos
B.S. (The College of New Jersey) 2012

Dissertation

Submitted in partial satisfaction of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Office of Graduate Studies

of the

University of California

Davis

Approved:

Dr. Sean Peisert, Chair

Dr. Matthew Bishop

Dr. Karl Levitt

Committee in Charge

2017

Dedicated to my sister and parents

Contents

List of Figures vi
List of Tables viii
Abstract ix
Acknowledgments x

1 Introduction **1**
1.1 Background 1
1.2 Problem Statement 2
1.3 Thesis Statement 3
1.4 Approach 3
1.5 Motivating Example 4
1.6 Definitions 7
1.7 Organization of the Dissertation 9

2 Related Work **10**
2.1 Power Analysis 11
2.2 Traffic Analysis 16
2.3 High Performance Computing Activity 19
2.4 Information Channel Bandwidth 21

3 Early Work **23**
3.1 Modeling User Behavior for Improved Network Efficiency in Android Applications 23
3.2 Modeling Input Protocol for Unknown Binaries using Hardware Performance Counters 28
3.2.1 Finding Input 28
3.2.2 Method Architecture 30
3.2.3 Experiments and Results 34
3.2.4 Conclusions on Finding Input Using Side Channels 36

4 Monitoring High Performance Computing Platforms **37**

4.1 Security in High Performance Computing Platforms 37

4.2 Power Analysis 40

4.2.1 Initial Experiments and Observations 41

4.2.2 HPC Experiments 43

4.2.3 Discussion and Future Work 60

4.3 I/O Analysis 62

4.3.1 Darshan 62

4.3.2 Data Collection 62

4.3.3 Methodology 64

4.3.4 Findings 65

4.3.5 Results 69

5 Monitoring Internet of Things Platforms Using Side Channels **72**

5.1 Background 72

5.2 Experiment Setup and Devices 74

5.3 Methodology 75

5.4 Results 77

5.5 Frequency Analysis 81

5.6 Conclusion 83

6 Side Channel Theory **84**

6.1 Definition 85

6.2 Model Definition 87

6.3 Side Channel vs. Program 89

6.3.1 Operation-To-Side-Channel Conversion 89

6.3.2 Side Channel Resolution 93

6.3.3 Putting It All Together 95

6.4 Information Loss 95

6.5 Noise 96

6.6 Turing Machine Side Channel Example 97

6.7 Limitations and Discussions 101
6.7.1 Limitations 101
6.7.2 Comparison with Entropy-Based Models 105

7 Towards Protecting Against Side Channels **106**

8 Conclusion **111**
8.1 Summary 111
8.2 Limitations and Future Work 113
8.3 Recommendations 114
8.4 Closing Remarks 116

References **117**

Appendices **124**

A HPC I/O Analysis: Darshan Features **125**

B Side Channel Theory **127**
B.1 Turing Machine Binary Counter Program 127

List of Figures

3.1 Screenshot of user interface of the demo Android application, displaying "friends" locations across a geographical map. 26
3.2 Diagram depicting the architecture of the finding input process. 30
3.3 Diagram visualizing the input strings being constructed throughout five consecutive time stages of the execution of InputFinder. 32
3.4 Diagram visualizing the user protocol state machine of one of the CGC programs, with the backdoor highlight in red. 35

4.1 Diagram of current and voltage during the execution of OpenSSL SHA-1 hashing. 41
4.2 Diagram of the CPU, memory, and current behavior of a HPC compute node during two stress tests 42
4.3 This figure depicts the mangitude of current over time during the execution of the Integer Sort (IS) and Fourier Transform (FT) benchmarks of the NAS Parallel bench suite. 44
4.4 Diagram visualizing the mean-shifted distribution of the current magnitude time series for each of the benchmarks. 45
4.5 Diagram describing the architecture of the time series analysis framework 46
4.6 Spectrograms of current magnitude time series representing two of the NAS Parallel Benchmarks, the Block Tri-diagonal Solver (BT) and the Lower Upper Gauss-Seidel Solver (LU). 48
4.7 Diagram depicting the variation in the classification success rate across the windows of the MiniFE program. 53
4.8 Diagram visualizing the impact of noise on the average accuracy across all programs and a comparison of the classification results with random guessing and mutual information 56
4.9 Diagram visualizing how noise affects the recall (top) and precision (bottom) of each individual program. 57

4.10 Diagram of the number of instructions executed over time, during the execution of two NAS parallel benchmarks, (a) Conjugate Gradient program and (b) Fourier Transform program 59
4.11 Boxplot and swarmplot representing the distribution of all 53 features across all 331 Darshan logs collected. 65
4.12 Diagram visualizing the I/O behavior of four different executions of the *track3p* application. 66
4.13 Radar plots representing two *Structured Grid* programs. 67
4.14 Radar plots representing I/O behavior of seven computational dwarfs. Each vertex of the radar plot represents a different feature. 67
4.15 Histogram of pairwise Euclidean distances between vectors representing programs of the same computational dwarf. 69

5.1 Diagram visualizing connections and their size (bytes sent) made by the Nest Thermostat to prominent destination over the span of three days. 76
5.2 Diagram showing NTP requests made by the Nest Thermostat over the span of two days (*Auto Away* periods are marked by vertical bars) 80
5.3 Diagram showing frequencies of connection time series during mode transitions. The windows before and after the events contain no relevant packets, hence no frequency components. 82
5.4 Spectrogram of connections over a single day. 83

6.1 Diagram visualizing the case of n distinct side channel values and n distinct state transitions. 90
6.2 Diagram depicting the case where each unique side channel value maps to multiple distinct state transitions. 91
6.3 Diagram visualizing the tape head location and contents at the beginning and end of the binary program execution 98

List of Tables

4.1 Table listing classification results of clean current magnitude samples using the full sample approach. 50
4.2 Table describing the results of classification of noisy current magnitude samples using the window sets approach. The scoreless programs have execution times too short for the selected window set size. 54
4.3 Table describing the breakdown of I/O libraries used across 54231 Darshan logs from 108 total applications. 63
4.4 Table listing 18 most commonly executed scientific codes and their corresponding computational dwarf. 64

Abstract

Modeling Systems Using Side Channel Information

Side channel analysis is the process of examining information leaked by a computing device during use, and leveraging such data to make inferences about various aspects of the system. Historically, side channels have been exploited for malicious purposes, from inferring sensitive data to infringing on the privacy of users. For example, power consumption has been exploited to reveal secret cryptographic keys, and features of wireless network traffic have been leveraged to reveal web browsing activity of a user. The goal of this dissertation is not only to explore the potential of using side channels to determine what types of activity a computing system is engaged in but also study the relationship between the operations performed by the system and the side channel.

In this dissertation we present two key concepts: the application of side channel analysis for security and privacy purposes, particularly for monitoring systems, and the development of a model for defining the relationship between side channel information and the operations performed by the system. The empirical studies presented in this dissertation demonstrate that side channel information can be leveraged to monitor the behavior of systems and describe advantages for doing so over alternative methods. In addition, we outline a model that describes how the operations performed by a system are represented in side channel information and how the information loss can be estimated. The goal of these two directions is to expand the understanding of side channels, their benefits and drawbacks, from both a practical point of view as well as theoretical. Our work shows how the outlined model can measure the information loss in side channels while our empirical studies show that despite information being lost, in many cases, side channels contain enough information to successfully monitor the behavior of systems and provide a non-intrusive, minimal impact method for doing so.

Acknowledgments

Throughout my graduate school years, I have encountered many people, who have impacted me in various ways. I am thankful for my adviser Dr. Sean Peisert. He has been extremely supportive and encouraging and I am immensely grateful for the opportunity to work with him. The experience has been incredibly rewarding and insightful. For all of their support and insightful conversations, thank you to Professor Matt Bishop and Professor Karl Levitt. It's not often you have the honor and privilege to spend time and pick the brains of such brilliant people.

To my family, without you I wouldn't be me and I wouldn't be here. Thank you for all of you support and encouragement throughout the years. To my sister, thank you for always being a role model and inspiring me to excel. Bunica, mersi pentru incurajare.

I'd also like to thank my friends, especially Henry Kvinge, Jonathan Ganz, Axel Saenz Rodriguez.

This research used resources of the National Energy Research Scientific Computing Center and was supported in part by the Director, Office of Science, Office of Advanced Scientific Computing Research, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. Other sources of funding include the National Science Foundation and funding from theResearch Initiative for Scientific Enhancement (RISE) Program. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the author and do not necessarily reflect those of the sponsors of this work.

This dissertation includes work presented in the following papers, with full permission from all authors. The papers have either already been published or are currently in submission.

- "Inputfinder: Reverse Engineering Closed Binaries Using Hardware Performance Counters," Bogdan Copos, Praveen Murthy, in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW)*, Los Angeles, CA, 2015.

- "Is Anybody Home? Inferring Activity From Smart Home Network Traffic," Bogdan Copos, Karl Levitt, Matt Bishop, Jeff Rowe, in *Proceedings of the IEEE Computer Society Security and Privacy Workshops (SPW)*, San Jose, CA, 2016.

- "Monitoring High Performance Computing Activity via Power Analysis" (in preparation)
- "Side Channel Bandwidth: Studying The Relationship Between Side Channel Information and System Behavior" (in preparation)

Chapter 1

Introduction

> *"What we call chaos is just patterns we haven't recognized. What we call random is just patterns we can't decipher. What we can't understand we call nonsense. What we can't read we call gibberish."*
>
> — Chuck Palahniuk, "Survivor: A Novel"

1.1 Background

Side channel analysis is the process of examining information channels leaked by a computing device during periods of activity and leveraging such data to make inferences about various aspects of the system and its activity. Examples of side channels include both streams (or waves) of physical particles, such as current, radiation and even sounds, as well as other types of data such as timing information. Unlike other types of information, side channels are naturally occurring and often cannot be countered. As such, side channel analysis presents a unique, non-intrusive, and often covert method of extracting potentially private information from a system.

At first glance, side channels are noise: "random" signals naturally leaked by various entities. Yet, as the quote above suggests, noise is often information we cannot comprehend and have yet to decipher. It did not take long for people to notice patterns in side channel information. Studying these patterns led to discoveries that side channel information is often a reflection of the system's activity. Historically, side channels have been exploited for

malicious purposes from inferring sensitive input to a program or system, to infringing on the privacy of users. Dating back to 1950s, researchers proved that cryptography can be compromised by analyzing data leaked from cryptographic devices. Academic publications since have demonstrate how side channels can be leveraged for various purposes, from weakening cryptographic implementations by inferring keys [1, 2, 3, 4], to identifying users' online browsing activity [5, 6, 7, 8], or even identifying keystrokes [9] or content of printer jobs via acoustic emanations [10].

Undoubtedly side channel analysis has both security and privacy implications. Yet despite the historic connotation as an attacker's tool, side channel information is "dual use". That is, since it discloses information about the system and its activity, side channel information can also have positive applications. Regardless of the application, it is important to study the full potential and drawbacks of side channel information, for both security and privacy reasons.

1.2 Problem Statement

With the increase in cyber-attacks, cyber security monitoring has become an essential component of any institutions' security framework. Researchers and computer scientists have developed a variety of tools for collecting information about the activity of a system. Network monitoring frameworks such as Bro [11] can be used to analyze incoming and outgoing network traffic. System-level utilities also exists for identifying processes running as well as other system attributes.

Cyber security monitoring is mostly ad hoc. To effectively monitor a system for security and privacy reasons, every system component should be carefully monitored. However, most security monitoring tools focus on specific attributes of a system. This has several implications. First, such detailed monitoring produces massive amounts of data whose processing is costly. Additionally, most existing monitoring tools are intrusive and impose an overhead to the system. Furthermore, intrusive monitoring tools increase the complexity of the system maintenance process.

Ultimately, the purpose of monitoring tools is to record or generate data which provides insight into the activity of the system. While side channels have historically been used for

nefarious purposes (violate the privacy of users or break security mechanisms), we examine the potential of using side channels to determine what types of activity a computing system or device is engaged in. This is useful in several scenarios including the monitoring of corporate networks in which privacy has been waived and administrators are concerned with inappropriate or illegal activity or in cases where the software operating on devices is not controllable by device owners. Side channels have two major advantages over other forms of monitoring due to the fact that they do not require software running on the system being monitored: users cannot easily tamper with the collection and the collection of data may have no (negative) impact on a system's performance. Additionally, some side channels, such as power, have the advantage of providing insight into both the behavior of the software and hardware. As such, exploiting side channel information for security and privacy related monitoring presents as a natural choice.

1.3 Thesis Statement

In this dissertation, we present a model and several empirical studies of leveraging side channel information to monitor the behavior of a system for both privacy and security reasons. Our model formally defines the relationship between side channel information and the activity of a program. Unlike previous efforts which focus on information flow, in our approach, by defining the relationship between side channel information and the activity of a program, researchers can estimate how accurately a side channel reflects system activity and specifically, the amount of information lost in the side channel.

1.4 Approach

Our goal is to expand the understanding of side channels and their benefits. Previous published efforts have successfully demonstrated only one aspect of side channels. Specifically, prior research has shown that based on the side channel information, it is possible to infer the input to a particular program. That is, previous research shows that there exists a flow of information from the input to the system and the side channel. In this dissertation, we seek to study side channels in more depth, particularly focusing on the relationship between the behavior of a program and a side channel.

In contrast to previous efforts, we study how control flow of the program is represented in the side channel. This relationship can have many security and privacy implications. As presented here, measuring this relationship can enable the leveraging of side channel information as a method for building security monitors. Research questions concerning the privacy and security implications of using side channel information to monitor the activity of systems, including spying and reverse engineering of programs, are beyond the scope of this dissertation and remain as future research problems.

To achieve our goal, we approach the problem in two ways. First, through empirical studies, we expose previously uncovered benefits of using side channel information in various settings, including the "Internet of Things" and high-performance computing platform. In our experiments, we apply statistical analysis, supervised and unsupervised machine learning algorithms to interpret and classify side channel data streams. Empirical studies, while valuable, are case specific and fail to generalize the nature of the relationship and identify important characteristics. In other words, the amount of information that can be interpreted from the side channel depends on the frequency of the data collection and ultimately on the relationship between the operations performed by the system and their footprint on the side channel. As such, we outline a model designed to formally define this relationship.

Our model builds upon automaton theory to formalize the generation of side channel information and help describe the relationship between the activity of a system and the side channel. Specifically, our model allows us to identify the factors which affect the resolution of this relationship and identify bounds on the resolution of information provided by the side channel (with respect to the activity of the system). As a result, by applying our model, researchers can determine how closely the side channel represents the control flow of the program. Our model does not make any assumptions about the type of side channel, the nature of the program, the hardware executing the program, or how side channel information is collected (except the frequency at which information is collected).

1.5 Motivating Example

As a motivating example, consider the problem of monitoring Internet of Things devices. *Internet of Things* (IoT) refers to a system of Internet-connected, networked, embedded

computing devices. These embedded devices are being inserted into everyday objects and often play the role of sensors and actuators. For example, at the time of this writing, there are a plethora of Internet of Things devices in many different settings, from homes (e.g., "smart locks") to industrial settings including the power grid and even manufacturing facilities. Many of these devices can be controlled over the Internet, often via a mobile application. The Internet connectivity and intrusive nature of these devices raise serious privacy and security concerns. However, monitoring these devices and assuring security properties is non-trivial and users are obligated to trust the developers and manufacturers of these devices.

For the purpose of this example, let us consider a set of Internet of Things devices all equipped with several hardware components including a central processing unit (CPU), random access memory, limited physical storage drive, and a network card. Let us also assume that a user has several of such devices, each with a different purpose, from a variety of different manufacturers. From the user's perspective, there is not much the user can do to verify if the devices have been compromised or even check what the devices are doing. The user must trust the manufacturer and companies involved in the production chain of the devices.

However, even if most devices are implemented correctly and securely, Internet of Things devices can communicate with each other through their environment, even when devices are "incompatible" and do not utilize the same protocols. Without verifying all combinations of devices from all manufacturers in every possible environment, it is difficult to guarantee security of IoT platforms. On the other hand, monitoring devices across manufacturers is challenging. Often devices do not use the same protocols or even technologies and are therefore incompatible. A security monitoring device capable of deciphering the protocol of a manufacturer may be useless for monitoring devices of another manufacturer.

The work presented in this dissertation attempts to demonstrate the side channel analysis can provide assistance in such scenarios. In particular, we believe side channel analysis can be leveraged as a method of non-intrusive security monitoring of various devices and systems.

In our example, despite the devices communicating different protocols, using different technologies, and perhaps even different hardware components, all embedded devices have

shared characteristics. First, it is important to note that every hardware component consumes electricity. In addition, due to physical characteristics of the circuitry (such as electrical resistance), these devices also emit heat and electromagnetic radiation. Such emissions are examples of off-system side channels. Side channels only exhibit themselves when the device is running. Off-system side channels are streams of information that can be collected without interacting with the system. In contrast, on-system side channel information is collected by interacting with the device. It is important to note that side channels are not limited to streams of physical particles. Even timing information can serve as a side channel. In our example, a side channel shared by many IoT devices is the wireless network traffic. Even if encrypted, features of the traffic such as size (in bytes) and even number and frequency of transmissions can be used to characterize the behavior of a device. Some examples are discussed in the background chapter, Chapter 2.

To extract more meaningful information from side channels, it is important to note that the fluctuations in side channel information depend on the amount of work performed by the hardware components. Consequently, side channel information is an indication of the activities performed by the computer. By analyzing and extracting features from the fluctuations in the side channel over time, it is possible to generate profiles describing various states of the system.

For example, consider a "smart" thermostat IoT device. During its life-cycle, the device performs several tasks. It may check the temperature and activate the heat-ventilation-air-condition (HVAC) system accordingly. It may check online weather services for weather predictions. It may contact time servers to get an accurate time. The behavior of these default, periodic actions may look different than the behavior that occurs when the user sends it a command to adjust the temperature. The behavior may different in several ways. It is possible that the network traffic generated when a user sends a command is represented by a different set of packets than the normal periodic traffic. Again, differences may be observed in size and frequency. Similarly, it may be possible to differentiate between states by looking at the power consumption of the device. For example, the thermostat may utilize more power when the user changes the temperature since it has to receive the command, process, verify the temperature, and possibly activate the HVAC system. On the other

hand, the power consumption at rest or during a minimal action such as checking time may differ. Even memory operations can serve as an indicator of what the system is doing. For example, if the device wants to save the user's preferences any time they adjust the settings or temperature, this may result in a different pattern of memory operations.

It is important to note, however, that the amount of information leaked about the system's activity depends on the relationship between the operations performed by the system and the side channel. This relationship is described in more detail in Chapter 6 and it plays a key role in distinguishing between states or operations of the system by solely relying on side channel information.

One may argue that there are other methods for achieving the same goal. Our work does not mean to diminish alternative approaches. We also believe our approach can be used in supplement to other methods. However, we do argue that the approach presented in this dissertation has several advantages over alternatives. First, our approach relies on side channels which are readily available and naturally emitted by the system. As such, there is no overhead in generating the data. Additionally, side channels, especially off-system side channels, allow such monitoring to be done non-intrusively and are agnostic of the underlying software and hardware. In a heterogeneous environment, such as that found in IoT platforms, monitoring devices across manufacturers can only be done by relying on such side channels (without the cooperation and coordination of manufacturers). In other words, all electrical components utilize power and share this side channel (with the exception of battery powered devices, where what is observable externally does not accurately represent their consumption). While the analysis details may change, the overall approach is not dependent on the characteristics of the hardware or software of the device.

1.6 Definitions

We define a *side channel* as an information channel generated by the physical implementation of a system during the processing of some task. Side channels exhibit themselves in many different forms. Although historically side channels present in physical particle streams (e.g., current, radiation, acoustics) have been most prevalent in literature, it is important to note that side channels exhibit themselves in many different forms. Other side channel types

include encrypted network traffic and even timing information.

Side channels are closely related to *covert channels*. Covert channels are information channels that are not intended for communication or information transfer but are used by two parties to covertly communicate. As such, side channels can be thought of as a one sided covert channel, where one party is the system and the other is the user recording the side channel information.

Extracting useful patterns out of information channels is done through various *data analysis* techniques. There are two types of data analysis: statistical analysis and machine learning. Statistical analysis leverages statistical features of the data to model it. Machine learning is an analysis method that involves the use of artificial intelligence to create models representing or predicting the data. Machine learning covers a variety of algorithms split into two main categories: supervised and unsupervised. Supervised machine learning algorithms are trained using labeled data. Unsupervised machine learning algorithms apply various functions to the data in order to infer the underlying structure of the data. In this dissertation, both statistical modeling and machine learning techniques (supervised and unsupervised) are applied.

Shannon entropy is an information theoretic metric used to measure the amount of information in a noisy channel. Shannon entropy provides a method for describing the unpredictability of information in each message and is defined by the following equation (1.1):

$$H(X) = -\sum_{i=0}^{n} p(x_i) \log_b p(x_i) \tag{1.1}$$

The equation defines that the entropy of a data stream is defined by the sum of the probabilities of each outcome, $p(x_i)$ multiplied by the logarithm of probabilities for each outcome. Using this equation, it is possible to measure the amount of information contained in an information channel. Consequently, Shannon entropy can be used to determine the minimum number of bits needed to encode a string of symbols. We use Shannon entropy to estimate the amount of information in side channel data and how it relates to the information describing the control flow of the system.

Information flow is defined as the transfer of information from one entity to another. As mentioned above, prior research efforts study the information flow between input to a system

and the side channel. In contrast, *control flow* is the sequence of steps (e.g., instructions, functions, etc.) of an imperative program p during an execution with some input x.

Automata theory is a theoretical branch of computer science that studies abstract models of machines and the computational problems such machines are capable of solving. One application of automatons is to allow researchers to describe and analyze the dynamic behavior of systems. There are two classes of automatons based on their behavior: deterministic and non-deterministic. A *probabilistic automaton* is a generalization of a non-deterministic finite automaton, where the transitions between states have probabilities associated with them. As a result, the transition function becomes a transition matrix. In our work, we rely on automata theory to describe the generation of side channel information and how the information relates to the operations performed by the system.

1.7 Organization of the Dissertation

This dissertation is organized as follows: Chapter 2 discusses related work and how the work presented here differs. Chapter 3 presents some of the early work done in the exploration of additional benefits of side channel information. Chapter 4 studies the use of various side channels to monitor a High Performance Computing platform. Chapter 5 demonstrates how side channel information can be used to monitor Internet of Things devices. Chapter 6 presents a model of side channel information and how it relates to system activity. Chapter 7 proposes future work, both with respect to interesting empirical studies as well as plans to apply our model for defensive purposes. Finally, Chapter 8 presents our conclusions.

Chapter 2

Related Work

Side channels have a long history. Unlike in the digital age of the 21st century, finding and gathering information in the past was difficult. As a result, people often found themselves leveraging whatever little information was available to make inferences. The skill of inferring secret and potentially private data from leaked information became especially valuable for attackers. It presented a non-intrusive method of revealing private information about a target without their consent and also often without the person or system gathering the information having to expose themselves.

This negative connotation is portrayed in some of the earliest document side channel attacks. For example, in 1956, the British intelligence agency MI-5 tapped the Egyptian embassy telephone lines and tried to infer the settings of the Hagelin encryption machine based on the sounds. Similarly, during the first World War, soldiers would measure the power flowing over on-field telephone wires to infer messages exchanged by enemy parties. This continued over time and evolved to exploitation of other side channels, As seen in the early examples described above, side channel information became known as technology used by attackers to spy and infringe on the privacy of others. This reputation of side channel analysis persists even among recent academic research.

Since their initial discovery, researchers have explored side channels in various settings. While side channels have many different applications, some of their main applications are in power analysis and network traffic analysis. Due to their relevance to the work presented in this dissertation, we will provide an overview of related works in those areas as well

as previous efforts in monitoring high performance computing (HPC) platforms and works related to the study of side channel bandwidth.

2.1 Power Analysis

Electrical power is one of the earliest side channels identified. As such, power analysis has a solid foundation of previous efforts. The majority of the efforts can be divided into two categories, by their use. The first category encompasses bodies of work that leverage power to infer information about a variety of systems.

Kocher et. al. demonstrate that the power consumption of encrypting devices during computation is strongly correlated to the instructions executed and the state of its internal registers [12]. Specifically, using relatively simple devices, the authors show that voltage differences can be sampled at frequencies as high as 1GHz. Using the Data Encryption Standard (DES) symmetric-key encryption algorithm as an example, the authors show how both simple power analysis (SPA) and differential power analysis (DPA) can yield information about a device's operations and even cryptographic key values. In simple power analysis, variations in the power consumption measurements collected during a cryptographic operation are directly interpreted and mapped to certain parts of the operation. In contrast, differential power analysis looks at power consumption measurement collected both during and before (or after) the operation occurs and uses signal processing techniques to filter the noise from the signal and build a model for the operation. In the paper, the authors demonstrate that given a set of power consumption measurements taken during a cryptographic operation, it is possible to visually identify the 16 DES rounds. The paper also describes that at a 3.5714 MHz sampling rate, it is possible to differentiate between instruction types. The authors argue that while information about the different stages of the DES can be obtained, SPA does not expose any information about the key. However, the authors show that unlike SPA, DPA is not hindered by low power consumption variations in hardware implementations of symmetric cryptographic algorithms and can be used to infer key material.

Similar to Kocher's work, Carmeli et. al. take advantage of bugs in hardware and power consumption of a device to analyze its operations at various stages and retrieve secret information [13]. The authors expand the work of Boneh et. al. [14] who show that anomalous

side effects of hardware bugs, which can be considered side channels, may be leveraged by an attacker to induce faults. Boneh et. al. provide a theoretical model for breaking cryptographic implementations which use the *Chinese remainder theorem* by leveraging random hardware faults. On the other hand, Carmeli et. al. show in practice how the Intel division bug, a bug which produces slightly inaccurate results on some inputs, can be leveraged to extract the secret key used during the decryption of a cipher-text using the RSA public-key cryptographic algorithm.

The second type of power analysis efforts originate from the rapid development of cities and towns throughout the world. As populations grew and energy demand increased, electric companies have become extremely concerned with the limitations of the power grid infrastructure. One way of relieving the stress on the power grid is by managing the distribution and use of power more efficiently; yet considering the dynamic nature of power demand, such a task is nontrivial. To combat such issues, researchers started using power analysis methods to monitor and understand how energy is being used by households. This area of research became known as *non-intrusive load monitoring* (NILM) and it describes methods for generating fingerprints for various electric loads in a given household. One of the earliest efforts is that of Hart et. al. [15]. The authors describe in detail NILM techniques and determine which should be considered in future studies. Specifically, the authors apply signal processing techniques on complex power (real and reactive power) of the total loads to estimate the number of and nature of the individual loads. The authors introduce the *switch continuity principle* which states that "in a small time interval, [...] only a small number of appliances change state in a typical load" and build their NILM algorithm based on this principle. Their approach analyzes the power traces for significant changes in the current or voltage. Given the time and size of these changes, the authors can determine which appliance is turned on (or off) or in use, given prior knowledge of their models. Although their prototype only uses an on/off model, the authors also describe two other appliance operational models, which are finite state machine and continuously variable. Using these three models, the authors argue that it is possible to adequately represent multiple appliance types. The paper also discusses the intricacies of utilizing different methods and data types (e.g., current versus voltage versus power) for generating signatures and even considers the

NILM in the context of a communication model, where appliances are "transmitters" with the wiring as the communication "channel".

While the previously discussed work analyzes the aggregated load, other prior efforts explore energy disaggregation techniques for identifying individual electric loads of a household from the total aggregated load.

Zia et. al. use the framework of hidden Markov models (HMM) to capture the structure of individual power loads of various appliances [16]. To disaggregate a total load, the authors merge permutations of previously learned individual models into a single HMM and compare it to the HMM of the total load. The generation of a HMM has two stages: structure modeling and parameter estimation. Structure modeling is concerned with defining the states, whereas the parameter estimation step is responsible for estimating the transition probabilities. While the parameter estimation is performed automatically using training data, the authors manually predetermine the topology of the states.

While a hidden Markov model can capture the power consumption as a stochastic process, it is hampered by changes in the operation cycle of an application (e.g., due to seasonal changes or devices with continuously variable loads). To overcome such limitations, Zhong et. al. use a new model called an interleaved factorial non-homogenous hidden Markov model to more accurately model how appliance usage varies over the day [17]. The non-homogeneous property of their model allows for transition probabilities between states to change over time. The interleaved feature imposes a constraint that at most one chain changes state between any two consecutive time steps. To test their model, the authors used a set of power traces sampled every 2 to 10 minutes, from 251 different households, over the span of multiple days. Their model performs best when compared to using a factorial hidden Markov model, a factorial non-homogeneous hidden Markov model, or an interleaved hidden Markov model.

To encourage future exploration of NILM, research have also produced public data sets for testing of energy disaggregation techniques [18, 19] as well as open source toolkit specifically designed to enable comparing of energy disaggregation techniques [20].

Thus far, the prior work mentioned focuses on identifying when and which devices are in use in a given household. Researchers have also worked on determining *how* devices are

used and infer activity of residents in a household.

For example, Bauer et. al. [21] build a simple device, iSensor, to measure current and induced voltage and show that by placing such a device between the plug of an appliance and the socket, it is possible to identify various activities such as water boiler use, fridge door opens, and the use of many kitchen appliances. The device measures the current consumption using electromagnetic induction and uses an analog-to-digital converter (ADC) to digitalize the induced voltage. The authors use ADC peak values and build feature vectors for each device using statistical values such as sum, maximum, minimum, average and variance. Using a feature vector of size five the authors are able to distinguish between different devices and their operating modes. For example, based on the ADC values and the duration, the authors are able to determine both if the water boiler is running and how much water is being heated by the appliance. To evaluate their approach the authors ask several users to utilize the appliance in different ways while the appliances are being monitored. Using the measured data and prior observations from testing of the appliances, the authors try to correctly recognize the actions performed by the users. Their evaluation reports accuracy ranging from 66% to 100% depending on the appliance and use mode.

Researchers at Cornell University and the University of California Berkeley leverage the demand-response property of the smart grid to infer private information about the residents of a household [22]. Specifically, the authors show that using a real (or active) power monitor with 1 Hz sampling frequency and 1-watt resolution, it is possible to estimate the presence/absence, sleep/wake cycle, appliance use of the residents, as well as other events such as showers and even meal preparations. To achieve this, the authors use a NILM algorithm that analyzes the power consumption data and performs edge detection. Once switch events (i.e., appliance turning on/off) are identified using edge detection, their approach uses cluster matching to classify each switch event against a database of load signatures. With the events labeled, a behavior extraction routine infers the activity of the residents. To determine the accuracy of their approach, the authors compare their results against camera data. While their work assumes that the attacker has a list of the appliances present inside the household as well as a database of on/off fingerprint events for each of the appliances, publicly available libraries exist with generic appliance fingerprints that can be used to match

against an unknown load signature.

Clark et. al. [23] use load monitoring to identify web pages accessed by users within a household. Web browsers often use hardware acceleration to aid in the rendering of rich user interfaces. Considering this, the authors leverage information about the power consumption during the downloading and rendering of websites due to changes in the graphics display and use this information to infer the web browsing activity of users. Specifically, the authors use a sensor to monitor AC current consumption and transform these current time series in the frequency domain using Fourier transform. To build feature vectors, the Fourier transform of each current magnitude trace is split into 500 segments each 250 Hz wide. Each segment represents the presence of the signal within one 250 Hz slice of the spectrum. These feature vectors are then used to train support vector machines (SVMs). Using two different computers (a laptop and a conventional desktop), the authors tested their approach by visiting 50 popular webpages. After training the SVM using 45 samples per website and testing on another 45 samples, the classifier achieves 87% accuracy. The authors' approach also shows promising results in various scenarios such as lower sampling rate of their measuring device (from 250 kHz down to 100 kHz), accessing webpages using a VPN service, accessing webpages via wired and wireless connections, and even accesses of cached web pages.

More closely related to our work, power consumption data has also been used to study the execution of programs. Isci et. al. [24] present a method for identifying phases in program power behavior and determining points in the execution of the program that correspond to such phases. The authors argue that programs exhibit changes in their behavior throughout their execution, which constitute phases of the program's execution. To build a representation of such phases, the authors use performance counters and a sensor for measuring total processor power consumption. As a program executes, their framework is able to estimate power attributes of each processor component using a combination of the performance counter information and the measured power consumed. Using this approach, the authors are able to generate what they call a *power vector* that represents the estimated power values for 22 processor components such as trace cache and integer execution unit. This power vector is populated at each sample point. To determine if two sample execution points are the same, the authors compute the similarity matrix using the raw and normalized

power vectors of the two sampled points. To compute the distance between the two power vectors, the Manhattan distance algorithm is used. This approach allows them to identify similar phases of the program executions. The authors also demonstrate how their approach can be used to define a program "signature" by first grouping similar execution points using a thresholding algorithm based on the aforementioned technique. Once similar execution points are grouped, a few "representative" power vectors are selected from each group to generate the program "signature." The authors use the gzip benchmark to demonstrate their approach.

2.2 Traffic Analysis

Side channel attacks in IP network traffic have been extensively studied in the past. Traffic analysis attacks were highlighted in "Attacks of the SSL 3.0 protocol" [8], by Wagner et al., who show how the URL of an HTTP GET request is leaked in SSL due to the inability of the cipher-texts to disguise the plaintext length. Over time, researchers have applied statistical modeling and machine learning techniques to a variety of data features in order to advance the efforts of traffic analysis. Some of these efforts are described below.

Cheng and Avnur [7] show that websites can be fingerprinted by performing traffic analysis of SSL encrypted web browsing traffic. In their work, the authors assume the attacker is on the same network as the target user. Additionally, the attacker is capable of sniffing the network using a tool such as *tcpdump*, a popular command-line packet analyzer. For identification of the website visited, their approach leverages only information about the size of the packets exchanged between two endpoints. Also, the authors consider the fact that Web users often follow several links on a website before leaving, and use this observation to devise a link analysis algorithm with a window size of three samples. In other words, for a given sample observed at time t, their algorithm also analyzes the samples at time $t-1$ and $t+1$. The authors also consider using hidden Markov models but determine the technique is inadequate since user behavior can vary (e.g., a user may backtrack, only follow a link and leave the page, etc.), which is not permitted by HMMs. To evaluate their approach, the authors run a user study as well as simulations using different web access patterns and even some countermeasure techniques, such as packet padding, fixed packet size and caching. The

accuracy of their approach shows that even in the presence of some countermeasures, it is possible to identify the website accessed using traffic analysis. Other research efforts expand beyond the use of length metrics for traffic analysis. These [25, 26, 27, 28, 29, 30] leverage various features including source and destination attributes (e.g., address, port), protocol, and even timing information (e.g., duration of connections, burst rate of transmissions).

Efforts have also focused on discovering countermeasures for such attacks. Luo et. al. [31] propose a configurable client side system, named HTTPOS, capable of applying a variety of traffic transformation techniques. HTTPOS implements two strategies to defend against traffic analysis attacks. A diffusion strategy is designed to introduce features as to cause the classification algorithm to perform poorly. A confusion strategy aims at manipulating features related to one web page to resemble a different web page. To this end, HTTPOS is capable of manipulating several features including packet size, web object and flow size, and timing of packets. These manipulations are possible by leveraging HTTP and TCP options. To evaluate HTTPOS, they use 100 of the most popular websites and test it against four different attack methods. Specifically, the four attacks use number and size of web objects, inter-arrival times and packet size, flow direction and packet size, and sequences of packets respectively. The results of their evaluation show that the accuracy of the four attacks drop significantly, in some cases to 0% for all 100 websites. Other efforts [32, 33] explore other countermeasure techniques, all varying by the method used (e.g., traffic padding, traffic masking) and location of enforcement (i.e., server side, client side, or both).

More recently, in "Peek-a-Boo, I Still See You: Why Efficient Traffic Analysis Countermeasures Fail" [5], Dyer et. al. provide the first comprehensive analysis of previously proposed traffic analysis countermeasures and show why they fail to protect against attacks. The authors show that nine countermeasures such as fixed packet size, fixed packet periodicity, and even traffic morphing techniques are vulnerable to attacks. To prove this, they use a naïve Bayesian classifier and a support vector machine classifier based on total connection time, bandwidth, and the size of bursts.

For example, using a test data set, the authors show that the accuracy of their classifier drops to 5% only after adding countermeasures that impose a 400% overhead in the number of bytes sent during the connection.

While most efforts present practical methods, researchers have also taken a theoretic approach to traffic analysis attacks. An example is the work of Backes et. al. [34] who take a formal approach to traffic analysis attacks and develop a mathematical model which represents an application's web traffic and can be used to derive security guarantees for the application. The authors consider network traffic as an information channel and model the network traffic for any given application using a directed labeled graph. The nodes of the graph represent the states of the application while the edges correspond to user actions. The patterns of the application web traffic are also reflected in the graph representation through vertex labels. Given such a graph, the authors are not only able to represent a web application but also a series of "fingerprints" for the application through the mapping of vertices to observable events (denoted by the vertex labels). Furthermore, users' traversal of the website can be represented by a path in the graph and can be defined by the set of observations denoted by the graph. The authors take this model to show how network fingerprints can be composed and how such fingerprints affect the security of a web application.

While the publications described above use traffic analysis only to identify websites accessed by a user, applications of traffic analysis techniques expand beyond that, both with respect to objectives and features used. The following are examples of such. Dusi et. al. [35] detect application-layer tunnels by using statistical features of connections such as packet size, inter-arrival time between consecutive packets, and number of packets. Kohno et. al. [36] leverage differences in TCP and ICMP clock skewness to fingerprint remote physical devices. Some researchers use channel state information values to recognize keystrokes [9] and even identify human presence indoors [37]. In wireless networks, channel state information provides information about the channel regarding phenomena such as scattering, fading, and power decay. This information plays an essential role in building reliable connections.

In our work, we consider encrypted network traffic as a side channel and we show how it can be used to model and monitor the behavior of Internet of Things (IoT) devices. In contrast to network traffic generated by a user on a desktop machine, IoT devices are delivered with pre-programmed firmware that defines a relatively small and finite set of network calls that the device is capable of making. This property makes this problem different than other types of side channel attacks against IP network traffic that we have

come across.

In our literature search, we were able to identify only one paper [6] analyzing wireless home automation communications. The authors conduct analysis on two installations of the HomeMatic home automation system and try to infer the user behavior based on the network traffic from these systems. Specifically, without prior knowledge of the HomeMatic installations, they use the content of communications between devices to not only identify the devices within the home, but also user behavior. The authors define any communication event as a tuple of four elements: source address, destination address, message type, and message content. For any time window, their approach looks at all events occurring and perform correlation analysis to determine events which occur together. Since they leverage content of decrypted communications, their work relies less on side channels and more on deep packet inspection.

2.3 High Performance Computing Activity

Previous work has also focused on studying various aspects of activity on high performance computing (HPC) platforms. Considering the expenses required to operate and maintain such a massive computing platform, researchers are interested in identifying bottlenecks both in the platform and in the programs in hopes for a more efficient use of the resources. To this extent, a large number of efforts have focused on studying and identifying I/O behavior of HPC applications.

For example, Liu et. al. [38] present an approach for reliably estimating the user-applications' bandwidth needs. The authors claim that using such an approach, it is possible to identify I/O intensive jobs and use this information to optimize the scheduling of jobs on a HPC platform. The authors leverage the fact that applications exhibit periodic bursts of I/O activity. To capture this phenomena, their approach applies a wavelet transform to I/O statistics reported by RAID controllers in order to generate per application I/O signatures. Their approach also performs noise reduction, given multiple samples for the same program. The authors evaluate their approach on both a set of pseudo-applications and a large-scale scientific application and compare their approach to one based on dynamic time warping, presented in a previous work. Dynamic time warping refers to an algorithm which aims to

align two time series by warping the time until a match is found. The evaluation shows that their approach not only outperforms the alternative but also imposes a lower overhead.

Similarly, Luu et. al. [39] analyze the I/O behavior of applications across multiple runs on an HPC platform and analyze the evolution of applications across time and across platforms. The authors rely on data from a scalable HPC I/O characterization tool named Darshan. Darshan aims to capture an accurate picture of application I/O by profiling POSIX, MPIIO and HDF5 I/O operations. In this work, the authors are not interested in generating signatures or models for applications based on Darshan data, but rather using Darshan data to identify I/O related bottlenecks in scientific applications. By mining Darshan logs, the authors are able to make a number of observations regarding the I/O behavior operations. For example, the authors discover that very low I/O performance is the norm for most apps on such HPC platforms. Additionally, they are able to determine that the I/O resource usage is most often dominated by a small number of application and that POSIX I/O is more widely used than parallel I/O libraries.

Other efforts have attempted to identify running applications using various data sources. Peiser [40] and Whalen et. al. [41, 42] use the concept of "computational dwarfs" to build a technique for classifying programs based on their computation type. Researchers previously showed that the patterns of communication and computations across all scientific applications can be captured by 13 models or computational dwarfs [43]. Considering this observation, the authors study the relationship between the 13 computational dwarf classes and patterns in Message Parsing Interface (MPI) function calls in HPC applications. MPI is a communication protocol which enables the communication between processors in a massively parallel computer. To classify applications into computation dwarf classes, the authors consider the communication patterns as a graph, where nodes are processors and MPI function calls are represented as edges between the nodes. Once the graph is composed, the authors can extract features describing the graphs such as node degree (i.e., number of edges connected) and node centrality (i.e., importance of a node in a graph). These features are able to capture information about which and how nodes (and associated ranks) communicate. With the features extracted, the authors apply machine learning techniques (e.g., clustering, naïve Bayes, hidden Markov models) and show that applications can be classified

into computational dwarf classes with accuracy over 90%.

Similarly, DeMasi et. al. [44] attempt to classify high performance codes by leveraging information available in performance logs. Specifically, the authors use both timing information and communication measurements. Timing data determines where each program spends its time (e.g., user, system, MPI calls). Communication measurements allow the researchers to determine total time spent for various communication commands, the number of calls made from the commands, the percent of MPI time accounted by the commands, and the wall time spent for the execution of the commands. As with the work of Whalen et. al., the data used for building models is *clean* data in that each performance log only contains information regarding one program and as such, it is free of noise. To capture the model of an application, the authors apply the rule ensemble method, which refers to the combination of many simple base learners into a larger model. Evaluation of their approach shows that HPC codes can be identified with 93% accuracy.

2.4 Information Channel Bandwidth

Researchers have previously studied information channels and the amount of information communicated by a given channel, also known as the channel's bandwidth. Most of the previous work in this area studies covert channel bandwidth. Millen et. al. [45] look at covert channels relative to information flow and formalize the idea of measuring covert channel capacity using Shannon's theory of entropy. Costich et. al. [46] analyze the covert channel of a multilevel secure database that uses a two-phase commit protocol to guarantee atomicity of transactions. The covert channel is introduced if two components communicate with each other via selectively aborting transactions. The authors study the capacity of this covert channel in order to understand how it can be reduced to improve security. Other efforts [47, 48] perform studies of covert channel bandwidth in various settings including multilevel secure operating systems and interrupted-related channels.

Kang et. al. [49] focus on combating covert channels. In this work, the authors introduce a pump used for controlling communication between two processes. The pump uses buffers that handle and processes messages between two processes at a rate which reduces the bandwidth of the covert channel compared to direct communication between those processes.

Other efforts [50] focus on automatically detecting side channels.

However, some previous works exploring side channels look into the amount of information exposed by such information channels. In [51], Micali et. al. build a comprehensive but general model for defining and delivering cryptographic security in the presence of side channel attacks. In their work, they formally define side channels in terms of a Turing machine and show how some basic properties of traditional cryptography do not hold in a physically observable setting. Although their work does not explicitly study the bandwidth of side channels, their model attempts to construct pseudorandom generators and encryption schemes that have no distinguishable effect on side channels. Implicitly, this property of the model aims at limiting the amount of information (i.e., *bandwidth*) exposed by side channels during the execution of cryptographic algorithms. Standaert et al.[52] build upon this work and evaluate the effect of physical leakages with a combination of security and information theoretic measurements . Goguen and Meseguer [53] consider side channels as channels enabling information flow. To this extent, they introduce a general automaton theoretic approach to modeling secure systems, specifically focusing on the concept of *noninterference*. The authors define noninterference as the scenario where the actions of one group of users have no effect on what another group of users can see. Using automaton theory the authors define a system as a state machine consisting of users, states, commands, outputs, and capability tables. This framework is then used to show how security policies can be generated to define which information flows are not permitted and how such a security policy can be reduced to a set of noninterference assertions. Sutherland [54] defines the concept of *nondeducibility* and presents a simple model of information and inference, giving an instantiation of the model to state machines and applies the instantiation to a simple example. In his work, Sutherland aims to formally define information and the meaning of inferring information from other information. These formal definitions of side channels consider side channels as implicit or explicit information flows and build frameworks to guarantee security policies against side channels attacks for cryptographic devices. To the best of our knowledge, no previously published work looks at how accurately the information exposed by side channels reflects system activity.

Chapter 3

Early Work

Early in our work, we began exploring side channels in various settings. The intent of the projects described below was to explore novel beneficial uses for side channel information. We begin by describing a project in which we use side channel information to infer user interests within the scope of a smart phone application and leverage these inferences to optimize network usage of the application. Next we present work that leverages side channel information to reverse engineer the protocol for user interaction with an unknown binary. By performing these exploratory projects, we gained valuable insight into the depth of information available in side channels. This insight helped us develop our formal model for side channel information.

3.1 Modeling User Behavior for Improved Network Efficiency in Android Applications

When building a smart phone application, developers must consider a variety of implicit and explicit desires that users may have about what the application does and does not do. Aside from the application's key features and usability of those features, smart phone users value battery life and network data consumption highly. Considering that the LCD display is one of the largest consumers of battery power, there is little developers can do to help with power consumption. On the other hand, developers can apply various methods for diminishing the network data usage footprint of an application. Specifically, developers may implement efficient network transmission strategies. For example, instead of sending

raw data, it is more efficient to use compression algorithms. If possible, it may also be advantageous to buffer packets and send them in batches rather than send each individually on arrival. This implies a compromise between the responsiveness of the application and data consumption.

While existing approaches can decrease the network usage of an application, such approaches fail to consider the users and their behavior. In this part of our work, we show another approach for diminishing network data consumption by leveraging patterns in the users' interaction with the application and using that knowledge to create a more intelligent network management framework. In this scenario, the side channel is the users' interaction with the application and it is leveraged to improve the network efficiency of Android applications. Specifically, we develop a framework and a proof of concept for Android that tracks user activity (e.g., scrolling through UI window) in an application in order to determine user's interests (i.e., user profile). The user profile is then used to limit the amount of information the application downloads from the server during the application's runtime.

The framework uses callback functions of the Android WebView component to monitor the scrolling movements of the user. A WebView component is a browser-like widget that allows HTML content to be displayed. WebViews are often used by developers to deliver a web application as part of a client application. To enable additional features and customization, the Android platform defines callback functions for several WebView events. For example, there are callbacks for overriding URL loading, introducing logic before and after the loading of a page, and even scrolling events, including movements and clicks. Our framework leverages scrolling callbacks in order to monitor users' movements.

For the framework to be valuable, it must not introduce significant degradations in overhead and performance of the application. This requirement impacts two aspects of our framework: user profile management and data download process.

The user profile contains information about the preferences of the user. These preferences are inferred from the scrolling actions of the user. Our framework uses a Bloom filter to store the user profile. A Bloom filter is a space efficient probabilistic data structure. Bloom filters are usually represented as an m bit array. When an element is added to the structure, k different hash functions are used to map the element to locations in the bit array. Each hash

function maps the element to one (and only one) of the m array positions. To test if an element is a member of a Bloom filter, the hashing functions are used on the target element and the appropriate index locations of the bit array are checked. This is what makes Bloom filters probabilistic. If all bits are marked, we can determine with very low false negative rate that the target element is contained by the Bloom filter. It is important to note that depending on the hash functions, it may be impossible to guarantee that a given element is a member of the filter (due to collisions).

Bloom filters are an ideal data structure in this application for several reasons. We wish to contain information about the user's preferences in a compact way. Additionally, the framework should aim to impose minimal runtime overhead, when checking data against the user's profile. Last, the framework should not greatly impact the user experience of the application. To that extent, the low false negative property of Bloom filters is also appropriate since a false negative would cause bad user experiences. This is because a false negative would prevent the framework from fetching all of the information of possible interest to the user. As a result, the framework will then have to fetch the data as the user scrolls to that particular area, which imposes a slowdown.

For our framework to be practical, it must also not impact the user experience. Users are dynamic and their preferences may evolve over time. To account for this property, we use an overlapping time window approach. Specifically, at any given time, the Bloom filter only contains information gathered over a finite., customizable time window. To be relevant and efficient, time is only measured when the application is the main window in focus on the smart phone. Aside from accounting for dynamic behavior of users, this approach also guarantees that the size of the Bloom filter is limited.

We also consider concerns regrading user privacy. Leveraging side channels such as user behavior to infer user preferences could be construed as a violation of user privacy. To protect the user, our framework is integrated within the smart phone application and the Bloom filter representing the user profile need never be transmitted. This statement does not hold against malicious application developers. Although methods exist, defense mechanisms against such attacks are beyond the scope of this work.

To test our framework, we developed a proof of concept. The proof of concept is an

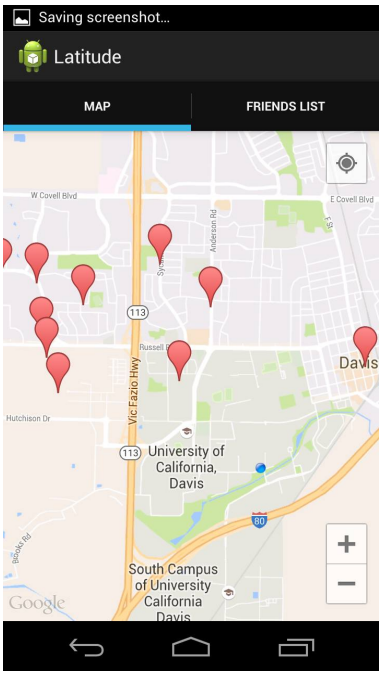


Figure 3.1: Screenshot of user interface of the demo Android application, displaying "friends" locations across a geographical map.

Android application that allows the user to see the location of several "friends" on a map. The application is shown in Figure 3.1. The application has a WebView component that displays a geographic map and "friend" locations that are displayed as pins. The "friends" locations are programmatically controlled and randomly generated by the server.

The locations of the "friends" change at random time intervals. The application is designed to periodically query the server for fresh data. This updating process can be done either while the app is in the background or when the app becomes active. Without the framework, at startup, the application downloads the location of all "friends" and caches it. In the case where the framework is enabled, the application only requests and downloads the location of "friends" in sections of the map of interest to the user as defined by the user profile generated by the framework. For example, supposed that the user is located in California and they have "friends" in Berkeley, San Francisco, Lake Tahoe, and Los Angeles. If the user scrolls to the Berkeley and San Francisco areas of California, the framework will only request data for those geographic locations and not for the Lake Tahoe and Los Angeles areas.

To achieve this, the server must allow for querying of certain data points, in this case,

based on geographic location. During testing of the framework, our framework was able to reduce the amount of network data up to 30%. This figure comes from comparing the total amounts of data consumed by the two versions of the demo application, with and without the framework. It should be noted that the savings in network data usage is dependent on the user behavior. If the user's behavior is such that it includes all data points (i.e., "friends"), the use of the framework does not introduce any advantage.

Our framework has certain limitations. One limitation is with respect to the type of applications. Currently our framework only works well when data is overlaid on top of a WebView (as the case with pins displayed on a map). Another limitation is that there must be cooperation between the framework and the application server. This is not a major constraint since generally both the smart phone application and the application server are created by and under the control of the same developers. Some of these limitations could be eliminated with some extensions of the framework. The framework could be improved to even handle dynamic websites. To achieve this, the framework would be embedded into the WebView library and would have the framework logic integrated with the DOM parser. As in the demo application, the framework would monitor a user's behavior and build a user profile. For example, the framework could attempt to identify if the user is interested in images and perhaps even what type of images. It should be noted that the current WebView has an option which allows for images to not be downloaded. However, such a limited binary option may not accurately reflect users' preferences and may result in a bad user experience. The DOM parser is responsible for parsing the Document Object Model tree of a webpage and fetching the required content. With the help of the framework, the DOM parser could only fetch objects that are of interest to the user by consulting the user profile generated by our framework. We believe the approach presented here can help reduce the network footprint of smart phone applications more than current approaches are capable of.

While the framework may not be suitable for all types of applications, we believe it is important to consider users' preferences and behavior. These features are often reflected in various side channels and can be leveraged for beneficial purposes.

3.2 Modeling Input Protocol for Unknown Binaries using Hardware Performance Counters

Side channels present other beneficial opportunities as well. As previous research efforts show, side channel information can be used to infer input to a system. In this part of the dissertation, we show how side channels can also be leveraged to infer input for unknown binaries [1]

3.2.1 Finding Input

Testing is an important step in the software development process. Aside from quality assurance purposes, testing can also discover security vulnerabilities in a program. However, thorough testing of a program is nontrivial. Most developers use documentation to interact with the program and exercise its functionality. Yet documentation can also be incomplete.

Static and dynamic analysis techniques can provide some assistance with this problem. Static analysis can be applied to extract user-commands the program understands. This can be a complicated process and often requires source code. While there are a number of dynamic program analysis techniques available, their effectiveness depends heavily on the completeness of the test suite applied during the analysis process and assumptions about the source of information being used for the side channel attack. Test suites are either manually created by developers or automatically generated by fuzzers or symbolic executioners. As is the case with documentation, when manually created, it is possible that the test suites are not complete. At the same time, while fuzzers can be effective, they can take a long time to run exhaustively and unless combined with symbolic execution or branch monitoring, provide poor code coverage. Symbolic execution, despite being successful, also has limitations such as (in some cases) manual modification of source code, scalability issues (linear-time for linear constraints over rationals). To make matters worse, constraint solvers also have innate limitations especially when it comes to solving Boolean formulas (decidable but NP-hard), linear constraints over rationals and integers (decidable, NP-hard), and non-linear constraints over integers (undecidable).

[1] This work was published in the proceedings of the 5th Program Protection and Reverse Engineering Workshop (PPREW-5) of the Annual Computer Security Applications Conference (ACSAC).

In this part of the work, we introduce a new method and its implementation called InputFinder, for generating valid user-input for closed binaries. The method relies on monitoring the amount of work the processor performs as a response to user input. We consider this to be a side channel since we are not monitoring the program itself but rather the hardware and its response to the program execution. This new method could be useful in autonomous vulnerability scanning systems, for reverse engineering unknown binaries, or as a complement to fuzzers or symbolic execution engines.

The majority of software programs accept input, perform transformations on the input, and output results. However, most programs do not accept completely random input. Input usually passes through an input validation filter. These filters are snippets of code responsible for distinguishing between valid or good input (i.e., input the program was constructed to understand and accept) and bad input (i.e., input which is not useful or does not follow the desired format). Although validation mechanisms can be very sophisticated, they are often a combination of string comparisons and conditional statements. Our method exploits changes in the number of user-land instructions retired during multiple executions with varying inputs to make inferences about the program's validation mechanism. These changes reflect different execution paths of the program as a response to both valid and invalid input.

It is important to observe that even if the computer performs more work, it does not necessarily imply the input is correct. To understand this, consider the trivial example where the input validation mechanism is composed of a series of nested conditional statements. Each depth level of the conditional statements verifies one index location of the input string, from 0 to n where n is the length of the string. The computer will perform less work if the input string passes the first conditional statement than if it does not. In other words, if the first character of the input string does not satisfy the first conditional statement, the execution iterates through all of the remaining conditional statements at the first level of depth. This example shows how *more* computations do not imply validity of the input.

Our method leverages the following assumption: Out of the total set of printable characters, only a small subset will be valid at any index location of the input string. Specifically, for our approach to work, the subset of valid characters at any index of the input string, has to be $\frac{1}{2} \times |C|$ where C is the set of printable characters. Based on this assumption, the

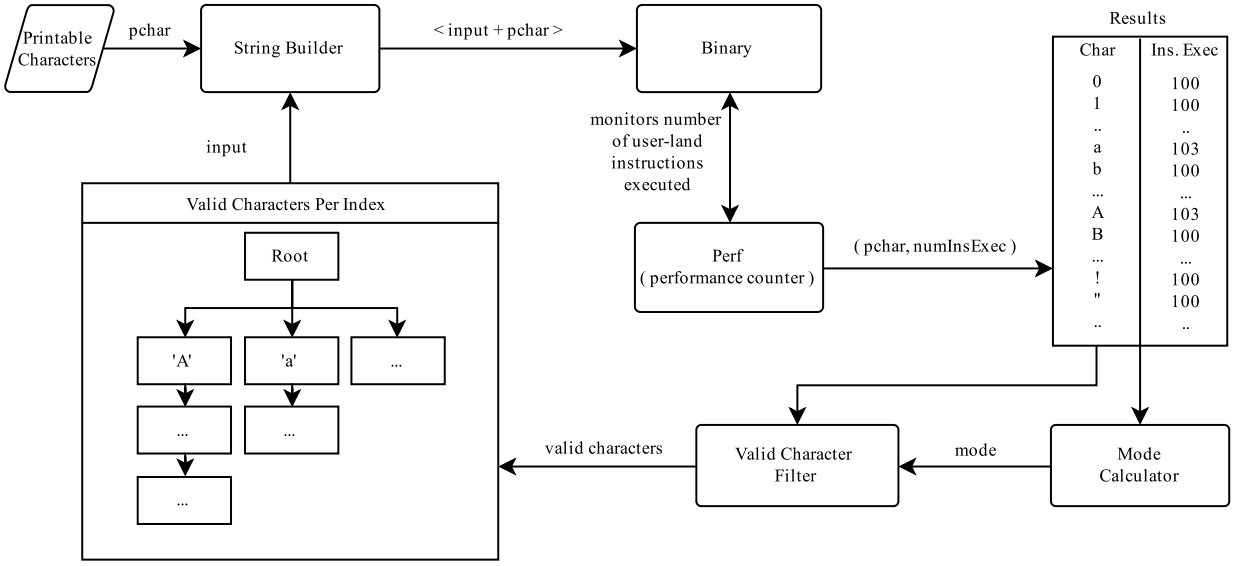


Figure 3.2: Diagram depicting the architecture of the finding input process.

control flow paths of any invalid character should be the same, while the control flow path of valid characters will differ. If *most* printable characters are invalid, our method can trivially identify the set of valid characters (i.e., characters that exhibit a different control flow path).

3.2.2 Method Architecture

The architecture of InputFinder is depicted in Figure 3.2. Our approach is uses two steps:

1. valid input strings generation

2. protocol state machine generation

The first component exploits hardware performance counters to build valid input for closed binaries. Specifically, as the program is given various inputs, InputFinder records the number of instructions *retired* during the program's execution for each input and uses a statistical measure to infer the program's expected input. The instructions retired represent the subset of instructions executed which leave the *Retirement Unit* once execution has been deemed correct (i.e., the instructions which actually impact the program). The Retirement Unit is responsible for assuring the correctness of the execution due to out-of-order processor pipeline. Specifically, once the execution order has been deemed correct, the unit writes the results of speculatively executed instructions into registers and removes them from the re-order buffer. These instructions are then deemed *retired*. As such, the number of instructions

retired differs from the number of instructions executed by a processing unit. Because of the out-of-order CPU pipeline, a CPU may execute more instructions than necessary (e.g., as a result of wrong branch prediction).

Since our input finding method works by leveraging information about how much work the processor performs during input validation, it does not depend on a particular platform, format (i.e., binary, source, etc), or method of obtaining input (e.g., *stdin*, arguments, network socket, file I/O). However, there are some constraints. The underlying method does not work if the user input is transformed (e.g., hashed) before validation. The implementation of our method also has some additional drawbacks. Our current implementation can only handle programs whose input is interpreted as a string (or a set of characters). The user input can have many components, however, there may not exist any internal dependencies between the components (e.g., value of one component defines the length of another component). If the input is composed of multiple components, the program must process the components sequentially in order for our method to work. The same hardware counters are used by InputFinder to also find the expected input size and to categorize the expected input based on type.

Figure 3.3 visualizes the process of building input strings. Multiple outgoing arrows represent a process forking. The steps of the process are outlined below. Starting with an empty string, s:

1. InputFinder executes the program once for every printable character and records the number of user-land instructions retired by the program with the given input. The input is a string concatenation of the string s and the current printable character. The Unix *perf* utility is used for monitoring the number of instructions retired. Note that InputFinder does not record the total number of instructions but only user-land instructions. The reason is that, in contrast to user-land instructions, the total number of instructions retired can vary greatly from execution to execution for various reasons (e.g., branch prediction, cache). The Expect extension to the Tcl scripting language is used for automating the program interaction.

2. Once all executions have been completed (one for each character), the mode number

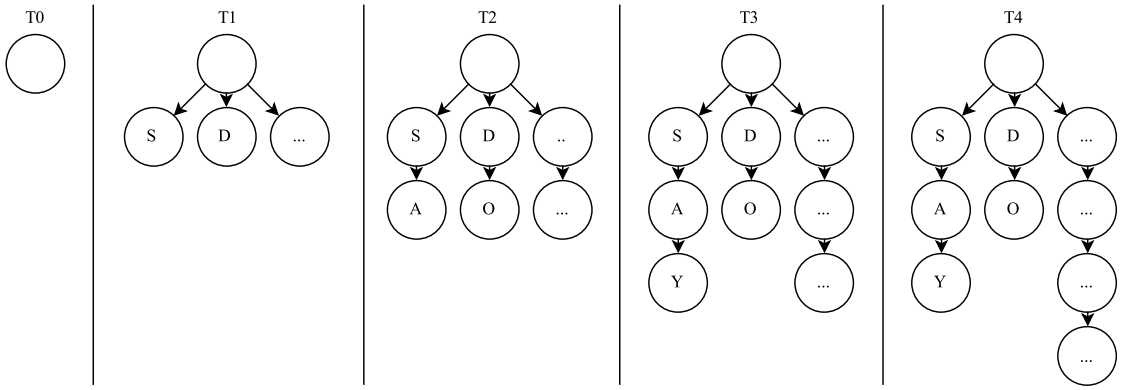


Figure 3.3: Diagram visualizing the input strings being constructed throughout five consecutive time stages of the execution of InputFinder.

of instructions retired is computed across all executions. The mode is defined as the value that appears most frequently in the set of recordings.

3. Next is the filtering stage which is responsible for identifying characters likely to be part of a valid input string at the current index. Under the assumption that most characters are not valid for the current index, InputFinder gathers all characters with the number of instructions retired outside the range of the mode +/- an epsilon value. These characters represent valid characters for the current index of the input string. The epsilon value accounts for small variations in the hardware counters. While the initial reaction is to expect a valid character to result in more instructions retired than a non-valid character, this is not always the case. As discussed earlier, the input validation mechanism can be complex and can vary in behavior. The validation mechanism may verify input against all accepted input strings before denying the provided input. Such a scenario results in more instructions retired for invalid input than valid input.

4. For each valid character c, InputFinder forks, with every child process repeating steps 1-4, starting with $c + s$ as the initial input string, where the symbol "+" signifies concatenation. If no valid characters are found, the end of a valid input string has been reached and the string is recorded. If the current index is 0 and no valid characters have been found, the given binary has no predefined input commands.

This process continues until all child processes fail to identify valid characters and gracefully terminate.

For improved results and increased code coverage, it is important to determine whether the program only responds when a user-program interaction protocol is abided by. Consider, for example, a database management program. Without first selecting a database and perhaps even a table, some commands such as *INSERT* or *SELECT* (or their equivalent) have no effect. We develop the second component of InputFinder to address this issue. Specifically, this component uses the valid input strings generated previously to identify the expected orderings (or permutations) of the discovered input strings, and constructs a protocol state machine. This enables the generation of a more thorough test suite.

The process can be split into three steps.

1. All of the possible combinations of user inputs (given the discovered input strings) are generated.

2. The inputs are passed to an instrumented version of the binary, which outputs a single execution trace for each execution (one execution per permutation of input strings).

3. To determine the correct input protocol, the execution traces are compared.

To generate all of the possible combinations of user inputs, all permutations of the discovered input strings are enumerated. The permutations range in size (i.e., number of strings) from 2 to n, where $n = |input\_strings\_set|$. However, enumerating all possible permutations of the discovered input strings may not be enough. Consider cases where the user input is composed of two (or more) strings where the first string is a command and the rest are arguments. Now also imagine that the arguments of an input string depend on the previously entered string. To cover such cases, for each permutation, we test each input string for arguments (using the previously mentioned input finding technique). For example, for a given permutation $\{AUTH, SET\}$, our tool will first try to identify arguments for the $AUTH$ input string. Without finding the correct argument for the $AUTH$ command, the execution of *SET* will not affect the program's state. Let us assume our approach was successful in identifying a single argument, *0*, for the $AUTH$ string. Now, the tool will attempt to find arguments for *SET* but with *AUTH 0* as the first entered user input. Again, this ensures that our approach identifies arguments for strings that depend on previously entered strings.

The resulting permutations of strings (and their arguments) are also passed through the instrumented binary in order to collect execution traces. The binary is instrumented with Intel's *Pin* tool [55] which logs the starting address of every basic block executed, as a function of time. A basic block is a set of instructions with a single entry and a single exit point. Once all permutations of a given size have been executed, the dynamic similarity test analyzes the resulting execution traces. Each execution trace is split into sections, each sections associated with the program processing one of the input strings of the input permutation. These sections are delimited by a short sequence of specific basic blocks which are executed right before the reading and processing of the user input. To determine the *correct* or *expected* input string permutation, the execution trace section of the last command in a permutation is compared against the execution trace sections of that same command, in the same index of the permutation. Under the assumption that the execution of the last command will exhibit a different behavior given the correct prior user inputs, our approach identifies which input string permutation causes the execution trace section of the last command to differ from other execution sections of the same command, in the same index of the permutation. If such a permutation is discovered, it is added to the protocol state machine.

3.2.3 Experiments and Results

In order to determine its effectiveness, our approach was tested on 24 x86 binaries from the DARPA Cyber Grand Challenge (CGC) Example set [56]. The tool described here was designed in preparation for the Cyber Grand Challenge, hence this test set was a natural choice. While designed specifically for this challenge, the programs implement a variety of real-life services such as a palindrome detector, interactive game, image compressor, and a multipass protocol implementation (i.e., RFID payment card binary protocol). Cyber Grand Challenge binaries are 32-bit x86 binaries in the CGC format. There are a few differences between CGC binaries and ELF binaries. The CGC format has a slightly different header and does not allow dynamic linking. To isolate the challenge environment from the rest of the world, the organizers of the CGC created a custom-Linux derived operating system that has no signals, no threads, and only seven system calls. At the time of this writing, there were only 24 CGC binaries available, all of which were used for the analysis of our approach.

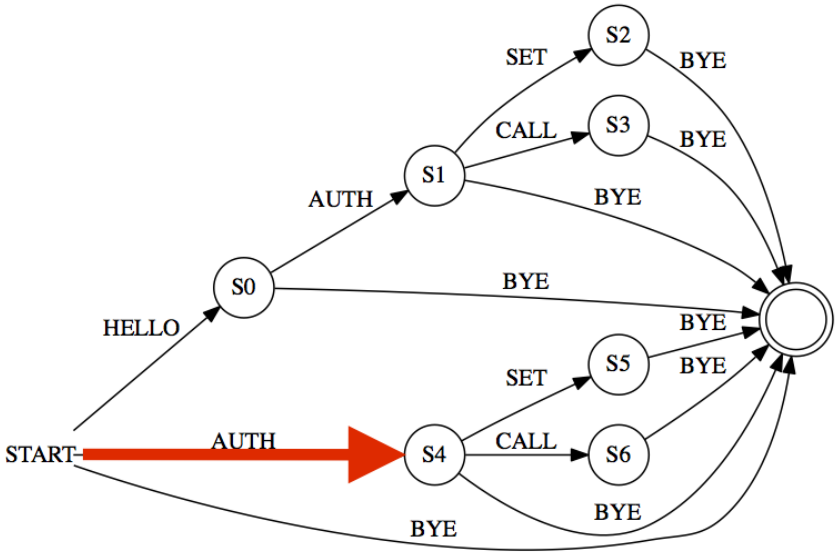


Figure 3.4: Diagram visualizing the user protocol state machine of one of the CGC programs, with the backdoor highlight in red.

Analyzing the source code showed that out of the 24 binaries, 21 had predefined valid input strings. Our approach found input strings for 13 of the 21 binaries and categorized the input correctly for 100% of the 13 binaries. The running time varied between a few minutes and an hour, depending on the number and length of the valid input strings discovered. Our method was able to determine the input length for 19 of the 24 binaries. Note that our approach was able to find the expected input length even if it was not able to find valid input. Additionally, our method was able to determine inputs that caused crashes for 5 of the binaries. In all cases, the input caused a buffer overflow and caused the binary to exit unexpectedly.

Using our approach, we were also able to identify an authentication backdoor for one of the binaries. The binary implements a protocol that is initiated by the user sending the "HELLO" command. The user protocol stat machine is visualized in Figure 3.4 and the authentication backdoor is highlighted in red (edge from node $START$ to node $S4$, represented by the $AUTH$ command). Once the user enters this command, the program generates a passcode and echoes it to the user. This passcode may be used by the user to authenticate.

The authentication must happen and enables the user to use the full functionality of the program. InputFinder was able to detect an authentication backdoor, where if the user does not type "HELLO", authentication can be done using the character zero (i.e., 0).

3.2.4 Conclusions on Finding Input Using Side Channels

While the method presented here has many advantages, it also has a few limitations. The input finding implementation presented is only capable of finding input for programs that interpret the user input as a string (or series of characters). The method presented cannot handle cases in which input is composed of multiple inter-dependent fields. As an example, the method presented fails in cases where the input is composed of multiple fields and the value of a field determines the length of another field or in cases where binaries validate input fields out of order. Such failures were observed in the evaluation of InputFinder. InputFinder does not handle binaries which accept non-printable input (e.g., hex characters, binary, etc.), although we believe our method could be extended to also cover non-printable input. Perhaps the most significant drawback of our method is the runtime complexity. For any given binary, to find valid input strings (without valid arguments), the runtime is $O(nm)$, where n is the length of the *longest* valid input string in the valid input string set and m is the total number of valid input strings accepted by the program. The process can be parallelized but it is ultimately limited by the number of hardware performance counter registers.

Despite these limitations, this study further supports the claim that side channels expose valuable information. It is also important to note that in our experiment, the side channel information is agnostic of the operating system, architecture and even binary format. The work presented above also shows that there is a relationship between the activity of the system during the execution of a program and the side channel information and lays the foundation for the following chapters.

Chapter 4

Monitoring High Performance Computing Platforms

In the previous chapter, we demonstrate how side channels, particularly hardware performance counters, can be exploited to learn about a program's response to (user) input, and consequently identify valid inputs that the program accepts. Ultimately, our work shows that in some cases, it is possible to infer changes in the control flow of a program based on side channel information.

In this chapter, we develop that idea further. However, instead of using side channels to infer properties of a program, we use side channel information as a "fingerprint" for programs and use these fingerprints to detect what programs are running on a given HPC system. Specifically, we demonstrate how side channel information can be used to characterize and identify the load of a high-performance computing platform.

4.1 Security in High Performance Computing Platforms

With the rise in the availability of data, scientists across the world are in desperate need of computing facilities capable of handling such massive amounts of information. Due to high costs of operating and maintaining such high performance computing (HPC) platforms, only a few fortunate institutions are able to build and provide such facilities. Additionally, the financial overhead imposes pressure on these institutions to achieve maximum efficiency. From a cyber security standpoint, this presents several challenges.

As with most systems, the users are among the biggest threats to security and efficiency.

Most high performance computing platforms are intended to be utilized by researchers for scientific purposes. Guaranteeing ethical user behavior is non-trivial. In some cases, such as at the Lawrence Berkeley National Laboratory, both procedural and technical measures are taken to assure these valuable resources are used as intended. When applying for access to the HPC platform, researchers are asked to provide a list of scientific codes that will be used. These codes may be reviewed by system administrators. Additionally, user agreements are used to legally force users to abide by the rules of ethical use of the super computing facilities.

Yet enforcing the user agreement is challenging, especially in real-time. Once access is granted, users could misuse the facilities for various reasons such as running electronic currency miner software or botnets. Incidents of such breaches of user agreement and ethics have been documented [57].

From a technical standpoint, determining what program is running on a given system is challenging. Monitoring tools can expose a variety of information regarding the behavior of a system. On the other hand, monitoring tools provide minimal insight into the computation type of a particular process. Static and dynamic analysis techniques can provide some assistance, especially in the absence of obfuscation. However, aside from being intrusive, such analysis is difficult and time consuming.

Our work is motivated by the problem of providing HPC system administrators with a non-intrusive monitoring alternative capable of identifying what programs are running. Specifically, in this chapter, we show how side channel information can be used as a non-intrusive alternative method to monitor activity on high performance computing platforms.

Programs, at runtime, can be described by the sequence of instructions executed, the contents of registers, and the input provided to the program. If we wish to fingerprint programs using side channel information, it is important to consider side channels that reflect CPU and memory activity.

Aside from identifying individual programs, in an HPC environment it may be sufficient to determine if a particular program is performing a particular type of computation. To this extent, our work is inspired by the observation made by Phil Colella regarding patterns in computation of various scientific applications. In 2004, Colella observed that most scientific

and engineering codes exhibit one of seven distinct patterns in terms of computation and communication activity [58]. He named these patterns *computational dwarfs* or motifs. Formally defined, a *computational dwarf* is a pattern of communication and computation common to applications which share that computation type. In his work, Colella highlights that despite variations in implementations, codes of a particular dwarf share the underlying pattern in communication and computation. Inspired by Colella, researchers from Berkeley have expanded the list of computational dwarfs to thirteen [43], as listed below (original seven in bold):

- **Dense Linear Algebra**
- **Sparse Linear Algebra**
- **Structured Grids**
- **Unstructured Grids**
- **Spectral Methods**
- **N-Body Methods**
- **MapReduce**
- Combinational Logic
- Graph Traversal
- Dynamic Programming
- Backtrack and Branch-and-Bound
- Graphical Models
- Finite State Machines

In our work, we study how patterns in computations and I/O are represented in side channel information and demonstrate that side channel analysis can be used to identify running programs.

4.2 Power Analysis

Historically, electrical power has been one of the most popular side channels. All electronic systems use power to operate. In physics, power is defined as "the rate of doing work". As discussed in Section 2.1, power presents researchers and scientists with a non-intrusive way of determining how much work a system performs. Consequently, this information may be used to make various inferences regarding the system.

In our work, we analyze power consumption of high performance computing nodes and study how computation patterns of programs are manifested in this side channel. Our objective is to study these patterns and leverage them as fingerprints for program behaviors (i.e., anomaly detection). This will enable identification of a program given a power signature sample.

While power is a reflection of the amount of work performed by a system, there are several challenges with using power signature as a fingerprint for programs. The optimization techniques, both hardware and software, built into modern systems can impact power consumption. For example, branch prediction can cause a differente number of instructions to be executed between two executions of the same program using identical input (e.g., increase in the number of instructions executed when prediction is incorrect). In addition, the location of the sensor can play a significant role. Measuring the power consumption at the CPU (e.g., via intrusive probes) will result in different readings than when the power is measured at the system unit level, or even further, at the rack level. In the latter case, it is important to consider how the architecture of power adapters and supplies can affect the power measurements.

For our experiments, we leverage data recorded from a high frequency micro-phasor measurement unit (μPMU) [59]. μPMU devices are capable of providing synchronous voltage and current data, for all three phases. The data is sampled at 512 samples/cycle but reported as root mean squared data at 120 Hz.

These devices are designed to be placed in-line with the system's power supply (or cable powering the system). For example, in our setting, a single μPMU is used to monitor the power distribution unit (PDU) that powers a HPC compute rack of 36 Condo compute nodes. In some cases, it is possible that multiple PDUs provide power to a single rack. In such cases,

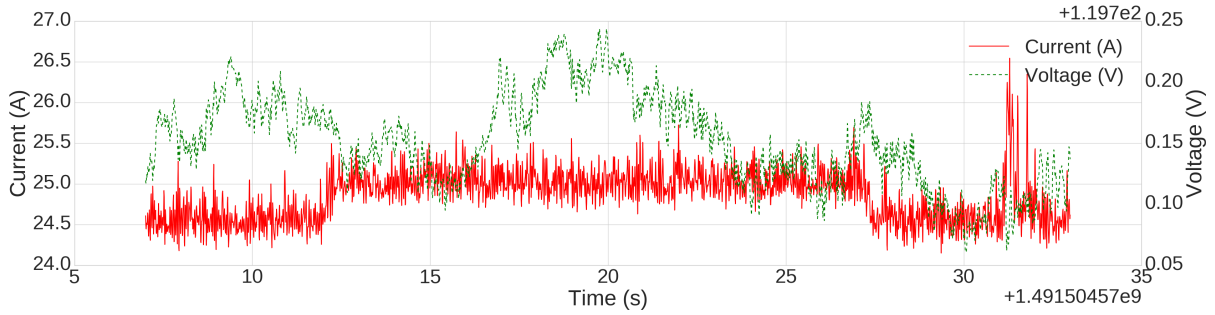


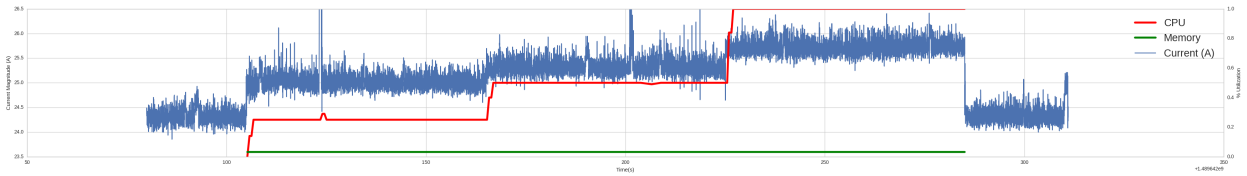
Figure 4.1: Diagram of current and voltage during the execution of OpenSSL SHA-1 hashing.

to achieve full resolution into the power consumption of the nodes, two measurement devices are required.

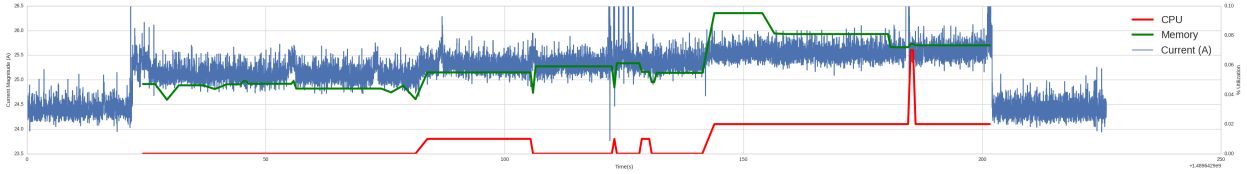
4.2.1 Initial Experiments and Observations

We begin our study by performing several small scale experiments using common programs, such as hashing tools, stress-test scripts, video encoding utilities, and even a prime factorization program. The objective of these experiments was to determine how system behavior impacts power consumption. In particular, we are interested in the impact of overall system activity on current and voltage as well as individually, the impact of CPU and memory activity on current and voltage.

Hashing is an established method of stress-testing modern processors. For our testing, we rely on the OpenSSL speed SHA-1 hashing utility. This program performs SHA-1 hashing for 3 seconds for several block sizes, in increasing order. Figure 4.1 shows the current and voltage data collected during a single execution of the OpenSSL tool. In the figure, we can see that the current magnitude time series resembles a step function. The edges of the step functions are aligned with the beginning and end of the experiment. On the other hand, the variations in voltage are minor (hundredths of a Volt) and show no recurrent pattern. Other than the behavior at the 25 second mark, we see no relationship between the current and voltage. From the figure, it is immediately obvious that current best follows the behavior of the CPU. Voltage must be relatively constant as to not cause electronic failures. Examining the power data from the μPMU during the execution of the OpenSSL speed SHA-1 hashing program we can observe that current magnitude best follows the behavior of the program.



(a)



(b)

Figure 4.2: (a) CPU stress test shows that the amount current drawn increases with the utilization in CPU (b) During a memory stress test, we see that current drawn also increases although there is some level of CPU utilization

We continue by studying the differences in current footprint between CPU and memory activity. To achieve this, we utilize the *stress* UNIX utility to impose heavy loads on the system and compare the impact on current consumption. The stress utility is a simple C program designed to stress-test various aspects of the system, including CPU, memory, and disk [60]. Since the compute nodes are not equipped with disk drives, we focus solely on CPU and memory.

Figure 4.2 depicts the results of the test. In Figure 4.2a the current magnitude, CPU, and memory utilization are depicted during a CPU-only stress test. Specifically, the utility spawns 2, 4, and 8 workers consecutively, each for 60 seconds. Each worker performs square root operations, an operation known for its CPU intensity. The diagram shows how the current magnitude increases as the CPU load increases. We also see that the memory overhead is minimal.

The same utility is used to impose a memory-intensive task. In this case, the utility spawns 1, 2, and 4 workers consecutively, each for 60 seconds. For a memory test, the workers perform a series of $malloc()/free()$ operations, for 256MB. In Figure 4.2b, we see the current measured during the execution of the test, as well as memory and CPU utilization. Looking at the figure, we can see how the amount of current drawn is also impacted by memory activity. It is important to note that there is also some CPU activity during the execution of this task.

4.2.2 HPC Experiments

4.2.2.1 Hardware and Software Setup

Our final objective is to identify programs running on a HPC platform given samples of power data. To recreate the real-life environment as closely as possible, we run scientific benchmark programs on a compute rack at the Lawrence Berkeley National Laboratory. The rack is part of the "Lawrencium" compute platform and it includes 36 Condo compute nodes. Each node is equipped with quad-core Intel Xeon E5530 processors and 24 GB of random access memory. The nodes are inter-connected using QDR InfiniBand, a well-established computer-networking communications standard for HPC platforms. The nodes are not equipped with any physical storage drives. Our application test suite is comprised of the NAS Parallel Benchmarks (NPB) [61] and the Trinity-8 Procurement benchmarks [62]. The NPB are a small set of programs, derived from computational fluid dynamics applications, designed to evaluate the performance of parallel supercomputers. These benchmarks implement scientific computations, similar to codes that are usually executed by scientists. Specifically, the benchmarks used in our analysis represent a block tri-diagonal (BT) solver, conjugate gradient (CG) program, embarrassingly parallel (EP) code, fast Fourier transform (FT) kernel, integer sort (IS) program, scalar penta-diagonal (SP) solver, lower-upper Gauss-Seidel (LU) solver, and a multi-grid (MG) application. The NPB define eight problem size classes (i.e., S, W, A, B, C, D, E, F in increasing order of complexity). The problem size class defines the input and parameters used during execution and consequently adjust the complexity of the computation. For example, class S is for small-scale experiments, while class W is for workstation size and class F represents the largest test problems. In our experiments, we compile the codes of version 3.3.1 for problem size classes C and D. We use two problem size classes since for class C, the execution of a few programs is relateively short (i.e., few seconds). At the time of this writing, there are twelve benchmarks total, eight of which we use in our analysis. Four of the benchmarks are eliminated because they either are not available for larger problem sizes or are designed to test I/O performance, which is not visible to our power sensors, since they have visibility only to CPU racks.

The NERSC-8 Trinity Procurement benchmarks are a set of programs created by the National Energy Research Scientific Computing Center (NERSC). NERSC designed this set

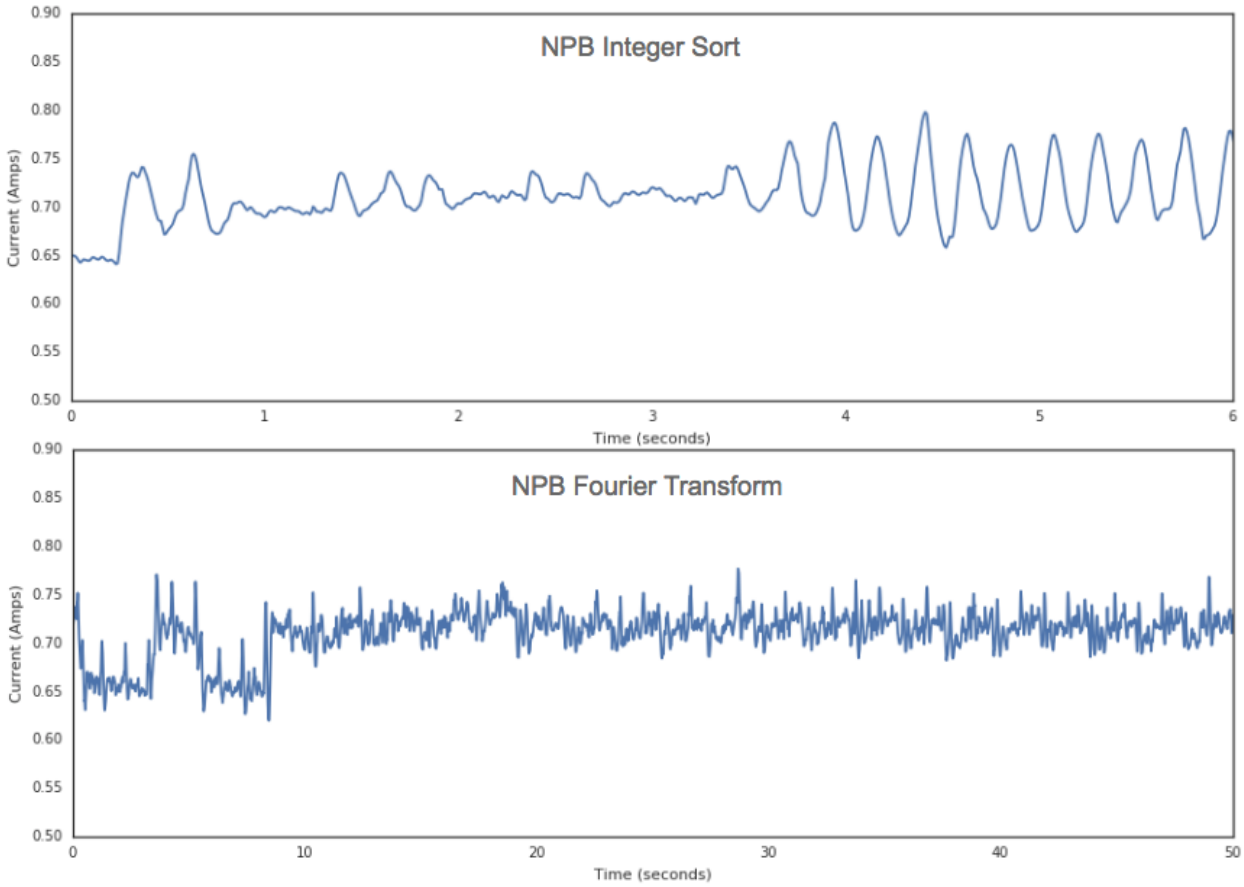


Figure 4.3: This figure depicts the mangitude of current over time during the execution of the Integer Sort (IS) and Fourier Transform (FT) benchmarks of the NAS Parallel bench suite.

of mini-applications to mimic the behavior of real applications and uses these benchmarks for system evaluation and acceptance testing. We use 4 Trinity programs in our analysis: a parallel algebraic multi-grid solver for linear systems (AMG), 3D Gyro-kinetic Toroidal code (GTC), finite element generation, assembly and solution code (MINIFE), neutral particle transport application (SNAP).

For each program we gather several samples by running each program independently multiple times. Since these applications are designed as benchmarks, the input is constant between runs.

Before performing our analysis, we examine the current fingerprint of these benchmarks. Figure 4.3 visualizes the current magnitude time series for two of the NAS parallel benchmarks. From the diagrams, we can determine that both time and frequency domain features

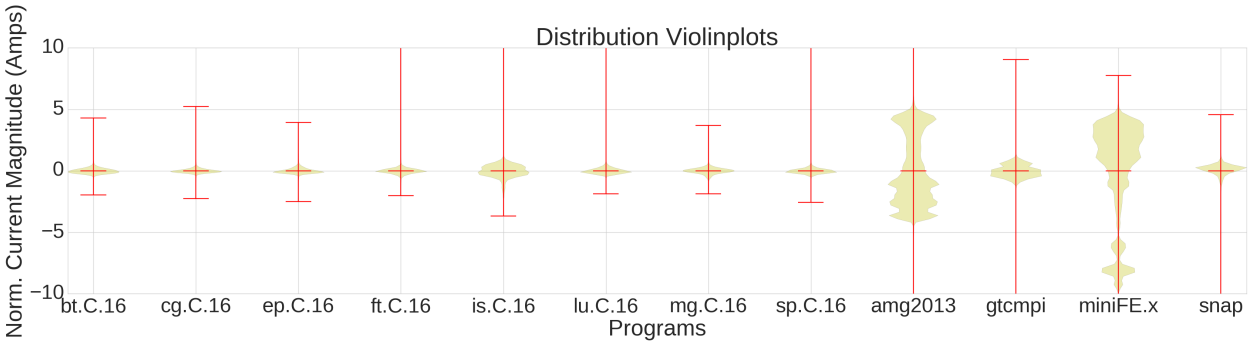


Figure 4.4: Diagram visualizing the mean-shifted distribution of the current magnitude time series for each of the benchmarks.

can be used to describe the time series.

As Figure 4.4 shows, the distribution of the mean-shifted current magnitude differs across the programs. Some programs, such as the Block Tri-diagonal (BT) program have a Gaussian-like distribution, while others have distributions that are almost bimodal.

The programs were executed concurrently and as such, we believe the noise to be similar between at least any two consecutive programs.

4.2.2.2 Analysis Framework

Side channel analysis is not a trivial process. Despite the power of the machine learning algorithms, there is a substantial amount of finesse involved in machine learning, particularly with pre-processing the data and feature engineering. The classification results of our analysis are the product of a long iterative process involving failed experiments followed by minor adjustments.

Although there is an abundance of machine learning libraries, there is a lack of frameworks for conducting machine learning experiments, especially time series analyses. In most cases, when performing such analyses, researchers will test different pre-processing (e.g., standardization, normalization, scaling) techniques as well as different sets of features. Too often, this results in redundant code, despite the similarity in the overall process.

Motivated by the lack of time series analysis frameworks, we design and implement a customizable time series analysis framework in Python. Our framework leverages the observation that the analysis process is always comprised of the same iterative steps. The difference between two experiments lies not in the overall workflow of the analysis process

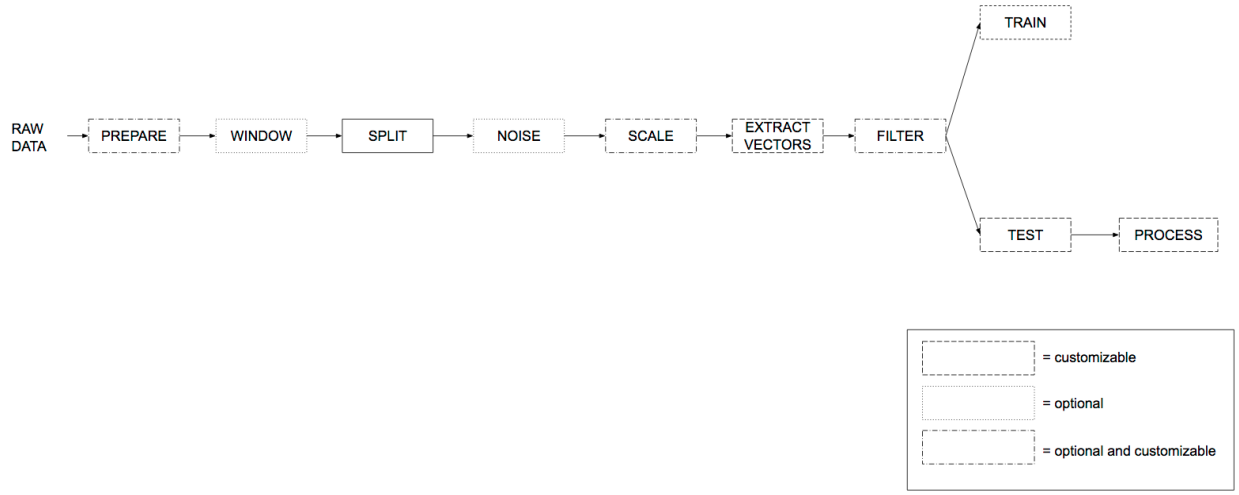


Figure 4.5: Diagram describing the architecture of the time series analysis framework

but rather in the details of the analysis. As shown in Figure 4.5, our framework defines the analysis process as a sequence of ten steps: Prepare, Window, Split, Noise Addition, Scaling, Feature Extraction, Filtering, Train, Test, and Result Processing. The framework is analogous to a Java interface, except some functions are predefined while others can be defined by the user.

These ten stages were selected carefully to keep the framework general while also enabling customization. The framework defines two types of analysis components: predefined and customizable. Additionally, each component is either optional or mandatory. For example, any time series analysis involving machine learning should include a mandatory step where the data is split into training and testing sets. This step is predefined by the analysis framework, although the user can control the split percentage.

Other mandatory steps, such as feature extraction, are fully customizable by the user. This enables users to easily adjust the feature set. The analysis framework also provides other potentially interesting features, such as windowing and generation of synthetic noisy samples (by adding samples together). The windowing step divides a time series sample into windows of user-defined length. Window overlapping is also supported. Each analysis component is defined by one (or more) function(s). If the component is customizable, the user must provide a reference to the function to be used. The only requirement is that the user considers the format of the function input and the function output.

To create an experiment, the user must initially define functions for each analysis component. Once these functions are defined, a single experiment can be composed using only a few lines of code by instantiating a new analysis object and providing the appropriate parameters. To alter the experiment, the user can instantiate a new Analysis object with a different set of parameters. For example, changing the pre-processing function is done by replacing a single parameter. Similarly, the feature set used by the classifier is defined by a list of user-defined feature extraction functions. This list is provided as an argument to the Analysis instance and modifying the contents of this list results in a new experiment.

The framework guarantees that code redundancy is reduced and allows users to focus on the details of the analysis steps, rather than engineering the entire process.

4.2.2.3 Feature Selection

For our analysis we rely on machine learning to build fingerprints for programs based on features of their current time series. Specifically, we rely on both time and frequency domain features.

Selecting time domain features is non-trivial. The well-known Anscombe's quartet elegantly describes this problem. Specifically, Anscombe's quartet presents four data sets that share statistical metrics, some of which are identical (i.e., mean, variance) while others are extremely close. Examining the violin plots depicted in Figure 4.4, it becomes evident that particular statistical metrics, such as minimum and maximum values are quite similar for some programs.

Additionally, time domain features are very sensitive to noise. In our problem setting, noise refers to the fact that the amount of current consumption, as measured, is a holistic representation of the system's activity. In other words, the current magnitude time series reflects all of the jobs running across all rack nodes. To establish a representative set of time domain features, we compare their distribution across all samples from all of the programs. The following features represent the final set of time domain features: maximum value, percentiles, auto-correlation coefficients with different lag values, standard deviation, empirical cumulative distribution function, stationary coefficient, discriminant function analysis, absolute energy, and ratio of variance to standard deviation.

As evident in the visualizations of the current magnitude time series, many samples

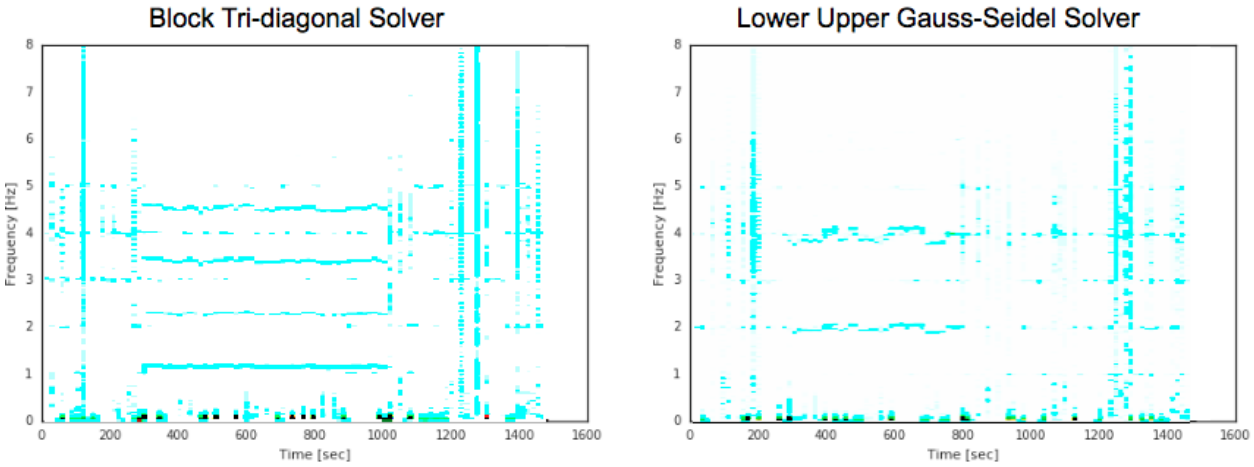


Figure 4.6: Spectrograms of current magnitude time series representing two of the NAS Parallel Benchmarks, the Block Tri-diagonal Solver (BT) and the Lower Upper Gauss-Seidel Solver (LU).

exhibit distinct frequency patterns. To visualize this more clearly, we generate spectrograms for two of the NAS parallel benchmarks. A spectrogram is a visual representation of the spectrum of frequencies of a signal over time. Figure 4.6 shows that the signal representing the current consumption of the Block Tri-diagonal solver benchmark has strong components in the 1.1, 3.5, and 4.5 Hz frequencies (i.e., horizontal lines from 300 to 1000 seconds at $y = 1.1, 3.5, 4.5$). On the other hand, the signal of the Lower Upper Gauss-Seidel solver benchmark is composed mostly of signals of 2 and 3 Hz frequencies.

For our feature extraction, we focus on three types of frequency analysis: discrete Fourier transform, power spectral density, and wavelet analysis. In comparison with discrete Fourier transformation, power spectral density estimates the power of various frequency components of a signal. To compute power spectral density, we apply Welch's method [63] to each sample. We use the implementation found in the SciPy [64] Python library (version 0.18.1). Welch's method is used to estimate the power of a signal at different frequencies and it does this by performing windowed short term discrete Fourier transform. Instead of using the raw power spectral density values, we extract the top four peak values (both frequency and corresponding power spectral density).

Any frequency analysis which leverages Fourier transform suffers of the problem of balancing time and frequency resolution. On the other hand, wavelet analysis does not suffer of this problem and can often be more successful in decomposing a signal. To perform wavelet

transformation, we leverage the Ricker wavelet function. We use the implementation found in the SciPy [64] Python library (version 0.18.1). For our features, we extract coefficients are different width values for the wavelet function. Ultimately, for our classification, we rely solely on wavelet coefficients.

As mentioned above, feature selection is a meticulous process that requires a lot of experimenting and finesse. The final features are selected using two types of analysis. First, we perform distribution and variance analysis of the feature values across the different samples in order to identify and remove features with values shared by many types of programs. We also use Gini impurity to determine how each feature contributes to the labeling of a sample. Gini impurity is a method used during the building of decision trees to measure the prediction power of the features. Gini impurity aims to measure the homogeneity of features and it does this by estimates how likely it is to mis-label a sample if the label is assigned randomly according to the distribution of the labels determined by the feature. When a feature is able to narrow down the label to only one option, its Gini index is 0. When there are multiple labels and each has an equal probability, the index is 1. Therefore, features with low index values are more valuable in classifying data.

After thorough examination of the features and several experiments, we converged on a set of 24 total feature values.[1]

4.2.2.4 Classification Methodology

For classification, we rely on the random forest machine learning algorithm. A random forest is an ensemble learning algorithm often used for multi-class classification. The random forests are constructed during the training phase. Specifically, to grow a random forest, subsets of features are selected (with replacement) and decisions trees are created for each feature subset. The label of the target sample is determined by the mode of the decision tree predictions. Unlike decision trees which are very susceptible to over-fitting, random forests reduce the bias by using a bagging process. Bagging occurs at two levels: data selection and variable selection. Using bagging, the construction of each tree uses a different subset of data and as well as a different subset of variables.

In our experiments, we rely on the *scikit-learn* [65] (version 0.17.1) Python machine

[1]Some features (e.g., percentiles, wavelet coefficients) have multiple feature values

Table 4.1: Table listing classification results of clean current magnitude samples using the full sample approach.

Program	**Precision**	**Recall**	**F Score**
BT	92.86	94.45	0.94
CG	82.85	92.62	0.87
EP	88.18	88.59	0.88
FT	96.00	99.00	0.97
IS	97.27	97.35	0.97
LU	88.75	98.75	0.93
MG	99.09	89.15	0.94
SP	92.00	82.23	0.87
AMG	100.00	97.65	0.99
GTC	100.00	100.00	1.00
MINIFE	100.00	100.00	1.00
SNAP	98.75	96.47	0.98

learning library for both the random forest implementation as well as tools to process results (e.g., compute accuracy, precision, recall scores). The modular and robust design of the scikit-learn library fits our analysis framework adequately.

As an initial test of our objective as well as a baseline comparison for our approach, we first classify clean, full samples. Clean refers to the fact that only one program was executed by the rack during the collection of the current data. Full simply means that we use the entire current time series sample representing the execution of a program, from beginning to end. The results are described in the Table 4.1. Overall, the classification achieves 94.5% accuracy with an approximate out-of-bag (OOB) error of 5%. As the table shows, most programs have precision and recall in the range of 90 to 100%. Precision is a representation of the fraction of retrieved entities that are relevant while recall represents the fraction of relevant entities retried. More simply put, precision gives us the ratio of true positives to the sum of true positives and false positives. On the other hand, recall computes the number of entities of a particular class correctly classified (true positives) out of all entities of that

class (true positives and false negatives). It is important to make two other observations. The NERSC-8 benchmarks, listed at the bottom of the graph, have higher scores than those of the NAS parallel benchmarks. This may be explained by the higher resource utilization exhibited by those programs, which translates to more prominent features in their current footprint. Tangentially, the results show that some programs are more easily identifiable than others, particularly CG, EP, and SP. The current magnitude time series for these benchmarks are also visually similar and examination of the confusion matrix shows that these three benchmarks are often mis-classified as each other.

4.2.2.5 Noisy Experiments

Our approach thus far neglects the issue of noise. We define "noise" as disturbances in the data recorded by the μPMU data due to natural phenomenon or due to other processes running on the computing platform. Although we refer to concurrent tasks as "noise", in a normal, production HPC platform, it is rare for only one job to run on on a single rack at a given time. Considering the μPMU is monitoring a 36 node rack, it is normal for the sensor to capture the power consumption of several jobs at any given time.

Noise introduces an additional challenge. When programs are executed serially, it is relatively trivial to determine the beginning and end of a new task. In contrast, when programs are executed concurrently, establishing the start and end of a new program is difficult. One option is to identify abrupt spikes and sags in the current time series. However, correctly associating each spike with its corresponding sag is non-trivial, even if considering the magnitude of such changes. In our experiments, many of the NAS parallel benchmarks exhibit similar magnitude changes at the start and end of the execution.

To combat these challenges, we modify our approach by splitting each sample into equal-length windows. The key idea is similar to Hart's *switch continuity principle*. The switch continuity principle states that "in a small time interval, we expect only a small number of appliances to change state in a typical load" [15]. We believe similar principle to also apply in our setting. In other words, we believe that in small time intervals, even if several programs are running in parallel, only a small subset of the programs are exhibiting a change in behavior.

Despite improvements in noisy classification, by treating each window independently

some information about order and time-dependency is lost. We attempt to introduce some of that information into our vectors by grouping windows into sets of consecutive windows. In other words, we construct feature vectors using n smaller feature sub-vectors, each sub-vector describing one of the n consecutive windows. Each sub-vector is composed of the above mentioned 24 features.

When using window sets, our classification will label each window set. As expected, a program sample is composed of several window sets. Consequently, we label each sample using by the *majority* of the window set labels. In the case of a tie, we remove the correct label from the set of predicted labels and pick the predicted label with the highest count. We choose to do this as a way to enforce fairness. It is important to note that the strategy used for assigning sample label can easily be modified.

Experiments support our hypothesis that coupling windows together produces better results. Particularly, the average accuracy for full sample, windows, and window sets classification are 98.48%, 93.12%, 98.86% respectively. To provide a fair comparison, these results only reflect 6 of the 12 programs. The 6 programs are the only programs with runtime duration longer than 18 seconds (minimum of two sets of windows, with three 3-second windows). As expected, the accuracy drops when going from full samples to windows but improves when classification is performed on window sets. Our experiments also show that the average accuracy drops significantly more when performing classification on full samples versus windowed sets. For example, for noisy samples generated by combining two samples (one target, one noise), the full sample classification average accuracy drops to 47% (from 98%) whereas the window sets classification average accuracy decreases to 81.06% (from 98%). As expected, classification of window sets samples is more resistant to noise.

Despite these improvements, some windows are mis-classified. We perform several experiments to determine the root cause of these mis-classifications. Our hypothesis is that some windows represent segments of power signatures that are common among many different programs ("common" windows). Visual examination of the current signatures shows that many programs exhibit windows during which activity is seemingly constant, without any apparent pattern or identifying behavior.

To test our hypothesis, we perform two tests. First, we study the prediction score for

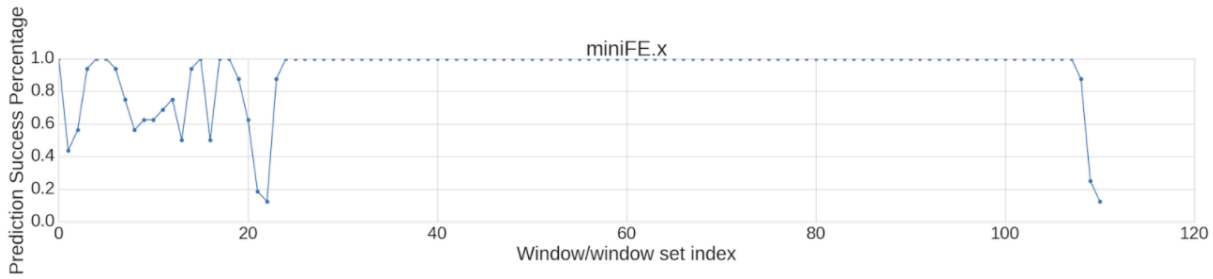


Figure 4.7: Diagram depicting the variation in the classification success rate across the windows of the MiniFE program.

windows of each program. The objective is to identify for each program, which windows are mis-classified. Next, we examine the confusion matrices to determine how windows are mis-classified. In particular, we wish to determine if the mis-classifications are spread throughout multiple labels or whether given our features, windows of certain program sub-groups are indistinguishable.

The results indicate that our hypothesis is true for short windows (i.e., 1-2 seconds). As Figure 4.7 shows, certain windows are classified with higher success rate than others. However, after performing several experiments with various window sizes, we determine that the false positive and false negative rates depend on the window size. As the size of windows increases, there are fewer windows mis-classified. While the mis-classification when using short windows span multiple labels, the opposite occurs as the window size increases. Specifically, the confusion matrices show that mis-classified windows do not spread across several classes.

To replicate noisy, real-life behavior, we generate synthetic noisy samples by combining electrical current samples of various programs. The synthetic samples are constructed by first selecting a target sample from the pool of samples. Then we randomly select a "noise" sample from the subset of samples with label other than the target sample and convolving it with the target test sample starting at a random location. It is important to make the location at which noise gets introduced random. The purpose of this is to mimic the unpredictable behavior of new programs beginning their execution. We choose to create noisy samples synthetically in order to minimize our usage of the HPC rack.

Table 4.2: Table describing the results of classification of noisy current magnitude samples using the window sets approach. The scoreless programs have execution times too short for the selected window set size.

Program	**Precision**	**Recall**	**F Score**
BT	79.55	86.48	0.82
CG	-	-	-
EP	24.82	65.65	0.33
FT	-	-	-
IS	-	-	-
LU	72.81	98.45	0.83
MG	88.89	44.14	0.59
SP	78.98	66.78	0.72
AMG	99.57	96.60	0.97
GTCMPI	94.30	64.78	0.78
MINIFE	96.83	94.30	0.97
SNAP	-	-	-

4.2.2.6 Results

We first train our random forest model on clean window-sets and test on the synthetically generated noisy samples. As previously mentioned, the level of noise refers to the number of "noise" samples added to the target sample.

Table 4.2 describes the results of classification of noisy samples with noise level of 1. The results presented below reflect a window size of two seconds with four windows per window set. Unfortunately some of the benchmarks have short run-times (e.g. IS program runs for less than 10 seconds) and are excluded from classification. As the table shows, the scores vary, although the classification approach performs well for a few programs. One interesting observation is that the classification performs better for the Trinity benchmarks, than compared to the NAS Parallel programs. This observation as well as others are discussed in more detail in Section 4.2.2.8.

It is important to note that these results reflect the classification of samples after each

sample is labeled based on the labels of its window sets. In other words, for every sample, we classify the window sets and collect the predicted labels. The sample is then labeled using the majority of the predicted window sets labels. In most cases, for a given samples, 70% of the window sets are correctly classified. This depends on the noise type and level. In some cases, as expected, the noise drowns out the target sample and causes classification to perform poorly.

We also perform tests to analyze how classification performs as the level of noise is increased. In Figure 4.8, we visualize average accuracy, across all programs, as the noise level increases. In addition, the performance of our classifier is compared with two baselines. As a lower bound, our results are compared to a random guessing strategy (visualized by a red dotted line). As an upper bound, the classification results are compared to the average *mutual information* between clean target samples and the corresponding synthetically generated noisy samples. Mutual information is an information theoretic measure of the dependence between two random variables. Specifically, it uses entropy to quantify the amount of information obtained about one random variable, given another random variable. In our experiments, we use mutual information to measure how much information is lost after noise is added to a sample. For each sample, we compute the mutual information between the original target "clean" sample and the corresponding "noisy" sample used during classification. The mutual information is displayed in Figure 4.8 using a dashed green line. From the figure, we can see that our classification performs considerably better than a naive random guessing strategy and is relatively close to the the upper bound described by the mutual information metric.

To better understand our results, in Figure 4.9, we visualize the precision and recall for each individual program.

We make several observations from Figure 4.9. First, we can see that for most programs recall decreases faster compared to precision as the noise level is increased. In most cases, noise causes more false negatives than false positives, causing recall to decrease more than precision. We also can see that precision varies for some programs as the noise increases. We believe this to be a side effect of the randomness involved in the noisy sample generation process. Specifically, the "noise" is added at a random location within the target sample.

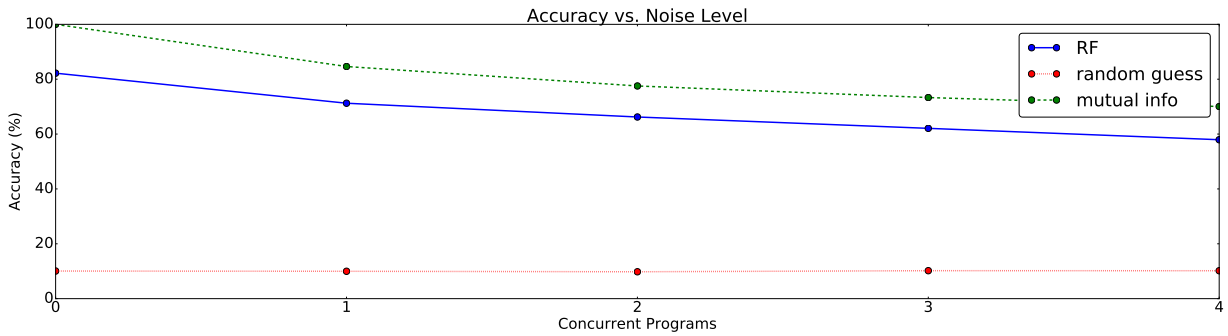


Figure 4.8: Diagram visualizing the impact of noise on the average accuracy across all programs and a comparison of the classification results with random guessing and mutual information

The value of the random location generated will affect how many window sets of the target sample will be tainted and consequently mis-classified. This side effect is diminished as the number of experiments increases.

Another interesting observation is that the precision and recall for some programs, such as AMG and GTCMPI, decreases slower than for other programs as the level of noise increases. Manual analysis shows that in many cases the noise samples drown out the target sample. In particular, the NERSC-8 programs tend to drown out the NPB benchmarks. This is explained by the fact that the NERSC-8 programs utilize significantly more resources hence producing more prominent patterns in their power signatures. Consequently, adding the NPB programs as noise has little impact on the classification.

4.2.2.7 Results Evaluation

Our results are evaluated using two separate methods. We use the out-of-bag (OOB) score to estimate the prediction error of the random forest classifier. Specifically this method allows us to estimate the prediction error on window sets. As mentioned previously, we aggregate the predictions for window sets for a single sample into one prediction, based on the majority of the window set labels. The out-of-bag method works by constructing trees using different bootstrap samples from the original data. After the trees are constructed, they are tested using the remaining samples which were not used during classification. The average OOB score for full clean sample classification is 96% (or 4% OOB error). For windowed-sets classification the average OOB score is approximately 95% (or 5% OOB error).

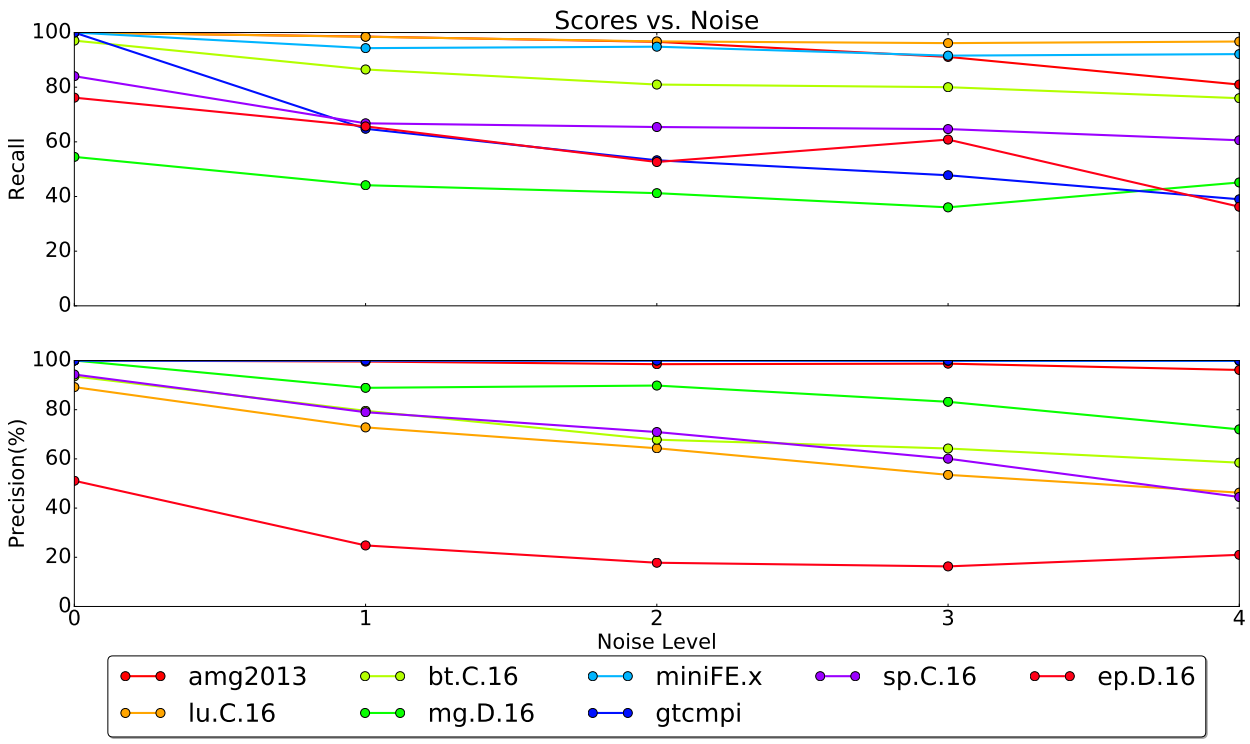


Figure 4.9: Diagram visualizing how noise affects the recall (top) and precision (bottom) of each individual program.

We further validate our results using the three-fold cross validation technique. For each experiment, the samples are divided into three equal size "folds" and the experiment consists of three classification runs, where training is performed on two of the three folds and testing is done on the remaining fold. The experiment is repeated several times, especially for noisy scenarios (since the placement of the noise is randomly chosen). The results presented here are the averaged scores across several such runs.

4.2.2.8 Program Behavior and Prediction Success Rate

The results presented above demonstrate that our approach is capable of identifying certain programs more than others, even in the absence of noise. Indeed, the current magnitude time series for samples with low precision and recall are visually similar. Manual analysis of the features and classification result supports this and shows that our classification approach fails to distinguish between certain programs. The machine learning and features used undoubtedly drive the success rate of classification. However, we argue that the success of classification techniques is limited to a large extent by the data. Thus we wish to understand, at a more fundamental level, the factors driving the ability to classify programs based on

their current signatures.

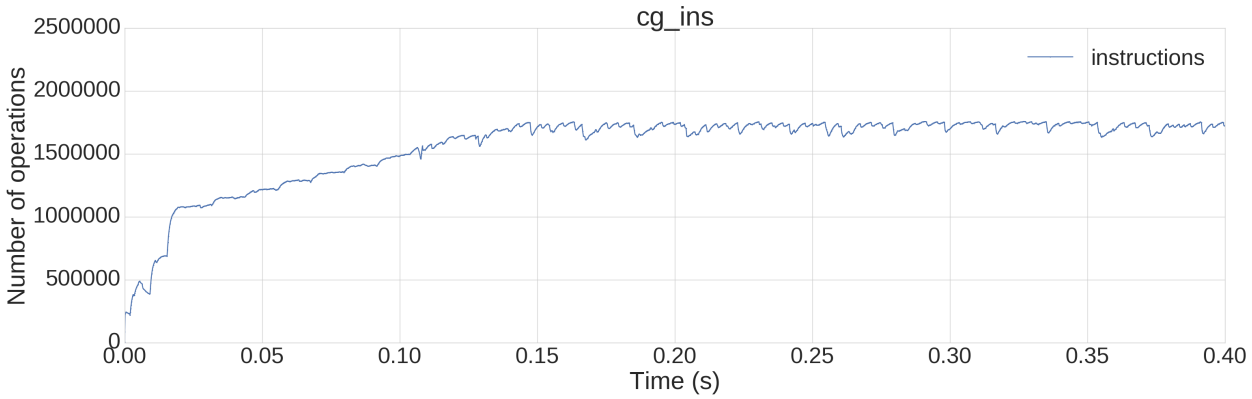
Considering our objective, there are several factors which can impact the classification rate:

- **Sampling rate**: How does frequency of power data impact classification?
- **Program behavior**: How does the behavior of a program affect classification? Are there patterns or observations about a program's behavior that improve its chances of being identified given its power footprint?
- **Other components**: What role do other components, both hardware and software, play into our analysis?

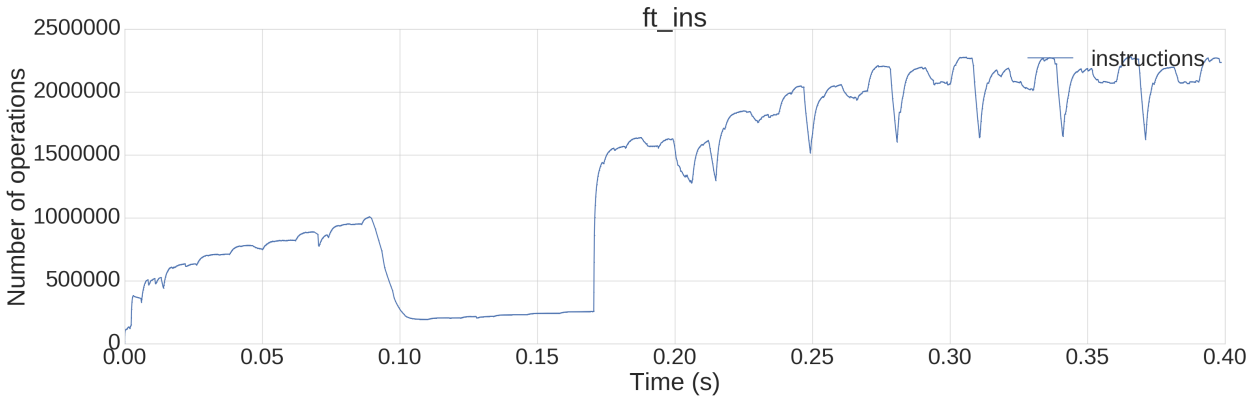
To help answer these questions, we leverage hardware performance counters. Performance hardware counters are special-purpose hardware registers used by processors to record hardware events. These registers can be programmed by a user to record several hardware events, for example the number of instructions retired. Most modern CPUs are equipped with such counters.

As mentioned previously, our analysis is built on the observation that programs' behavior can be described by a permutation of CPU intensive and I/O intensive periods. In our analysis, the hardware performance counters provide us with high-resolution data about a program's behavior (e.g., number of instructions executed, memory activity, etc.). Specifically, we are interested in hardware counters regarding CPU and I/O (mainly memory) activity. Although there may be other factors at play, comparing the hardware counter with the power data provides insight into the relationship between power consumption and system behavior. Additionally, the performance hardware counter information can be collected at relatively high frequencies (compared to the μPMU 120 Hertz power data).

Unfortunately, access to performance counters is not enabled for the high performance compute nodes used in our analysis. As such, for these experiments we rely on a separate system, a conventional desktop. The desktop is equipped with Intel Quad Core Q660 processor, 4GB of random access memory, and is running Ubuntu 14.04.4 LTS (generic 3.13.0-86 Linux kernel). We also execute the same benchmarks on the desktop and compare the (physical) current signatures with those obtained from the HPC rack.



(a)



(b)

Figure 4.10: Diagram of the number of instructions executed over time, during the execution of two NAS parallel benchmarks, (a) Conjugate Gradient program and (b) Fourier Transform program

Our experiments enable us to make two observations. Earlier in this chapter (see Figure 4.2), we showed that the current consumption follows behavior of CPU (and memory to a lesser extent). However, as Figure 4.10 depicts, there are patterns in CPU behavior, with respect to the number of instructions retired, that are not visible in the power data. For example, in Figure 4.10a we can see frequent dips in the number of instructions executed for the Conjugate Gradient NPB program. This pattern is absent from the current magnitude time series data.

There are two possible factors can explain this. First, there is a large discrepancy between the sampling rate of our power sensor and the operating frequency of the CPU. As a result, only significant changes in the CPU activity will be captured by the μPMU data. Second, it is possible that the power distribution and supply units limit the resolution of the power

data. This would not be a limiting factor if the power measurement technique involved probing the processors directly.

Other observations can be made by further comparing the performance hardware counter time series data of various programs. For example, it is apparent that even at the relatively higher performance counter sampling rate, programs that have higher classification rate exhibit greater changes in the magnitude of instructions retired. Also, it is important to note that the programs with the highest classification success rate exhibit variations in the behavior throughout their execution. For example, although there is a distinct visible pattern for the Conjugate Gradient program (Figure 4.10a), the pattern persists throughout the execution. On the other hand, a program like the Fourier Transform program (Figure 4.10b) manifests a few different patterns through it's execution.

4.2.3 Discussion and Future Work

As our results reflect, we believe it is possible to identify programs based on their power signature. It is important to note that some programs may drown out others, less resource intensive programs. We do not believe this to be a critical limitation. Our current approach may not be able to determine when a user is running a small script for non-scientific purposes, but it would be able to detect users running computationally intensive applications such as bitcoin miners.

From our experiments, we also deduce the window size can play a significant role in detection. While our window size was limited by the durations of the benchmarks executions, most HPC jobs are considerably longer, allowing for much larger window sizes.

In the future, we plan on continuing the work described above. First, we would like to test our classification approach on more programs, especially on the HPC platform. We hope to capture power consumption for real compute jobs from production HPC platforms and test our approach on the new data. Most of the work done to this point is analysis in the frequency domain using the raw data. However, we believe accuracy could be improved by filtering noise. In our work, "noise" is any concurrent program or device impacting power consumption that is unrelated to the target program. One of the most common ways of filtering noise in signals is using spectral subtraction. In spectral subtraction, the signal spectrum is adjusted using noise spectrum in a manner that improves the average signal-to-

noise ratio (SNR). In our case, we believe that the best way of estimating the noise spectrum for a given sample is by looking at the period before and after that sample (execution of the program). Using that noise spectrum, we can adjust the signal spectrum to optimize the SNR.

Additionally, we are interested in studying the relationship between classification accuracy and the distance between the sensor and the computing resource. In other words, we would like to determine how "far" the μPMU can be placed from the target computing rack and still obtaining a reasonable level of accurately with respect to fingerprinting running programs. The "distance" component has two main attributes: noise and loss of signal. There are significant differences when comparing the power consumption of a compute rack to that of an entire building. Buildings have hundreds of devices and machines that introduce considerable noise. However, it is also important to study how the loss of signal affects our approach and how the modeling techniques must be adapted to account for it. In summary, this study would have two contributions: demonstration of the ability to fingerprint activity on HPC platforms using power-related data and the study of impact of "distance" on the accuracy of such fingerprinting.

4.3 I/O Analysis

As presented above, CPU and memory activity of a program are reflected in the current drawn by the system during the execution of said program. In the next part of the work, we describe analysis performed on I/O data pertaining to programs in order to classify the activity.

4.3.1 Darshan

In a high performance computing (HPC) setting, efficiency is a priority. As a result, collecting data about the activity must be done with minimal overhead. Darshan [66] is a lightweight, scalable I/O characterization tool designed by Argonne National Laboratory specifically for HPC environments. It is implemented as a set of user space libraries which are linked to the source code during the linking phase of the build process. As such, Darshan does not require source code modification, works for both static and dynamic complication, and requires no additional supporting infrastructure.

Darshan captures information regarding POSIX, MPIIO, STDIO, HDF5, PNETCDF, BG/Q, and Lustre I/O operations that an application performs. In this study, we focus solely on POSIX I/O operations. This is because the other I/O libraries are rarely used by applications, as shown by Table 4.3. Features include number of operations, size of operations, time of first and last operation of a particular type, access patterns, and even switches between read and write operations. For file I/O operations, the Darshan log includes the name of the file. Each Darshan log generated also includes information about the job identification number, the start and end time of the job and the number of MPI processes utilized. Additionally each log is named such that it contains the name of the user who submitted the job, the program name, the date and also timing information.

4.3.2 Data Collection

Data was collected from the National Energy Research Scientific Computing (NERSC) center, where Darshan is enabled by default on two of their systems. There are two possible sources for data. One source is by executing benchmark applications. Benchmarks are an easy way of generating data, however compared to real scientific applications, their behavior is predictable. The second possible data source is by analyzing Darshan logs generated as

Type	% of logs	% of apps
POSIX	99.8%	78.7%
Lustre	53.1%	63.8%
MPI	9.28%	29.6%
PNETCDF	1.1%	7.4%
HDF5	0.3%	10.1%
STDIO	0%	0%
BG\Q	0%	0%

Table 4.3: Table describing the breakdown of I/O libraries used across 54231 Darshan logs from 108 total applications.

a result of compute jobs submitted by researchers. In comparison to the logs generated by benchmark tests, these Darshan logs provide a more realistic representation of the applications' behavior. Additionally, using these logs, the variance in behavior between executions of the same application (but with different input and/or parameters) can be studied. In our work, we focus on logs created as a result of researchers' compute jobs.

For our study, we examined 54,231 Darshan logs representing researchers' compute jobs over 30 days and 108 unique applications. Both application name and user name were extracted from the Darshan log names. It should be noted that two logs may share an application name but vary in their content (i.e. different inputs and parameters can cause the same program to behave differently).

As mentioned, each Darshan log represents one execution of a particular application. We assign a computational motif label to each Darshan log. This process was done manually by researching the application (by name) and inferring its computational motif based on the descriptions found. Out of the total 108 applications, we were able to classify 18 applications into only seven computational dwarfs, representing 1683 Darshan logs. Due to time constraints, we select a maximum of 30 samples per program, if available. A total of 331 logs were used for the analysis. Some other applications included popular unix utilities, preprocessing tools, or scientific codes that we were not able to label with confidence. Table 4.4 lists the applications and the computation dwarf label assigned.

Computational Dwarf	Application
Dense Linear Algebra	pstg3r, scalapack
Sparse Linear Algebra	superlu
Spectral Methods	getsfcensmeanp
N-Body Methods	gem, lammps, pmemd
Structured Grids	cactus, castro, hyperclaw3d, milc, nyx3d, pflotran, shengbte, wrf.exe
Unstructured Grids	gtc, suolsontest
Map Reduce, Monte Carlo	mcnp6, track3p

Table 4.4: Table listing 18 most commonly executed scientific codes and their corresponding computational dwarf.

4.3.3 Methodology

Darshan provides over 90 features for POSIX I/O. Many of these features reflect the program's logic (and its implementation). Such I/O features should be considered in our study. However, other features may also be impacted by factors unrelated to the program (e.g., operating system, file system, and even I/O load of the system during the execution of the program). We carefully select a subset of the total POSIX I/O features.

It is important to note that in some cases, comparing the values for a feature between two Darshan logs has little, if any, value. Two applications may exhibit different I/O behaviors, with respect to both the number of operations and timing of operations. An applications may read input from one or several files; it may write debugging information to a file. Even if the two logs represent the same application, the two executions may differ (e.g., due to differences in input size), causing the Darshan features to take different values. To combat these shortcomings, instead of analyzing the data provided directly by Darshan, we leverage the Darshan data to generate a total of 53. These features include total count of number of operations, histogram of read/write operations (with respect to size) as well as ratios (e.g., the average number of bytes per *read()* operation, the number of sequential reads/writes per total read/write count, etc.) which aim to abstract away some differences in I/O behavior. We believe these features are more useful in identifying patterns in I/O behavior. The

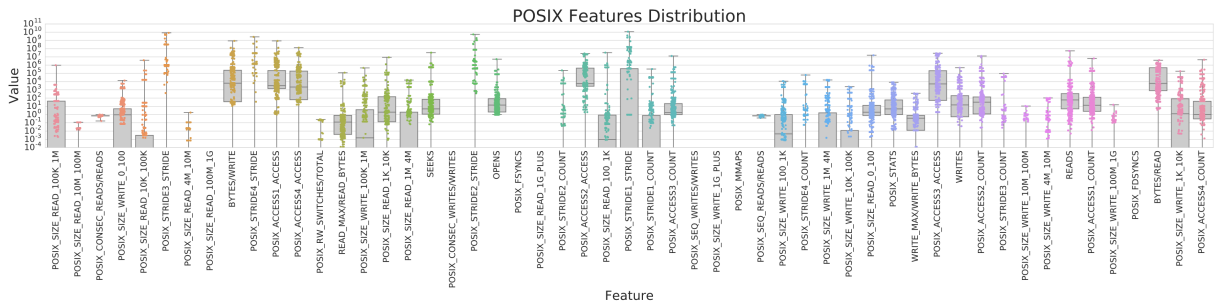


Figure 4.11: Boxplot and swarmplot representing the distribution of all 53 features across all 331 Darshan logs collected.

complete list of features is listed in Appendix A.

Figure 4.11 visualizes the distribution values (logarithmic Y axis scale) for all features across all 54,231 Darshan logs collected. While the distribution does not unveil clusters, it does show that the values for each feature do vary. To determine if this data can be used to classify a program as a member of a particular motif, we continue to examine how these features vary between executions of the same program, programs of the same computational dwarf, and finally between dwarfs.

4.3.4 Findings

For the Darshan data to be representative of the computational motif encompassed by an application, we expect several conditions to hold. First, we expect all programs of the same computational dwarf to exhibit some pattern(s). In order to successfully differentiate between computational dwarfs, we expect variations in the Darshan data (or computed features) across dwarfs.

After several experiments, we are able to make several observations, specifically:

1. Executions of the same program may exhibit different I/O behavior
2. Programs of the same computational dwarf exhibit different I/O behavior
3. Programs across different computational dwarfs exhibit different I/O behavior

We discuss each observation in more detail below.

4.3.4.1 Comparison of Executions of The Same Program

Analyzing the NERSC Darshan logs, we can observe that the I/O behavior of some applications vary between runs. We use the *binary name* component of the Darshan log filename to

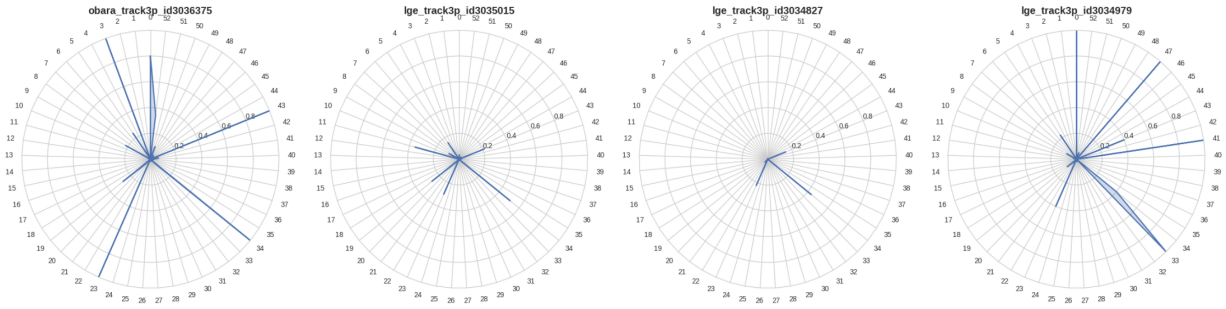


Figure 4.12: Diagram visualizing the I/O behavior of four different executions of the *track3p* application.

identify logs representing the same application. This may happen due to the input provided to the application, including parameters or even due to the way the application was compiled (if built from source). Figure 4.12 depicts the POSIX I/O behavior of four samples of the same application, track3p. Track3p is a 3D parallel particle tracking code for multi-pacting and dark current simulations. In this figure, each radar plot represents an individual Darshan log (i.e. execution of the program). Each vertex of the radar plot is associated with one of the 53 POSIX features. In the radar plot, each POSIX feature has a value in the range $[0, 1]$. To achieve this, we find the maximum value for each feature across all of the 331 and normalize each feature using its respective maximum value. This makes the comparison of features across different entities easier. In the figure, we see two types of differences in the pattern. One difference is in the length of the radius (i.e., value of feature). The other difference is in the subset of dominant features that reflect the behavior of the program. The former difference is not as concerning as the latter; differences in values for a feature may exist for several reasons. However, differences in prominent features tell us that the programs behave differently. Similar differences were observed for other scientific applications.

4.3.4.2 Comparison of Programs of Same Computational Dwarf

Despite differences in executions of the same program, we can also see variations in the behavior of applications of the same computational dwarf. For every application, the features were normalized as before, using the maximum value across all logs, and averaged across all the program's logs. Figure 4.13 depicts the POSIX I/O behavior of three applications of the same computational dwarf, structured grids. Similar to the previous figure, this diagram also highlights differences not only in the magnitude of features but also differences in the

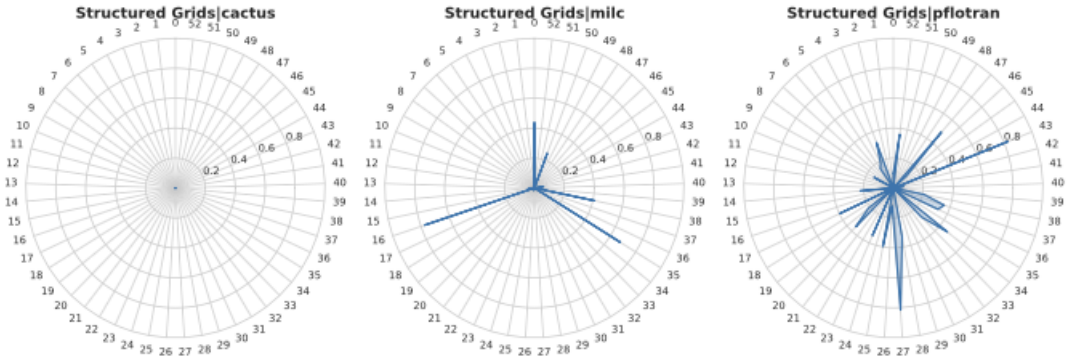


Figure 4.13: Radar plots representing two *Structured Grid* programs.

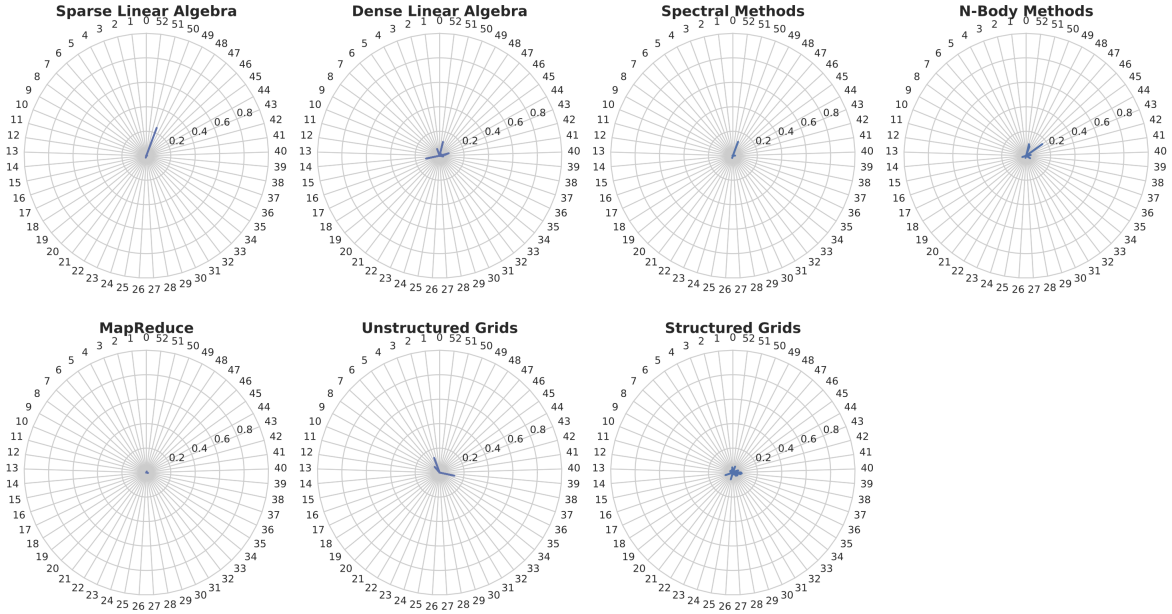


Figure 4.14: Radar plots representing I/O behavior of seven computational dwarfs. Each vertex of the radar plot represents a different feature.

prominent I/O features of the applications. For example, comparing the second and third plots, we can see that the magnitude of feature 38 (number of POSIX stats operations) is different. Additionally we also see that some features (e.g., 20, 23, 25) are dominant features in the third plot, representing the pflotran application, yet appear absent in the behavior depicted in the second plot of the milc application.

4.3.4.3 Comparison of Computational Dwarfs

Additionally, we investigated Darshan logs of applications from different computational dwarfs in order to determine if there are clear differences in the I/O behavior across im-

plementations of computational dwarfs. For this analysis step, we used many of the same features as described above. Figure 4.14 depicts the behavior of seven computational dwarfs. This figure shows that most computational dwarfs share similar characteristics with respect to their POSIX I/O behavior. Furthermore, in the process of building this figure, we observe that the patterns for some computational dwarfs, change as more programs are added to the set of samples representing the dwarfs. This observation further emphasizes the lack of patterns across programs of the same computational dwarf and the lack of differentiating behavior across dwarfs.

To further test for patterns in our data, we apply a density-based unsupervised clustering algorithm. Density-based clustering algorithms aim to find high density areas separated by areas of low density. This approach has several advantages. The algorithm scales well for very large number of clusters and number of samples per cluster. It also does not assume that clusters have a flat geometry or even sizes. Density-based clustering algorithms, however, do require two parameters: $min\_samples$ and ϵ. $min\_samples$ defines the minimum neighborhood size. The value for $min\_samples$ is selected based on knowledge of the data. For example, we know in our data set, some dwarfs are only represented by a single program. Consequently, we set this value to 1. ϵ represents the maximum distance between two samples for them to be considered in the same cluster. The value of ϵ is determined by looking at the Euclidean distance between points of the same partition. In other words, for every partition (e.g. computational dwarf), we calculate pairwise Euclidean distance between the parts. Then, we analyze the ϵ values across all partitions and choose a value such that for most samples, the algorithm will be able to find neighbors. The histogram depicted in Figure 4.15 shows that most vectors have at least one neighbor of the same class within an Euclidean distance of 0.1.

We apply the clustering algorithm in two different ways. First, we generate a feature vector representing every program in seven computational motifs: Sparse Linear Algebra, Dense Linear Algebra, Spectral Methods, N-Body Methods, MapReduce, Unstructured Grids, and Structured Grids. In other words, for every program, we take the feature vectors of all Darshan logs for that program and average them. The clustering algorithm identified four clusters, however the clusters did not have any significance. One cluster contained six of

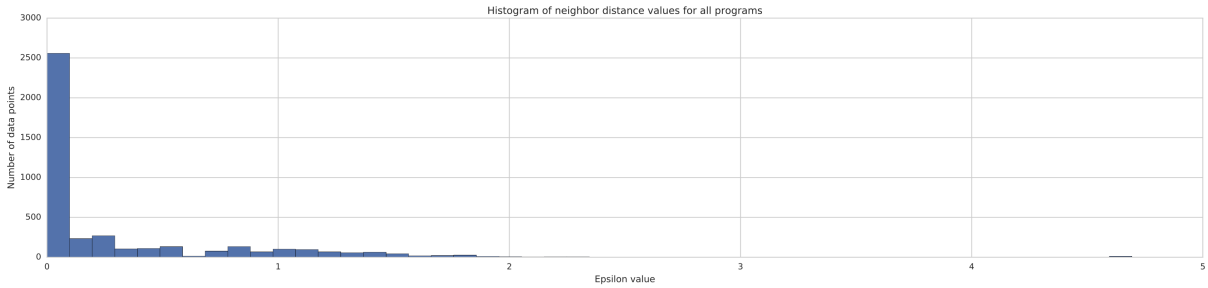


Figure 4.15: Histogram of pairwise Euclidean distances between vectors representing programs of the same computational dwarf.

the computational motifs and the other three clusters represented different programs of the Structure Grid motif. We also evaluated the homogeneity, completeness, and the adjusted mutual information score. Homogeneity is a metric between 0 and 1 which uses ground-truth class labels as a reference to determine how many clusters contain only data points which are members of a single class. Completeness tells us whether all members of a given class are elements of a single cluster. The adjusted mutual information score is a metric used to measure how much overlap there is between cluster (i.e., if clusters share members of the same class). It is adjusted such that it accounts for the fact that a few larger clusters usually have more mutual information than many small clusters. The values of these scores are 0.096, 0.234, and -0.140 respectively.

Our second approach, takes individual feature vectors for every single Darshan log and cluster them. For our second approach, we adjust the value of *min_samples* since now we more samples per class (each computational dwarf is represented by at least one program, each with several Darshan logs). The clustering algorithm was able to detect 23 clusters (despite only having 18 programs across seven dwarfs). The homogeneity and completeness of the clusters were poor, 0.262 and 0.237 respectively. However, the results do show that in some cases, Darshan logs representing the same program are clustered together.

4.3.5 Results

The observations above lead us to believe that classifying programs into computational dwarfs using Darshan data as presently collected on the machines that we were observing is not be feasible. In our experiments, we fail to see patterns in programs of the same computation type. Additionally, we can see that the pattern (as visualized by the radar

plots) of a computational dwarf changes as more samples (i.e., programs) are introduced. The clustering results further support our observations. The results are not surprising. File I/O behavior is not intimately related to the underlying computation performed by a system. On the other hand, inter-process communication or message parsing interface (MPI) communication may be, as shown by [42, 41].

On the other hand, our clustering results (second approach) hint at the possibility of identifying the underlying program based on Darshan data. While the behavior across executions of the same program may vary, in most cases the behaviors share common elements and the differences are in the magnitude of features (not the set of dominant features). We verified this by computing the pairwise Euclidean distance between samples of the same program and compare those values with the pairwise Euclidean distances between samples of one program versus all other programs. On average, the Euclidean distance between samples of the same programs were smaller than the Euclidean distance to samples of different programs. More investigation is needed.

There are several concerns with the data provided by Darshan. The lack of time series data makes it difficult to recreate a clear image of the program's I/O behavior during its execution. Additionally, Darshan provides a significant amount of POSIX I/O behavior, which mostly relates to file I/O. We believe that file I/O behavior may not be representative enough of the underlying computation performed by the application.

4.3.5.1 Limitations and Future Work

Our study has some limitations. Despite our best efforts, it may be possible that some executables were misclassified as the wrong computational dwarf. However, in most comparisons, there was little consistency between any two entities.

One may question our conclusion due to the naïve clustering approach used, which relies on Euclidean distance for identifying similar entities. Due to the severity in variations in the behavior of programs of the same dwarf, we have little reason to believe a more complex approach (such as neural networks) would perform better without overfitting the data.

Some applications may have options which causes the program to produce additional output. Considering our objective, such I/O operations introduce noise to the data, which may cause their respective I/O behaviors to differ. We do not have any mechanism for

detecting this. It is non-trivial to determine which subset of entities (e.g., files) are important for such analysis and which introduce noise.

While we may not be able to infer the computational motif, we believe that identifying some individual programs based on I/O data may be possible. One possible approach is to break down the I/O behavior contained by a Darshan log for a program into a series of partial, per file(/entity) I/O behaviors. To identify the program behind an unknown Darshan log, machine learning could be leveraged to identify the label for each partial I/O behavior. These labels may then be used to label the unknown log using a majority-rule method.

With respect to future work, we believe that it may be possible to classify a program into computational dwarf using CPU and memory activity information. We believe such information to be more closely related to the program's underlying computation type. CPU and memory activity information can be collected through hardware performance counters with low overhead in comparison to binary instrumentation techniques.

Chapter 5

Monitoring Internet of Things Platforms Using Side Channels

> *"... purchasing 10-20 different services from 10-20 different vendors using 10-20 different apps with 10-20 different user interfaces. If that's the way IoT goes, it will be a long tough slog to Nirvana."*
>
> – Bob Harden, Principal, The Harden Group

Previously we showed how side channel information can provide insight into the behavior of a program. In this part of the work, we show how side channel analysis can be used as a non-intrusive, device and technology-agnostic method for modeling and monitoring the behavior of heterogeneous Internet of Things (IoT) devices.

5.1 Background

The Internet of Things has become increasingly popular in both industrial and personal settings. Despite such devices invading our lives, many individuals do not understand the implications and intricacies of the Internet of Things.

"Internet of Things" is a phrase used to describe a platform of inter-connected, networked, computerized components. Such platforms are comprised of three types of components: sensors, actuators, and the "smarts". Sensors are devices which collect information about users and/or the environment. This information is then processed by the "smarts". The "smarts" represents the platform and resources intended to process the sensor data and

extract some meaningful information from it. In some cases, the "smarts" reside directly on the devices but increasingly they are located in the cloud. The processed information is sent from the "smarts" to actuators, which are devices designed respond to the data and perform a task. Sensors and actuators are often the same device. For example, in home automation, smart thermostats are both sensors since they collect temperature information and actuators since they control heat-ventilation-air-conditioning (HVAC) systems.

From a security and privacy standpoint, the combination of intrusive sensing abilities and Internet connectivity raises many concerns. Recent events demonstrate the need to secure IoT, yet introducing security mechanisms to such platforms is nontrivial. Securing IoT platforms is also a multi-faceted problem and it concerns not only securing the devices but also the "smarts" and the transmission of potentially private and sensitive data between devices and the "smarts".

At the time of this writing, security researchers as well as industry partners are discussing the possibility of creating a government agency responsible for regulating and certifying the Internet of Things. Other groups have been formed to design and implement standards for various aspects of the IoT, including communication protocols. The changes proposed require time before they will be adapted into existing IoT platforms. In the meantime, the state of IoT security continues to be poor and attacks on IoT platforms are on the rise.

In this part of the dissertation, we present a method for monitoring IoT sensors and actuators. Our method is not intended as a replacement for the proposed standards and certifications but rather as supplementary security mechanisms that can be applied in many settings without many modifications.

IoT devices are very heterogeneous in nature. Devices are spread across several producers, built using components from a plethora of manufacturers, utilize a wide variety of (often proprietary) protocols, and operate on different, often incompatible platforms. Considering all of the differences, it is extremely challenging, if not impossible, to create a single security solution for all IoT devices.

It could be argued that only one security monitoring device per platform is necessary. There are several problems with this. An obvious problem is the cost imposed on users. Perhaps more concerning is the explicit and implicit interactions between devices of the

same manufacturers and across various manufacturers. Without accounting for all possible combinations of devices, attack vectors could be missed. Overall, the lack of consistency makes it difficult to achieve a holistic view of a given IoT network.

Our method is based on the observation that, despite their differences, IoT devices have common characteristics. IoT devices are limited in their computing capabilities and rely heavily on network communications. While the network communications are often (although not always) encrypted, they present a technology and protocol-agnostic side channel for monitoring devices. In this part of our work, we investigate how device-to-device and device-to-cloud smart home network traffic can be used to monitor the state of the IoT devices. Recalling our definition of side channels as information exposed as a result of the program's execution on a physical computer, we categorize the encrypted smart home network traffic as a side channel since the value of the individual packets (the content) provides no insight into the state of the devices. Unlike deep-packet introspection, our method does not rely on packet content, but rather uses patterns in communication and packet attributes to infer the state of the devices. In our study, we apply traffic analysis techniques on network traffic generated by devices from Nest Labs [67], a subsidiary of Alphabet Incorporated [68], to identify key events in the life-cycle of the devices. Traffic analysis is the process of intercepting and analyzing network packets in order to deduce information from patterns in communication.

5.2 Experiment Setup and Devices

The experiments involve two smart home devices, the Nest Thermostat version 2 [69] and the Nest Protect smoke and carbon dioxide detector version 2 [70]. These devices are equipped with motion sensors and designed to detect human presence, in order to learn the daily schedule of the household occupants. The Nest Thermostat has several operation modes: *Home, Auto Away, Away.* The *Home* mode is designed to best accommodate the user, based on their preferences. The *Auto Away* mode is automatically activated when the device establishes that the household occupants are away. We believe motion sensor data from the Thermostat is used to determine the presence of occupants. The *Away* mode can be manually enabled by the user via the Thermostat or the smart phone application. The Nest Protect is also equipped with a light ring, called *Pathlight*, which activates when it is sufficiently

dark and human presence is detected. The events described above are communicated to the server and are presented to the user through a mobile application.

Our goal is to identify network patterns that allow us to infer information about the state of the devices. Specifically, we are interested in the occurrence of events such as Thermostat operation mode transition (i.e., *Home* to *Auto Away* and vice versa), motion detection, smoke alarm activation and Pathlight turning on.

5.3 Methodology

For analysis, we collect encrypted network traffic from the devices over the span of 60 days. The packet captures are consolidated into connection logs using the Bro [11] utility. In our analysis, we are interested in unique behavior that occurs during an event. While this can be extracted by looking at individual packet sizes (barring any re-transmissions), the same information is preserved when aggregating packets into single connections. This also has the additional side benefit of decreasing the number of entities processed during our analysis (i.e., there are fewer connections than packets). As mentioned, when considering connections as opposed to packets, it is important to consider re-transmissions, since they can impact the results. We observe a negligible number of retransmits in our traces.

Of the data, we analyze approximately 60% of the connection logs during our learning phase (i.e., discovering patterns) and use the remaining logs for testing.

Initially, we study the characteristics of the network traffic originating from the two devices. The Nest Thermostat communicates with 14 different hosts per day, on average. Protocols used include HTTP, NTP, DNS, SSL/TLS and ICMP. HTTP requests are used to obtain weather information and provide no value for our objective. For our analysis we ignore packets with protocols other than NTP and SSL/TLS. While the address of some remote services such as DNS and NTP change over time, some of the contacted IP addresses remain the same, particularly those for Nest cloud services. Figure 5.1 visualizes time series data of connections made by the Thermostat to one of the most frequented hosts. Each point represents a connection made by the Thermostat defined by the number of bytes transmitted. The figure highlights several characteristics. First, there are visible patterns with respect to the size of connections (i.e., payload bytes sent by the device). We see several instances of

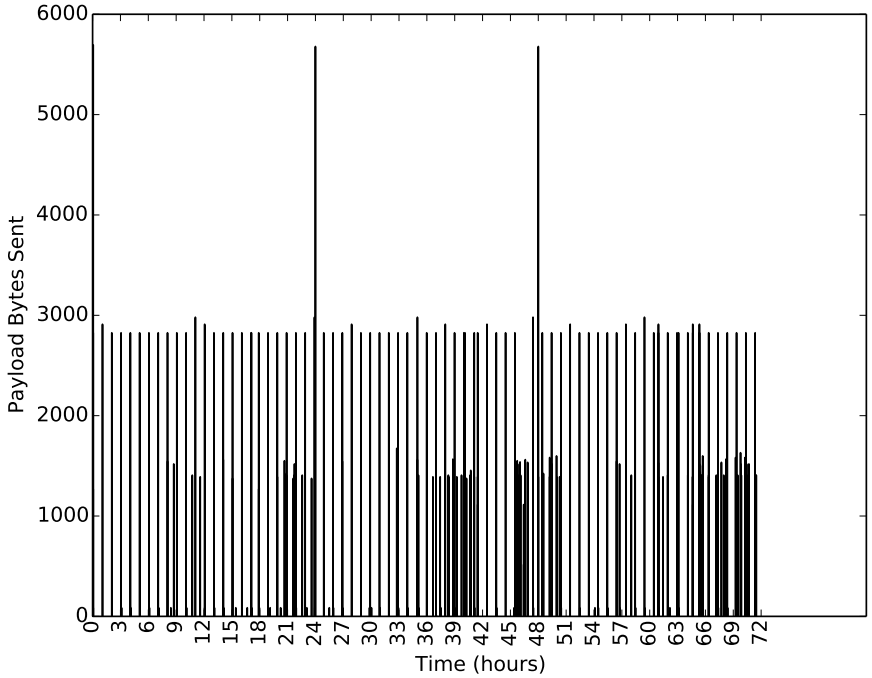


Figure 5.1: Diagram visualizing connections and their size (bytes sent) made by the Nest Thermostat to prominent destination over the span of three days.

packets of 1500, 2900 and 5700 bytes. The diagram also depicts temporal patterns. We can see that certain connections are made regularly. Additionally, we can see irregular bursts of connections.

To identify patterns in the encrypted network traffic, we leverage correlation analysis. Our correlation analysis uses a sliding time window approach with time windows of 10 seconds and a window displacement of 2 seconds. We build an $N \times N$ correlation matrix where N is the number of unique connections. A connection is defined by the destination IP address and the number of bytes sent by the source (i.e., Nest device). The matrix describes the number of occurrences of any two connections over the whole observation time. More precisely, the value of a given entry (i, j) in the correlation matrix is equal to the number of instances connections i and j occurred together in a 10-second time window. We create a mapping between connections and indices in the correlation matrix. To increase time efficiency, we also generate the SHA-256 hash of each connection (defined by the destination IP address and bytes sent) for connection equivalence comparisons. A two-dimensional correlation matrix

allows us to find pairs of correlated connections. The same approach is extend to a three dimensional matrix to allow the discovering of sets of three correlated connections.

Once the correlation matrix has been generated, we filter out some of the correlated connections. To begin with, we eliminate all connections with low correlation (i.e., number of occurrences lower than three). We determine that connections which rarely occur together are insignificant and can be disregarded. We also wish to discount regularly or frequently occurring connections. While such regular connections may provide useful insight, it is harder to correlate them with device state due to their regularity. Such connections may appear to be highly correlated with other connections simply because of the number of occurrences, however the relationship between such connections may have no special significance. Distinguishing between significant and insignificant correlations in such cases is nontrivial. For example, one of the most frequent connections made by the Nest Thermostat is to IP address 54.204.245.223 with 2826 bytes sent. This connection shows up in the top 12 most correlated connection pairs. To carefully filter out some of these correlations, we perform two tests. First, to identify regularly occurring, correlated connections, we look at the difference between the time stamps of when the connections occur together. If this difference is consistent (plus/minus a small threshold), we ignore these connections. We also look at frequency of each connection and for a given pair of correlated connections, if both connections are very frequent, we ignore the correlation.

While filtering narrows the search space, it does not produce the desired patterns. To learn the patterns, we obtained the exact time of the occurrences of the events of interest from the Nest web interface. Having such information available, we were able to draw associations between connections (or sets of two and three connections) and the occurrence of an event. To finally select between patterns of one, two, and three connections, we choose the patterns with the best accuracy.

5.4 Results

Our analysis was successful in discovering network traffic patterns for the Nest Thermostat operation mode switch, the Nest Protect smoke alarm trigger, and the Nest Protect Pathlight activation. Additionally, we were able to make an interesting observation relating to the

distribution of NTP requests. To verify the validity of our findings and obtain accuracy measurements, we automatically test the remaining 21 days of network traffic for the presence of the discovered patterns. The results of this test were compared against the ground truth, obtained from the Nest web interface. As mentioned earlier, a user can login to the Nest web interface and obtain a log of events (e.g., mode change, smoke alarm) and their associated time of occurrence, as recorded by the devices.

The details of each finding and the accuracy results are described below independently.

- **Mode Transition** One of the events of interests was the transition between modes of operation in the Nest Thermostat. If the transition between modes was reflected in a unique identifiable pattern in the network communications between the Nest Thermostat and the Nest severs, such a pattern could be used to infer information about the occupancy status of a building. Our correlation analysis discovered a number of sets, of both sizes two and three, of correlated connections that occur during the Thermostat's transition from *Home* to *Auto Away*. The connections are identified by the destination IP address 54.204.245.223 and sizes of 1375, 1391, and 2911 bytes. The correlated connection sets were comprised of permutations of these three connections. However, it should be noted that the set of three correlated connections had the best accuracy rate.

  The transition in the opposite direction is identified by connections of to the same destination IP address. However, in this case, the connections have different sizes, specifically 1663, 1631, 1711, 1786, and 1819 bytes. In contrast to the transition from *Home* to *Auto Away*, in this case the single connections which occur at the time of the mode transition represent the mode transition the best. In other words, there are no sets of two or three correlated connections which occur during this mode transition.

  For the transition between *Home* and *Auto Away* modes, our analysis resulted in 88% precision and 82% recall (23 true positive, 3 false positives, 5 false negatives). For the transition in the opposite direction, from *Auto Away* to *Home*, the 86% precision and 46% recall (13 true positives, 2 false positives, 15 false negatives) . Further manual analysis showed that in cases where there are multiple transitions throughout a sin-

gle day, transitions after the second or third instance sometimes do not produce the identified pattern.

- **Pathlight Activation** Another event of interest was that of the Nest Protect Pathlight activation (i.e., light turning on). Using the correlation analysis, we were able to identify a set of SSL connections of certain sizes (with respect to number of payload bytes sent by the device to a particular IP address) which are observed together only when the device senses motion and the Pathlight activates.

  Validation testing shows 50% accuracy (100% sensitivity). The sensitivity is a measure of the true positive rate. In other words, it expresses the proportion of positives that are correctly identified. This implies that the 50% accuracy rate is due to only False Positives (i.e., no False Negatives). Manual investigation shows that all of the False Positives occur due to the fact that the same unique connections repeat exactly 30 minutes after the initial occurrence. We are unable to explain why this occurs.

- **Smoke Alarm** Our correlation analysis shows that two SSL connections of sizes 805 and 662 (with respect to payload bytes sent by the device) occur together only when the device detects smoke and triggers the smoke alarm. Manual verification shows that there were no False Positives or False Negatives in our pattern recognition.

  When checking the validity of the network pattern associated with the smoke alarm triggering, our tests show 100% accuracy. The analyzer correctly identifies all 5 instances of the smoke alarm being triggered, with zero False Negatives and zero False Positives.

- **NTP Requests** When looking at characteristics of the network traffic, we were surprised to observe such a high number of NTP packets. Unfortunately, the correlation analysis wasn't able to identify any relationship between NTP requests and the mode of the Thermostat. However, manual investigation revealed that there is a discrepancy in the frequency of NTP requests generated between when the Nest Thermostat is operating in *Home* mode and when it is in *Auto Away* mode. Figure 5.2 shows the occurrences of NTP requests during two days.

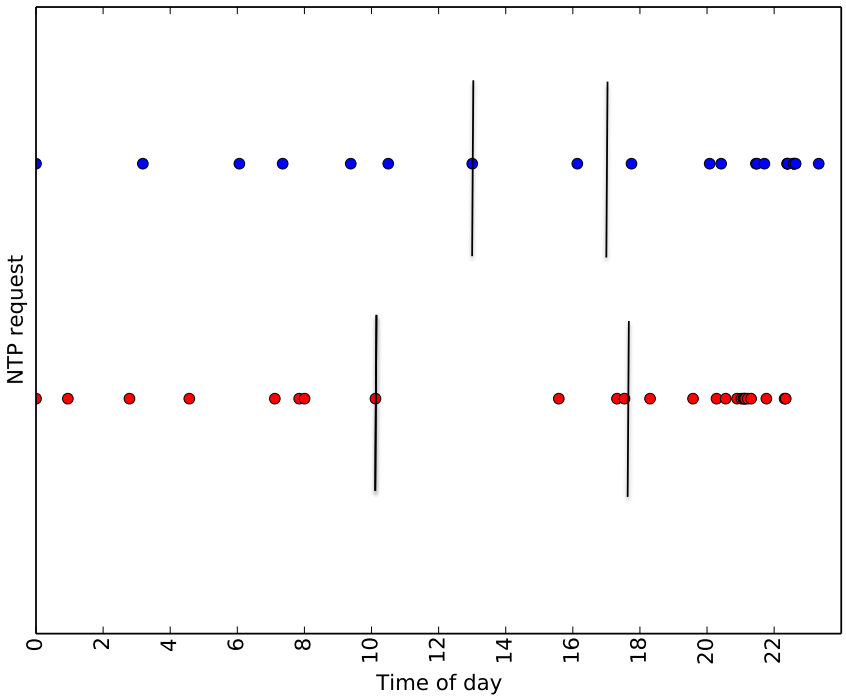


Figure 5.2: Diagram showing NTP requests made by the Nest Thermostat over the span of two days (*Auto Away* periods are marked by vertical bars)

As mentioned earlier, one of the features of the Nest Thermostat is the ability to learn a user's schedule for energy optimization. Our hypothesis is that the device updates the Nest servers with motion (or lack of) activity, which includes a time stamp. To guarantee the accuracy of the time stamp, the Nest Thermostat will use NTP to synchronize its clock. For example, the thermostat will report to the server that the user is home or that the user is not home in comparison with activity from the previous day.

To test our hypothesis, we use a simple Support Vector Machine (SVM) approach. Specifically, connection logs are split into non-overlapping one hour periods. To build the feature vector, for each period, we compute the number of NTP requests during that period. Periods between hours of 12 AM and 6 AM are ignored due to lack of user activity and random distribution of NTP requests. Each hour period used for the learning process is also labeled as 0 or 1 (i.e., 0 means *Home* whereas 1 represented the device being in *Auto Away* mode).

Testing of the SVM model shows 81% accuracy with some false positives (i.e., device was identified to be in *Auto Away* mode when it was not). To improve accuracy one could build a model where confidence in the classification adjusts over time according to the observations made. It should be noted that there was no strict schedule in the user's times of arrival and departure. The start and end times of the *Auto Away* modes varied.

5.5 Frequency Analysis

Our approach leverages the size of connections (in terms of bytes sent). Time domain features are fairly fragile. For example, our approach can be defeated trivially by padding all connections to the same size. However, our observations, particularly those depicted in Figures 5.1 and 5.2, suggest frequency analysis may also disclose information regarding the devices' state. Specifically, we wish to determine how the frequency of connections varies over time, and whether there is a change in the frequency during the occurrence of an event. Unlike time domain features, the frequency of connections cannot easily be adjusted. To avoid degradations in quality of service, many connections need to be established when a particular event and may not be buffered (e.g., notification of fire alarm or door unlocking event should not be delayed).

To enable such frequency analysis, we aggregate connections into 100 millisecond windows. This is done to replicate a regular data sampling approach. We represent each window as a single "connection" of size equal to the sum of all connections which occurred within that time window. It should be noted the destination address is ignored.

We begin by performing frequency analysis on 40-second windows, centered at the time of a mode-switch event. We compare the results with the frequency spectrum of windows immediately before and after the event window. Figure 5.3 shows the frequencies of two windows, one for the *Home* to *Auto Away* transition and one for the transition in the opposite direction. It is important to note the lack of activity in the windows before the event and after. This observation held for most transitions, especially for the windows before the transition from *Auto Away* to *Home* mode. In our experiments, we also noticed that the frequencies during the event transition windows varied, even among windows representing

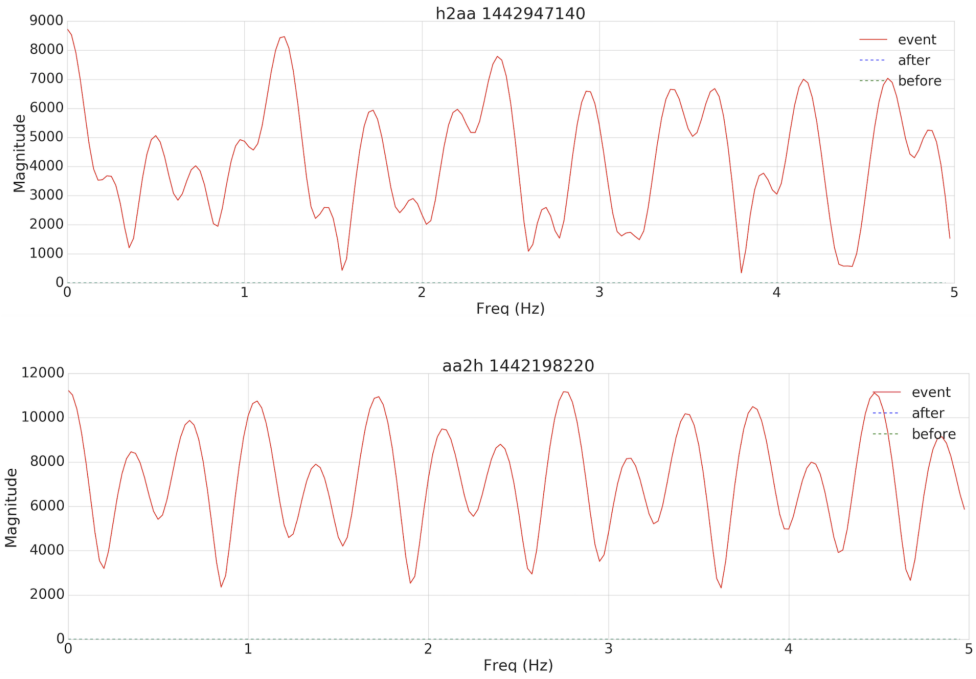


Figure 5.3: Diagram showing frequencies of connection time series during mode transitions. The windows before and after the events contain no relevant packets, hence no frequency components.

the same mode transition.

Additionally, we generate spectrograms of the Nest Thermostat connection time series for multiple days. Spectrograms are a visual representation of how the frequency spectrum varies over time. Our spectrogram uses a grey scale to show changes in frequency. In Figure 5.4, we see both the raw connection time series (top) and corresponding spectrogram for a single day. The beginning and end of *Auto Away* periods are marked by green dashed lines (at approximately 63000 and 83000 seconds) and red dashed and dotted lines (at approximately 3400 and 78000 seconds). The spectrogram shows changes in the frequency spectrum during the mode transitions, as well as at other times. Specifically, directly below the vertical lines representing the mode transition events (in the top plot), we see slightly darker vertical lines in the spectrogram, signifying change in frequency. Despite the identification of changes in the frequency of network transmissions we were unable to identify particular frequencies which describe an event. Even for the state transitions, the frequencies observed during the transitions varied. Due to limited ground truth information about the device's action and states, we cannot explain some of the other observations.

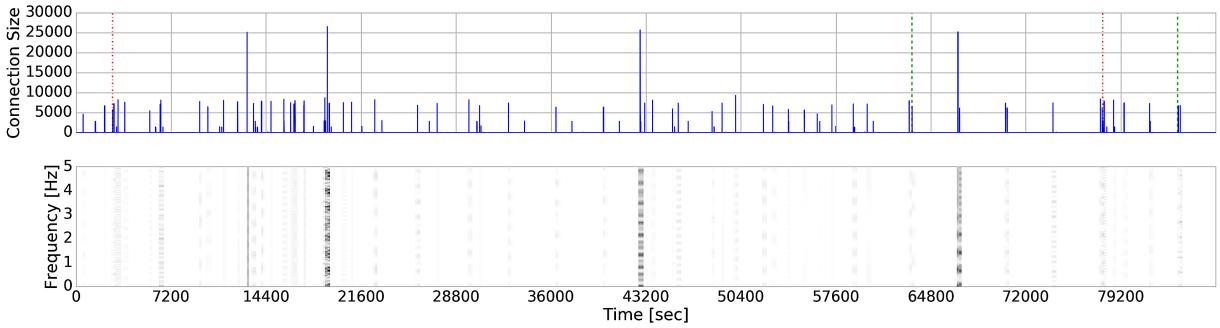


Figure 5.4: Spectrogram of connections over a single day.

5.6 Conclusion

Our approach has several limitations. First, we rely on a single side channel: encrypted network traffic. Aside from the evaluation performed, it is difficult to identify exactly what the devices are doing. In other words, without the decrypted traffic, it is impossible to determine whether the connections made during various events that we identify as "unique" are truly representative of the event occurring. Decrypted traffic would also allow us to better explain the few false positives observed as well as some of the other interesting behavior. As described above, in our analysis we do not rely on sophisticated machine learning although we believe it would be interesting to explore what patterns machine learning techniques can reveal.

Finally, our approach demonstrates how key events in the behavior of the Internet of Things devices could be identified using patterns in the network traffic. Our study shows that time domain features such as the size of connections can provide insight into the behavior of the devices. Additionally, we show that frequency analysis can also provide additional useful information.

Initial results of the work presented here were published and received the *Best Paper Award* in the proceedings of the 2016 Mobile Security Technologies (MoST) workshop of the IEEE Symposium on Security and Privacy.

Chapter 6

Side Channel Theory

In this dissertation, through our empirical studies, we aim to demonstrate that side channel information can often be used to monitor the behavior of various systems for security purposes. We argue that there are many advantages to this approach. Side channels are readily available and monitoring them has minimal, if any, overhead on the performance of the system. Side channels are also agnostic of the hardware, software stack, and technology used by the system. For example, all electrical devices consume power. Perhaps most importantly, monitoring devices via side channel analysis is non-intrusive, and in many settings (e.g., Internet of Things and other heterogeneous platforms) it is the most feasible option.

Yet, such side channel analysis is not always successful. Despite advanced data analysis methods, side channels fail to provide useful information regarding the system's behavior. However, we believe the issue is often not with the analysis process but rather lies within the side channel information. The challenge is that side channel information does not simply expose the contents of registers or the operations performed by the system. Side channel information is (usually) an indirect, obscured representation of the operations performed by the system. The ability to model the system's behavior may be heavily dependent on how much of the behavior is defined by the operations performed in comparison with the input provided. Additionally, side channels are often noisy. To this extent, it is crucial to consider the relationship between the operations performed by the system and the side channel information. Particularly, it is important to note that *information is lost during the translation from system activity (i.e., operations) to side channel.*

The work presented in this chapter is motivated by this observation as well as the lack of metrics for describing the relationship between system activity and side channel information. Specifically, in this chapter, we study *how* and *why* information is lost during the translation from sequence of operations to the sequence of theoretical side channel values. This allows researchers to understand the uncertainty associated with identifying system state transitions (i.e., sequences of operations), and consequently, states in the side channel.

In this chapter, we lay the foundation for an automata theory based model which aims to describe the relationship between system activity and side channel information. In summary, our model provides assistance with questions such as:

- In theory, how well does side channel information reflect the behavior of a program?
- What is the relationship between the side channel information and the execution of a program (and what variables does it depend on)?

Researchers can use our model as an initial step of a side channel analysis project, to estimate the theoretical relationship between system activity and the side channel information. For example, our model can be leveraged to solve the following problems:

- estimate the theoretical loss of information regarding system activity in the side channel
- measure and compare the theoretical effectiveness of side channel defense approaches

6.1 Definition

Before we dissect the relationship between side channel information and the activity of a system, we must define "side channel information". Specifically, we must establish how side channel information relates to the activity of a system and how it is generated. We define *side channel information* as information leaked by the system during the execution a program on a physical system. There are two types of side channels.

1. **on-system side channels**: information that is collected from a device, through either physical or remote access to the system
2. **off-system side channels**: information that is collected by physically monitoring the system without interacting with it

Although side channels do not exhibit themselves until a program is executed on a physical implementation of a system, we can build an abstract model representing a system, the execution of a program, and the generation of side channel information. An abstract model also has the benefit that it does not make any assumptions about the type of system, the program being executed, or the nature of the side channel. Our model builds upon existing automata theory models and defines the execution of a program P with input x (denoted P_x) on a physical implementation of a system as the following finite state machine:

Definition 6.1.1. A state machine M consists of the following sets and functions:

- set S: "states" with distinguished states $start, end$ representing the initial and ending states
- set Σ: "input alphabet"
- set Γ: "side channel alphabet"
- state transition function $\delta : S \to S \times \Gamma$

The model above achieves two goals. First, it defines the execution of P_x on system M as a sequence of states separated by state transitions. Second, the model defines side channel information as a sequence of values generated when the system exhibits a state transition. In other words, when the system changes states, whether to a new state or the current state, a side channel value is generated. It should be noted that if the state transition results in the current state being repeated, the same side channel value will again be generated. Our model defines a state transition as one (or many) operations to be performed by the system.

Using this model, the execution P_x produces two *traces* of information. One trace is a sequence of states exhibited during the execution. The second trace represents the side channel information.

Definition 6.1.2. The execution of program P_x on machine M, as defined above, produces the following two finite traces:

- *state trace* T is a finite sequence of alternating states and transition labels ending with the *end* state $[start, t_0, s_0, ..., s_{n-1}, t_n, s_n, end]$, where $s_i \in S$ and t_i represents the

transition from s_{i-1} to s_i. For every triple s_{j-1}, t_j, s_j of the form (state, transition, state) appearing in the sequence, there must be a transition in M from state s_{j-1} to s_j labeled t_j

- *side channel trace* V is a finite sequence $[v_1, ..., v_n]$, where $v_i \in \Gamma$. Each value in this sequence represents the side channel value produced as a result of the operation(s) performed during the corresponding state.

It is important to note that there is a one-to-one ratio between the number of state transitions $t_i \in T$ and the number of side channel values $v_i \in V$. In the rest of the work, we ignore the state labels of the *state trace* and focus solely on state transitions. In our model, states are abstract concepts to support the finite state machine representation. States do not represent system operations and as such do not impact side channels. Additionally, our model assumes the execution of the program halts.

6.2 Model Definition

The model above defines how side channel information is generated and its relationship to the execution of a program but fails to describe the details of this relationship. Specifically, we wish to answer the following question:

> Given a sequence of observed side channel values V and a priori knowledge of the operations encompassed by P_x, T_{P_x}, what is the *theoretical* probability that the side channel information represents the sequence of operations exhibited by program P_x?

Or more formally:

$$\text{What is } \Pr[T_{P_x}|V] \text{ ?}$$

To answer this question, it is essential to consider three characteristics:

- *operation-to-side-channel conversion*
- *side channel resolution*

- *noise*

Generally speaking, side channel information is generated by some component(s) (hardware or software) as a consequence of the work performed by the system during the execution of the program. More accurately, each side channel value is a function of the operations performed by the components during a given state transition. An *operation* is the system-level unit of work whose processing causes the generation of the side channel information. The meaning of *operation* depends on the side channel and the system. If the target side channel is power consumption of a processor, an operation is equivalent to a CPU/GPU instruction or memory/storage operation. When considering encrypted network traffic as a side channel, an operation could be the transmission (or lack of) of a single packet. Our model does not make assumptions about the meaning of operations except that there exists a relationship between the operation and the side channel considered.

The *operation-to-side-channel conversion* property considers the correspondence between the set of states transition functions t_i and the side channel alphabet Γ. In other words, it aims to determine whether or not each unique state transition is represented by a distinct side channel value.

It should be noted that our model aims to measure the correspondence between side channel values and state transitions. A state transition is not strictly limited to a single system-level operation. This depends on the granularity used to define states and state transitions. A user interested in modeling the behavior of a program's execution may define states in several ways. At the highest granularity level, any program could be defined by two states: *RUNNING* and *NOT_RUNNING*. In most cases, such a definition has little value. Alternative options include defining a state relative to functions and consequently defining state transitions the operations encompassed by those functions. The level of abstraction of the state definition determines how many system-level operations are included in a single state transition. As such, the higher abstraction, the more system operations (e.g., instructions) are encompassed in a single state transition.

This property alone has no impact our model if it is possible to capture every side channel value for every operation performed by the system. However, due to factors like the high operating frequency of the system or the side channel recording method, it may not be

possible to collect high resolution information. The operating frequency of a system refers to the rate at which the system performs basic units of work. An example of the operating frequency is the frequency at which a processor (such as a CPU) is running, often stated in gigahertz. Per our definition of the state machine M, the ratio of the number of side channel values to the number of state transitions exhibited during the execution of P_x is one. To account for this, we define the *side channel resolution* property (Section 6.3.2) which allows for flexibility in the frequency of the side channel collection method and the definition of state transitions (with respect to number of operations).

Furthermore, side channels are often "noisy". In other words, the side channel information reflects the work performed by many components of the system, not only the target component. For example, the power consumption of a processor is defined by the load imposed by the target program as well as other concurrent tasks. The noise also impacts how well a program's behavior is reflected in a given side channel.

6.3 Side Channel vs. Program

We begin by discussing each property separately. Once explained, we describe how the properties can be combined and used to estimate the relationship between system activity and side channel information.

6.3.1 Operation-To-Side-Channel Conversion

The *operation-side-channel conversion* property defines the relationship between individual, distinct operations performed by the system and their footprint on the side channel. As such, for this section, we assume that each state transition t_i is associated with a single system operation and we use the terms "state transition" and "operation" inter-changeably. The *side channel resolution* property, discussed in the next section, eliminates this assumption.

We now consider the possible scenarios for the correspondence between the side channel values and each unique state transition. In the ideal case, the values of the side channel and the possible set of system operations is represented by an *isomorphism* (i.e., bijective morphism). An *isomorphism* is a map that preserves distinctness and relations between elements of a set: each element of its co-domain is mapped to at most one element of the domain. As shown in Figure 6.1, given a stream of observed side channel values, it is possible

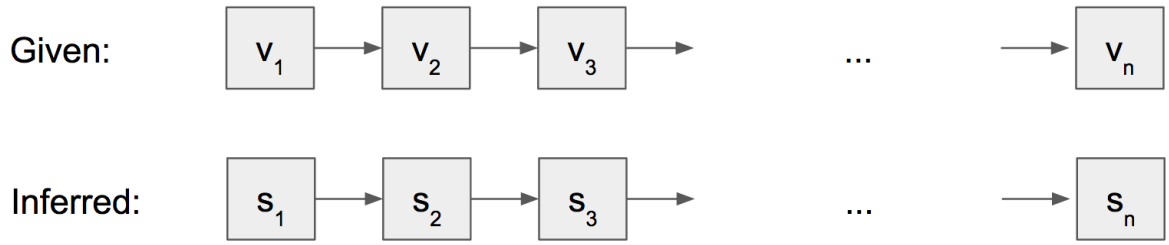


Figure 6.1: Diagram visualizing the case of n distinct side channel values and n distinct state transitions.

to infer exactly the sequence of operations (consequently, states) exhibited by the system. As such, the side channel models the behavior of the system and indirectly the execution of program P_x perfectly.

We now consider the cases where the ratio between the side channel values and the system operations is represented by a one-to-many function. Under this scenario, given a sequence of side channel values, the precise sequence of state transitions exhibited cannot be inferred. This is depicted in Figure 6.2. Instead, for each side channel value, there is some uncertainty around the precise state transition exhibited.

Based solely on the side channel trace observed during the execution, the state transitions exhibited by the system can be modeled as a probabilistic state machine. In other words, given the uncertainty regarding the specific operation that the observed side channel represents at any given time, we model the execution P_x using a probabilistic state machine (as opposed to the finite state machine alternative used in the one-to-one correspondence scenario).

A *probabilistic state automata* is a tuple $A = \langle Q_A, \Sigma, \delta_A, I_A, F_A, P_A \rangle$ where:

- Q_A is a finite set of states
- Σ is the alphabet
- $\delta \subseteq Q_A \times \Sigma \times Q_A$ is a set of transitions
- $I_A : Q_A \to \mathbb{R}$ (initial-state probabilities)
- $P_A : \delta_A \to \mathbb{R}$ (transition-state probabilities)

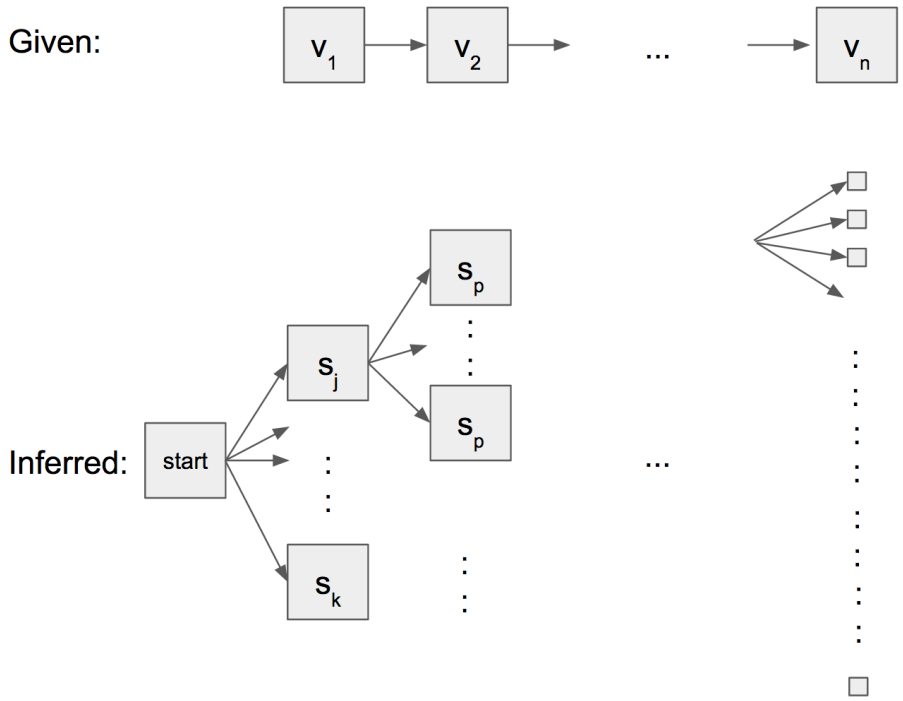


Figure 6.2: Diagram depicting the case where each unique side channel value maps to multiple distinct state transitions.

- $F_A : Q_A \to \mathbb{R}$ (final-state probabilities)

$$I_A, P_A \text{ and } F_A \text{ are functions such that } \sum_{q \in Q_A} I_A(q) = 1, \text{ and}$$
$$\forall q \in Q_A, F_A(q) + \sum \alpha \in \Sigma, q' \in Q_A P_A(q, a, q') = 1$$

Using our model we can now address questions regarding the relationship between the side channel information and the behavior of the program, specifically:

> Given priori knowledge of T_{P_x} (i.e., the sequence of state transitions of P_x), and a sequence of observed side channels V, what is the probability $\Pr(S_{P_x}|V)$?

The answer to this question lies within the structure of the probabilistic state machine. There are two ways of approaching this problem.

One method iterates through the sequence of observed side channel values V and represents the state transitions of the system during the execution of P_x as a tree structure (as shown in Figure 6.2). Each possible state permutation is represented by a path from the root of the tree (i.e., *start*) to a leaf. Out of all possible state permutations depicted by this

tree structure, only one represents the target program. This assumes that the execution of the target program is deterministic and may neglect optimization techniques such as branch prediction. The probability that the sequence of observed side channel information V represents program P_x is proportional to the number of paths from root to leaf (consequently, number of leaves). If there are n observed side channel values (i.e., n state transitions) and each observed side channel value v represents exactly m possible state transitions, the probability is equal to $1/(m^n)$ (i.e., the number of leaves in the tree). This assumes that every distinct side channel values maps to the *same* number of unique states transitions. This assumption may not always hold. It may be possible that the mapping from distinct side channel values to unique state transitions varies from one side channel value to another. In this case, we can provide a lower and upper bound of probability based on the average of the smallest and largest number of state transitions per side channel values (i.e., branch factor of the nodes in the tree structure). This estimation assumes no dependency between any two state transitions.

A better method is to use the probabilities of the probabilistic state machine defined above and calculate the result. This method accounts for dependencies between any two state transitions and achieves a more accurate result.

Knowing the trace T_{P_x}, priori knowledge about dependencies between state transitions, and a sequence of observed side channel values V, we can calculate the probability that $\Pr[T_{P_x}|V]$ as such:

$$\Pr[T_{P_x}|V] = \Pr[s_0 \rightarrow q_1] \Pr[q_1 \rightarrow q_2] ... \Pr[q_{k-1} \rightarrow q_k]$$

There is an advantage to representing the above problem using probabilistic state machines (PSM). Particularly, by allowing the user to define the probabilities of the edges, it is possible to control the number of total possibilities considered. Every edge in the PSM represents a state transition. Simply put, this representation allows the user to leverage knowledge about dependencies between state transitions. For example, if each state transition encompasses an x86 instruction, it is well known that two disable interrupts (i.e., *sti*) can not occur sequentially. This restriction can be reflected in the PSM by adjusting the probabilities of the state transitions accordingly (i.e., setting the probability to 0).

Additionally, when the probabilistic state machine lists all possible permutations of state transitions of length n, all programs of n operations are encompassed. In some cases, this may not be necessary. In controlled, real-world environments, the number of unique programs operating on a given system is limited. Consider for example, a high-performance computing environment where side channel analysis is leveraged to monitor the activity of a compute node. Despite the large number of users, only a few hundred unique codes (i.e., programs) are executed. It is reasonable that the user wishes to find the probability that the observed side channel values represents one of the many known codes (as opposed to all possible programs). In such a case, it is possible to adjust the probabilities of the PSM based on *a priori* knowledge about the set of codes executed on the system.

6.3.2 Side Channel Resolution

Until now, our model assumes that every observed side channel value v_i (as defined in Definition 6.1.2) reflects a single state transition. We also restrict a state transition to represent a single system operation. This assumption may not always hold. As previously mentioned, it may not always be possible to collect side channel information at the same rate as the operating frequency (e.g., processor clock rate) of the system. In such cases, a single state transition encompasses several system operations. This affects our model in the following way: every state transition observed side channel value may represent more than one system operation.

Our model can be adjusted to reflect this observation. We begin by distinguishing between per-operation side channel values and observed side channel values. Per-operation side channel values represent changes in the side channel due to the execution of a single operation. An observed side channel value is the sum of one or more per-operation side channel values. Formally:

Definition 6.3.1. The observed side channel value v_t at time t is a sum of the individual per-operation side channel values o_i in the interval $[t-1, t)$:

$$v_t = \sum_{i=0}^{m} o_i, m \geq 1$$

Given this, the task or even operations performed by the system cannot be directly inferred from a sequence of observed side channel values. Instead, we must first determine the number of permutations (with repetition) of m operations the observed side channel value can represent. We assume the number of operations which contributed to the observed side channel values is known (or can be estimated). Specifically, we are interested in the number of m-sized per-operation side channel values configurations which sum to the observed side channel value v_t.

This problem is similar to the counting coin problem, where given coin denominations, we wish to find the number of coin combinations that generate the target sum. However, there are two differences. The counting coin problem determines the number of combinations, whereas our problem requires the number of permutations (permutations of the same instructions represent different programs). Assuming the number of operations executed is known, we wish to know permutations of a fixed size m.

To solve this problem, we leverage *generating functions* composed of variables with exponents equal to the per-operation side channel values. *Generating functions* count weighted objects. In our problem, the weights represent side channel values. The variable exponents of the formal power series described by the generating function are the side channel values. The coefficient of a variable represents the number of configurations of per-operation side channel values that sum to the observed side channel value. For example, x^1 represents a side channel value of 1 and x^M represents a side channel value of M.

As an example, assume that the set of per-operation side channel values $O = 1, 5, 10$ and we are interested in the number of ways various sums of $m = 3$ operations can be obtained. The generating function becomes:

$$f(x) = (x^1 + x^5 + x^{10})^3$$

Using this generating function, the number of configurations of m operations that can sum to S is defined by the coefficient a_v of the x^v term. Continuing with the example above, the number of permutations of three operations that can sum to 25 is 3, according to the coefficient of the x^{25} term. However, only one of those permutations represents the sequence of side channel values generated by the target task. Therefore, the probability that the

observed side channel value represents the sequence of operations performed during state transition before state s_t of the target program p_x is:

$$\Pr[s_t|v] = \frac{1}{a_v}$$

6.3.3 Putting It All Together

The *side channel resolution* property defines a method for counting the number of ways distinct per-operation side channel values can be combined to reach a target observed value. As described by the *operation-to-side channel* property, a distinct side channel value can represent multiple different operations. Therefore, the probabilities computed by these two properties must be combined to determine the theoretical probability that a sequence of observed side channel values represents the execution of a particular program. More formally, the probability is defined as:

$$\Pr[p_x|v] = \prod_{i=1}^{|t|}\left(\Pr[s_i|v] * \prod_{j=1}^{|s_i|}\Pr[action_j|o_j]\right)$$

where $|s_i|$ represents the number of operations per state transition/observed side channel value, $action_j$ represents the j-th action/operation of state transition prior to state s_i and o_j represents the per-operation side channel value.

6.4 Information Loss

The model above provides an upper limit on the probability that a given sequence of observed side channel represents a particular execution of a program. Another benefit of the model, is that it can help determine the *theoretical* loss of information for a side channel measured while running a program. Shannon entropy is a natural tool for calculating this. Shannon entropy is an information theoretic method for measuring the amount of information associated with a random variable and is defined by the following formula:

$$H(X) = -\sum_{i=0}^{n} p(x_i)\log_b p(x_i)$$

Shannon entropy can be leveraged to represent the amount of information lost in a side channel. As our model indicates, a program is represented by a sequence of operations

(e.g., instructions) and its entropy H_{p_x} can be trivially computed. We can also generate the expected side channel values by converting each operation to its appropriate side channel value. Given this sequence of expected side channel values, we can easily compute the entropy of the side channel H_{sc}. Consequently, the amount of information lost in the side channel is defined by the ratio of the entropy of the side channel and the entropy of the program:

$$Loss = 1 - \frac{H_{sc}}{H_{program}}$$

In practice, the results may vary due to noise and other factors.

6.5 Noise

The construction and definition of our model assumes the side channels do not contain noise. In this setting, noise is defined as the contributions to the side channel by other software or hardware components of the system running at the same time as the target program. For example, noise can be introduced by other tasks executing on the system at the same time as the target program. Noise can also be introduced by hardware components of the system that are unrelated to the execution of the target program but impact the same side channel. Considering desktop power consumption as a side channel, the readings between any two time steps is a combination of both multiple software and hardware components. From a hardware perspective, the power consumption of a standard desktop computer is affected by the CPU, memory, disk drives as well as fans and lights. However, CPU, memory, and disk utilization are driven by several concurrent processes, including operating system tasks.

It is important to note that noise may differ between time steps. In the desktop power consumption example, not all hardware components have the same impact on the power consumption. Furthermore, operating system tasks may be less CPU intensive than the targeted program.

There are two ways of accounting for noise. In the ideal case, each component contributing to the side channel is considered independently and can be described using our model. However, this ideal case may not always be feasible and increases the complexity of the model. The other alternative treats noise as a stochastic process. To reflect the latter, we expand on our previous definition of the measured side channel value:

Definition 6.5.1. The collected side channel value v_t at time t is a sum of the individual per-operation side channel values o_i pertaining to program P_x plus a random value C:

$$v_t = \sum_{i=0}^{m} o_i + C_t, m \geq 1$$

Additional research is required to study the impact of noise and develop accurate methods for accounting for noise.

6.6 Turing Machine Side Channel Example

To demonstrate the value of the above model, we describe a theoretical example. Specifically, we describe how our model can be applied to a physical implementation of a Turing machine.

A Turing machine is a mathematical model of a hypothetical computing machine that can be used to simulate any computer algorithm. A Turing machine is composed of the following main hardware components: tape head, tape motor, and an infinite tape. The set of possible actions to be performed by our Turing machine include, and are limited to, movements of the tape head (left and right), read operation, write operation, as well as unary arithmetic operations. We denote these operations as: *movr*, *movl*, *read*, *write*, *add*, *sub*, *mul*, *div*.

Implementing a physical Turning machine is impossible due to the requirement of an infinite tape. Furthermore, Turing machines are not very practical, due to their inefficient, sequential operating nature. Yet, despite the physical limitations, we can imagine that a physical implementation of a Turing machine consumes power to perform various operations.

Below, we apply our model to a hypothetical physical implementation of a Turing machine and consider power consumption as the side channel. While the power consumption values are arbitrarily selected, the differences between power consumption across various operations are based on realistic expectations. For example, even in modern hard drive technologies, read operations are less power intensive than write operations. It is also logical that moving the tape head takes the same amount of power regardless of the direction. Specifically, for our hypothetical case study, we make the following assumptions:

- Head movement actions consume equal amount of power, 5 power units
- Read operations require 2 power units

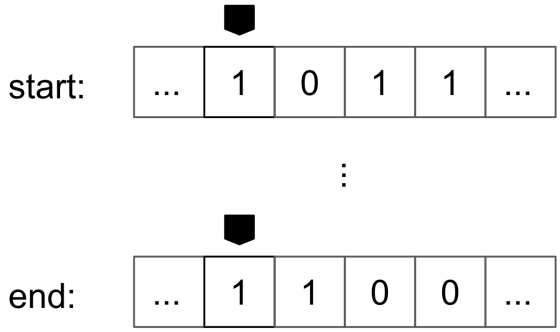


Figure 6.3: Diagram visualizing the tape head location and contents at the beginning and end of the binary program execution

- Write operations require 3 power units
- Read and write operations may only use binary symbols (i.e., 0 and 1)
- Addition and subtraction operations require 1 power units
- Multiplication and division operations require 7 power units

In our case study, we leverage our model to measure how well power describes the behavior of the Turing machine during the execution of a binary counter program. The binary counter program reads the binary number starting at the current tape location and increments it by one, overwriting the input with the incremented value. The Turing machine pseudocode for the program is listed in Appendix B.1.

Let us consider the execution of the binary counter program with decimal "11" (i.e., "0b1011") as input. Figure 6.3 visualizes the location of the tape head and the contents of the tape at the beginning and end of the execution of the binary counter program. The execution involves twelve steps during which the Turing machine will move the tape to the rightmost position, read the contents of that cell, increment the value read, and write the result. After the result is written, the tape will move left and modify the contents of the tape as needed, due to the carry. Per the definition of our model, the execution of the binary counter program would produce the following state transition trace:

$$T = \{movr, movr, movr, read, add, write, movl, read, add,$$

$$write, movl, read, add, write, movl\}$$

Given the side channel values per operation defined above, the execution of the binary counter program with decimal 11 as input will produce the following side channel trace:

$$V = \{5, 5, 5, 2, 1, 3, 5, 2, 1, 3, 5, 2, 1, 3, 5\}$$

For reasons of simplicity, we first assume we can observe one side channel value per state transition. We also assume our side channel information is collected at the same frequency as the operation frequency of the Turing machine (i.e., one operation per side channel value).

Due to the assumption made regarding the power consumption of each operation, it is trivial to see that the state transitions exhibited by the Turing machine cannot be precisely inferred. Starting with the first side channel value of 5, there is uncertainty whether it represents a left or right tape head movement. As such, there is a 50% chance the side channel value represents the first state transition of the execution of the binary counter program with input "11". In total, per our model, the probability that the observed side channel values represent the execution of the binary counter program is:

$$\Pr[T_{P_x}|V] = \Pr[movr|5] * \Pr[movr|5] * \Pr[movr|5] * \Pr[read|2] * \Pr[add|1] * ... * \Pr[movl|5]$$

$$\Pr[T_{P_x}|V] = 0.5 * 0.5 * 0.5 * 1 * 0.5 * ... * 0.5$$

$$\Pr[T_{P_x}|V] \approx 0.002 = 0.2\%$$

This value represents the probability that the side channel trace observed V represents the execution of the binary counter program *out of all of the possible programs that produce the same trace.*

In the case study above, we assume we can collect power information at the same rate as the Turing machine's operating frequency. Depending on the speed of the Turing machine,

this may not be feasible. Considering the length (in terms of operations) of the execution of the program, let us consider the case where each side channel measurement contains $m = 3$ operations. The observed side channel sequence now becomes:

$$v = \{15, 6, 8, 10, 9\}$$

To calculate the probability that the observed side channel trace represents the execution of the counter program with input (decimal) "11", we apply the *side channel resolution* property and adjust the computation accordingly. First, we write the generating function that will help us compute the sums :

$$g(x) = (x^1 + x^2 + x^3 + x^5 + x^7)^3$$

Expanding this function results in:

$$g(x) = x^{21} + 3x^{19} + 6x^{17} + 3x^{16} + 10x^{15} + 6x^{14} + 12x^{13} + 9x^{12} +$$

$$+15x^{11} + 12x^{10} + 13x^9 + 9x^8 + 9x^7 + 7x^6 + 6x^5 + 3x^4 + x^3$$

As described previously, the exponent represents the observed side channel value and the coefficient represents the number of configurations of per-operation side channel values that sum to that observed value. For example, given the first observed side channel value of 15, the coefficient of x^{15} is 10. In other words, there are ten ways to compute the value 15 using the sets of pre-defined per-operation side channel values.

To compute the probability that observed trace V represents the execution of the binary counter program, we must combine the probabilities:

$$\Pr[s_1|v_1] * ... * \Pr[s_5|v_5] =$$

$$= \Big(\Pr[5,5,5|15] * \Pr[movr|5] * \Pr[movr|5] * \Pr[movr|5]\Big) * ...$$

$$*\Big(\Pr[1,3,5|5] * \Pr[add|1] * \Pr[write|3] * \Pr[movl|5]\Big) =$$

$$= 1.98 * 10^{-8}$$

The computed probability is low but it is important to note that our computation considers *all* possible programs (with the same number of operations) that would have produced

the same side channel trace. In certain settings, where the set of programs executed by the systems are known, such broad computation may not be of interest. For example, if we know (or expect) that only the binary counter and an unary subtraction programs are executed on our Turing machine, we can improve our calculation. Specifically, since between the two programs, only the execution of the binary counter program with input "11" can generate the observed side channel sequence V, we can be 100% certain that V represents the execution of our target program.

Using our model, it is also possible to estimate the amount of information lost in the translation from the system operations to side channel. In the example of the binary counter program, given the sequence of state transitions and respective side channel values, we compute the following entropies:

$$T = \{movr, movr, movr, read, add, write, movl, read, add,$$

$$write, movl, read, add, write, movl\}$$

for which,

$$H_p = 2.32193$$

and

$$V = \{5, 5, 5, 2, 1, 3, 5, 2, 1, 3, 5, 2, 1, 3, 5\}$$

with

$$H_{sc} = 1.92193$$

Consequently, the theoretical amount of information loss becomes:

$$Loss = 1 - \frac{H_{sc}}{H_{program}} = 1 - \frac{1.92193}{2.32193} = 0.1723$$

6.7 Limitations and Discussions

6.7.1 Limitations

In our attempt to build a model that treats a sufficiently wide variety of systems and side channels, we make several assumptions. These assumptions may impact the results of applying the model in certain settings. It is important to note that due to the definition and construction of our model, our ability to model the system's behavior heavily depends on

how much of the system's behavior is defined by the input data and how much is defined by the operations. In other words, two identical sequences of operations may represent different program executions, depending on their input (e.g., addition program; same instructions but different executions when inputs are different).

We suspect that our model could be applied to the IoT/Nest devices traffic analysis work if more information was available. Specifically, our model relies on a sequence of system operations and an equal-length sequence of side channel values both of which are assumed to be known. When performing network traffic analysis, the sequence of packet transmissions represents the sequence of system operations. An option for the side channel could be the size of the packets. However, despite having network traces, it is difficult to distinguish between two packets/connections of the same size. It is important to note that just because two packets or connections have the same total payload size, it does not mean their contents are equal. Having access to decrypted network traffic would eliminate this problem.

Unfortunately we are also unable to apply our model to the Darshan I/O study due to the lack of time series information. Darshan provides aggregate information about I/O activity and our model requires time series (i.e., sequences) data of operations performed as well as side channel measurements.

Circumstantial lack of data hindered our ability to apply our model to the previously mentioned experiments. On the other hand, while the HPC power analysis work meets all of the pre-requisites of our model, there are several complications involved with applying the model. First, it is important to note that in our HPC power analysis work, we are measuring the power consumption of the power distribution unit feeding the HPC rack. This has several implications. On one hand, it means that we are measuring the power consumption of all of the hardware components across all nodes of that HPC rack. This extends beyond CPU power consumption to memory, as well as network chips and other hardware components. Moreover, by measuring the power consumption at that location, the power consumption is also affected by the power distribution unit and the power supplies. For any meaningful results, our model would need to be applied to all of the components contributing to the power consumption. In addition, the relationship between CPU's execution of instructions and power consumption would need to be studied at a physical level. We also would like

to note that determining the power footprint of each individual application is non-trivial. Researchers have attempted to characterize the energy per instructions in a few and rare cases [71] yet, aside from the fact that they are dealing with energy (not power), such attempts are only estimates.

In addition to those concerns, there are several software and hardware optimizations implemented that affect the application of our model to the HPC power analysis experiment. Our model assumes that the execution of a program with a given input P_x is deterministic. In modern computers, this is not true in the sense that two execution of the same program with identical inputs do not necessarily result in the same sequence of instructions being executed. This is due to a number of optimizations including branch prediction and cache status. It is also important to note that the number of instructions retired is not the same as the number of instructions executed. In other words, in some cases the computer executes more instructions than needed for the current execution (due to branch prediction). These additional instructions, while not representative of the program's execution, also contribute to the power consumption observed and further affect the relationship between power consumption and the program's behavior.

Hardware optimizations such as dynamic frequency scaling further complicate matters. Dynamic frequency scaling refers to the dynamic ("on the fly") adjustment of the microprocessor's operating frequency based on the workload. This optimization is designed to reduce power consumption and decrease the temperature of the microprocessor during periods of low computation. Combined with the large variety in the latency of microprocessor instructions, in the case of power analysis, it is impossible to determine exactly how many instructions contributed to a single μPMU measurement (reminder: the μPMU reports data at 120 Hertz, compared to GHz frequencies of modern CPUs).

As previously stated, in most cases the model is only able to compute a theoretical estimate of the information loss. However, due to the above mentioned challenges, even if the per-instruction power consumption is known and the number of instructions per power measurement is estimated, the computed information loss would be considerably inaccurate.

We wish to highlight that our model makes assumptions about the number of operations which contributed to the generation of a single side channel value. This assumption is

reflected in the *side channel resolution* property. The generating functions define use N to constraint the number of operations that have contributed to the generation of a side channel value. However, we believe this assumption can be eliminated at the expense of accuracy and the generating functions can be adjusted. If we wish to only impose an upper limit on the number of operations that contribute to a given side channel value, we can modify our approach such that each per-operation side channel value is defined by a generating function

$$f_o(x) = \sum_{i=0}^{m} x^{oi} = \frac{1 - x^{o(m+1)}}{1 - x^o}$$

Consequently, the generating function for selecting per-operation values $o \in O$ that total to the observed side channel value v is:

$$g(x) = f_{o_1}(x) \cdot f_{o_2}(x) \cdot ... \cdot f_{o_n}(x) = \frac{1 - x^{o_1(m+1)}}{1 - x^{o_1}} \cdot \frac{1 - x^{o_2(m+1)}}{1 - x^{o_2}} \cdot ... \cdot \frac{1 - x^{o_n(m+1)}}{1 - x^{o_n}}$$

Expanding these generating functions, the number of ways in which per-operations values $o \in O$ can sum to the observed side channel value v is equal to the coefficient a_v of x^v. However, this approach will not provide all of the permutations. Alternatively, it is possible to adjust a dynamic programming-based approach to count the number of solutions.

It is important to note that our model does not account for all possible optimizations. Considering these limitations, when applying our model to a real scenario, it is important to note that the probability computed by our model is a theoretical upper bound. In other words, optimizations and various noise levels may further decrease the probability that a given sequence of observed side channel values represents a particular sequence of system operations.

We believe the model presented in this dissertation can be expanded. First, we believe it is important to study and address problems regarding the relationship between state transitions and states. In other words, it may be possible that two state transitions for two sets of different states represent the same sequence of operations. In such cases, despite being able to infer the state transition from the side channel information, there is still uncertainty regarding the state exhibited by the system. Furthermore, we believe additional research is required in order to study the limitations described above.

6.7.2 Comparison with Entropy-Based Models

Intuitively, Shannon's entropy is a natural way to measure the loss of information for a side-channel measured while running a program. It could be argued that Shannon's entropy could be used directly by defining the measure the amount of information lost in the side channel as a ratio of the entropy of the measured side channel sample and the entropy of the program:

$$\frac{H_{sc}}{H_{program}}$$

As in our model, the entropy of the program is computed by representing the program as a sequence of operations. On the other hand, the entropy of the side channel H_{sc} can be computed using real-life side channel samples.

The advantage of this approach is that is uses real observations and as such, it naturally accounts for noise. Using repeated experiments, the average expected information loss can be trivially computed.

However, there are some concerns with this approach. Shannon's definition of entropy, $H(X) = -\sum_{i=0}^{n} p(x_i) \log_b p(x_i)$, applies to a discrete random variable X with possible values $x_1, x_2, ..., x_n$ and probability mass function $P(X)$. The discrete nature of the random variable is important for this definition and despite generalizations of the definition for continuous random variables, there are concerns associated with using such definitions for measuring side channel entropy. Perhaps most important, the entropy of a side channel is dependent on the measurement method, specifically the precision and sampling rate. Similar to our model, entropy-based models also suffer of the need to synchronize the beginning of the execution of a task with the start of a side channel sample. As such, the entropy is not intimately linked with the underlying computational process but rather a reflection of the technology and method used to measure the side channel.

On the other hand, our model starts by defining the relationship between individual system-level operations performed and the side channel. By doing so, our model provides researchers with the theoretical loss of information measure, agnostic of the side channel sampling rate and other attributes. We believe our model can be further extended to eliminate some of the limitations, in particular with regard to noise.

Chapter 7

Towards Protecting Against Side Channels

"Technology is, of course, a double edged sword. Fire can cook our food but also burn us."

— Jason Silva

The objective of this dissertation is two fold. First, it explores novel applications of side channel analysis. In this process, it demonstrates how side channel analysis is a viable and efficient method for non-intrusive monitoring of systems, even in heterogeneous settings. Specifically, the work contained in this dissertation shows how side channel information can be leverage for monitoring systems for security and privacy purposes. In addition, we present a theoretical model that describes the relationship between the activity of a target system and a given side channel.

Most of the dissertation presents side channel analysis as a cyber security monitoring tool. Yet, as previous efforts show, side channel analysis information can be exploited with malicious intent – it is "dual use," as it is sometimes called. Without a stretch of imagination, it is trivial to envision how approaches presented in this dissertation can be used for malicious purposes, such as spying and beyond. For example, the approach used to monitor Internet of Things devices could also be used by attackers as an eavesdropping method to determine human presence or discover other sensitive information about the users. The approaches

used to monitor activity of high-performance computing facilities could be leveraged by a nation-state to spy on the activity of foreign or domestic entities.

As cyber security researchers, it is our responsibility to consider both types of use cases of security tools. To this extent, it is important to continue exploring potential benefits and drawbacks of side channel analysis. Potentially interesting experiments include the use of side channel analysis for reverse engineering of programs and identifying trojans or attacks at runtime using side channel information.

Regardless of the application and intent, it is important to understand the relationship between side channel information and the activity of a system. Our model lays the foundation for describing this relationship and we believe additional research is needed to expand and refine our model. Empirical studies are also needed to discover novel applications of side channel analysis. For example, in this dissertation we show how the control flow of a program is exhibited in side channel information. This can have many implications. It may be valuable to determine whether a program can be reverse-engineered solely from side channel information. Furthermore, studying the application of side channel analysis for control flow integrity purposes could prove valuable.

It is especially important to consider this problem from a defensive perspective. As previous research shows, side channel analysis has grave security and privacy implications. Combined with the ubiquitous nature of side channel information, cyber security researchers are presented with important challenges.

From our experiences with side channel analysis, we believe the best side channel defense mechanism is a defense-in-depth approach, implemented at multiple levels. The side channel analysis defense problem is composed of two main problems: access to side channel information and entropy of side channel information.

Preventing the collection of side channel information is non-trivial. Side channels expose information about a system's activity, which may be considered sensitive information. As such, it is natural to consider side channel information as information that flows from a high security level to a lower level, similarly to the Bell-LaPadula (BLP) model [72]. The Bell-LaPadula model is a model that relies on state machines to describe and enforce access control in government and military applications. The model focuses on data confidentiality

and defines both subjects and objects as having security levels. In the side channel problem setting, the program and hardware executing the program maintain a high security level. However, given the availability and nature of side channel information, it may be possible that any entity, despite their security level, may observe and collect this information. In the case of on-machine side channels, the side channel information may be "readable" by any system user. For example, any user on the system may be able to collect system wide performance counter data. For off-machine side channels, the information is "readable" by any user who has physical access to the room where the computer resides (or even further). As an example, power consumption of a server may be measured by individuals who do not have digital access to the server but have physical access to the server room or the building where the server resides. In this regard, it is possible to consider side channels as a specialized subset of covert channels. Specifically, we can think of side channels as a singleton, one-way covert channel, where one party is the computing system which "writes" information to the side channel and the user who is only capable of performing "reads" of the side channel information. In our definition, a user has the following capabilities, depending on the side channel type.

1. For **on-machine side channels**, the following assumptions are made:
    - the user can access and record side channel information (either by physical or remote access to the computing system)
2. For **off-machine side channels**, the following assumptions are made:
    - the user does not have access, physical or remote, to the computing system
    - the user can collect side channel information non-intrusively while not being in contact with the target computing system

From this perspective, the problem of side channel information is both an information flow and inference problem. The inference problem occurs when sensitive information can be disclosed from non-sensitive information. While this problem has mainly been studied in database settings, the problem of side channel analysis is no different. Previous researchers

have studied various aspects of these problems. For example, Goguen and Meseguer [53] introduce a general automaton theoretic approach to modelling secure systems. In their work, they distinguish between security policies and systems and provide a simple language for defining security policies. This language is based on the concept of *non-interference* which states that the actions of one group of users are non-interfering with another group, if the actions of the first group has no effect on the second group. While, their approach does not deal with inferences "either logical or statistical, of unauthorized information from information which is authorized", we believe the concept of *non-interference* can be extended to fit to the side channel defense problem. Specifically, the leakage of information via side channels can be described in terms of *interference*, the semantically opposite of *non-interference*. Formally defined, the concept of *non-interference* in our problem setting can be defined as follows:

> A computing system is non-interfering with a group of users if the operations performed by said system has no effect on what the users can see.

Other related efforts [73, 74, 54] study the problem of information flow control. We believe these ideas can be applied to the problem of restricting access to side channel information.

However, controlling the collection of side channel information may not always be plausible. As history shows, breaches occur despite security measures. Consequently, it is important to consider the second aspect of defending against side channel attacks: side channel entropy. Specifically, it is important to tackle the problem of minimizing the entropy of side channel information by both enforcing regularity or adding noise.

Theoretical models can help describe properties of the relationship between side channel and system activity, as they relate to security and privacy. While our model provides researchers with a starting point, more research is needed to develop methods for measuring the impact of side channel attacks. Having established appropriate metrics and models, the next step is to implement and test empirically.

Researchers have studied covert channels and explored defensive mechanisms to minimize the capacity of covert channels. As mentioned earlier in this dissertation, a side channel is a one-way covert channel, where the system produces information and a user listens. Consequently, defensive approaches for combating side channel analysis can be inspired from

previous efforts in covert channel analysis and steganography. Generally speaking, defensive approaches may include both software and hardware implementations. Software defense techniques include disguising approaches which aim at making all programs indistinguishable from each other, with respect to their representation in the side channel. Some work in this domain has been done; for example, the work presented by Ambrose [75] introduces random operations to mask the side channel footprint of a target program. Their approach, while effective, introduces an overhead of almost 30% in both runtime and average energy. In the future, we would like to explore the effect of code obfuscation techniques and compiler optimizations on side channel analysis. Preliminary experiments showed that different C compiler optimizations cause distinct power signatures for the execution of the same program with identical input. Such techniques have been extensively studied in specific domains yet we believe they can be of value in this new setting. Software-level defense mechanisms must be customizable and balance the trade-off between security and overhead. In some cases, such as the Internet of Things and other embedded devices, introducing noise into the side channel may result in depletion of limited resources. Hardware defense mechanisms such as those presented by Grabher and Ambrose [76, 77] can also help protect against side channel attacks. As the authors describe, hardware-based techniques minimize the runtime overhead compared to software approaches. However, hardware approaches require hardware modifications, sometimes even software revisions, and are often less customizable.

A supplemental defense strategy could be to exploit weaknesses in the machine learning or statistical inference process usually associated with side channel analysis. In machine learning, adversarial examples are carefully crafted inputs provided to machine learning algorithms to cause the model to make a mistake. Adversarial examples are analogous to optical illusions, but for machines. Recent research has shown adversarial examples are quite successful [78]. Currently, the success of adversarial examples is dependent on the machine learning approach used. Additional research is required to investigate the robustness of such approach for defending against side channel attacks.

Chapter 8

Conclusion

8.1 Summary

In this dissertation, we present methods for leveraging side channel information to perform non-intrusive monitoring of systems. Throughout the dissertation, we present several empirical studies that showcase the advantages of using side channel analysis for monitoring systems. Specifically, we demonstrate that our method is not only non-intrusive, but can also be used to monitor heterogeneous devices (as is the case with Internet of Things platforms). We also show that relying on side channel information enables a platform and software agnostic method for monitoring systems for security and privacy reasons.

To showcase the robustness and broad range of applications of our approach, throughout our empirical studies, we cover a variety of side channels in a number of different platforms. For example, in our work (Chapter 4), we show that by analyzing the power consumption of a high performance computing platform, it is possible to identify what program is running, in the presence of limited noise. Furthermore, in Chapter 5, we show that by analyzing encrypted network traffic, it is possible to identify different states of Internet of Things devices, without any prior knowledge. Through our empirical studies, we demonstrate that side channels can contain enough information regarding system activity to serve as a non-intrusive method for monitoring the behavior of systems and can even act as an input to intrusion detection systems.

As shown by our empirical studies, side channels are often noisy information channels. Additionally, side channel information is an altered representation of particular aspects of

the system's behavior. For example, encrypted network traffic can help describe the network behavior of a system but provides no insight into the CPU and memory activity. Motivated by this observation, we lay the foundation for a theoretical model that aims to describe this relationship. The model uses automata theory to define how side channel information is generated and how it relates to the operations performed by the system. Specifically, our model considers three factors essential to such side channel analysis:

1. impact of each individual system operation on the side channel
2. impact of side channel collection rate on the accuracy of differentiating between states
3. impact of multiple components (hardware and software) simultaneously affecting the side channel (noise)

While further research is needed to improve the robustness of our model, we show how our model can be used in some cases to compute the theoretical probability that a given side channel sample represents a particular program. Additionally, our model shows how it is possible to apply entropy and compute the theoretical loss of information between a sequence of operations performed by the system and the observed side channel sample.

Our work differs from previous side channel analysis work. At a high level, previous efforts prominently treat side channel information as an attack vector. Consequently, the majority of the previous efforts exploit side channel information to break the security of systems (e.g., recover private cryptographic keys) and even infringe on the privacy of users (e.g., infer web browsing activity or printed information). In this dissertation, we aim to show that side channel information can be beneficial, even for security purposes such as system monitoring. At a deeper level, previous efforts focus on the information flow between the input to a program (or system) and the side channel information. In contrast, the work presented in this dissertation studies how control flow information is reflected in side channel information. In our experiments, we leverage the relationship between the operations performed by the system and the side channel information to enable the monitoring.

8.2 Limitations and Future Work

In this dissertation, we also present some negative results. One such instance is our attempt to use Darshan I/O characterization data to classify programs into computational dwarf classes. In addition, in our HPC power analysis experiments, we show that noise can have a significant impact on the accuracy of identifying running programs.

The reasons for failure vary from experiment to experiment. For example, the negative results of our Darshan experiment are mostly explained by the lack of relationship between I/O operations performed by a system and the underlying computation type performed by the system. In other words, the POSIX I/O operations performed by a program are not representative of the underlying algorithm implemented by the program. For a more detailed explanation, we redirect the reader to Section 4.3.5.1. As stated above, in our HPC power analysis experiments, while we can identify programs running with very high accuracy when monitoring a single node (or when only one process is running at a time), the accuracy drops when these conditions change. In such cases, noise is the primary culprit.

It is important to note that there are innate limitations to relying on side channel information for monitoring the behavior of a system, as described in Chapter 6. Side channel information is rarely a direct representation of the operations performed by the system. As such, information is often lost when translating system operations to a sequence of side channel values. In addition, side channels are often controlled by many hardware and software components. Properties of the relationships between hardware and software components impact the resolution of side channel information with respect to the control flow of a program.

In many ways, extracting useful information from side channels is similar to signal processing problems. With respect to side channel analysis, noise refers to the fact that the side channel is often a representation of the work performed by several components of the system. In our work, we do not study the potential of noise filtering techniques in this setting, although we believe it may be valuable.

Our work has revealed several questions about the benefits and drawbacks of such side channel analysis. Despite featuring a variety of side channels, in each experiment we rely solely on a single side channel for our analysis. We believe there are benefits of using multiple side channels although this often comes at the cost of intrusiveness. When considering

multiple side channels, we believe it is important to consider the amount of additional information each side channel can contribute. For example, for our experiment involving the monitoring of IoT devices, we believe power analysis would be extremely valuable and a great complement to the traffic analysis. These two complimentary side channels would provide information about both network activity as well as CPU and memory behavior of the devices. It is also important to note that performing power analysis of IoT devices and other embedded devices would be relatively simple compared to the HPC power analysis performed in our experiments (especially considering the single-application nature of such devices).

As described in Chapter 7, we believe the work presented in this dissertation merely begins to explore a new path. Our empirical studies and theoretical model are pixels in the larger picture. Continuing with the theme of the work presented in this dissertation, we believe additional research efforts are needed to explore the benefits of side channel analysis for anomaly and intrusion detection. From a theoretical point of view, we believe additional research is needed to study the limitations of our model and extend the model. From a practical standpoint, while we present such side channel analysis as an advantageous opportunity, it can also be used with malicious intents. Additional work is required to develop models that defend against such side channel analysis.

8.3 Recommendations

While side channel information can provide valuable insight into the activity of a system, processing and extracting meaningful information is non-trivial. As evident from the empirical studies described in this dissertation, the analysis can vary greatly, depending on the objective, setting and the nature of the side channel. Our model describes the complexity of the relationship between system activity and side channel information. Despite these challenges, there are measures researchers can consider to facilitate such analysis.

Our recommendations are inspired by both our experiences and our model. These recommendations apply to the selection, collection, and preprocessing stages of the analysis process. Aside from our recommendations, the success rate will vary depending on the analysis methods used. We describe our recommendations below in detail:

- **Side Channel Selection**: In most settings, researchers will have several side channels available for processing. It is important to consider how each side channel is generated and how it relates to the researcher's objective. In particular, the research should establish which system operations are most descriptive of behavior of the target program/system for their purpose and how those operations affect the different side channels. For this part of the process, it is important to consider our model and the properties described. At the same time, researchers should consider the difficulty involved with recording each side channel. Off-system side channels can be collected passively and non-intrusively, yet may contain more noise.

- **Side Channel Collection**: Once the target side channels have been selected, it is important to consider two things about the collection of side channel information. Whenever possible, it is best to get as close to the source generating the side channel information as possible. For example, when performing power analysis for monitoring of CPU activity, ideally the power consumption would be collected at the processor. While this may not always be acceptable, it is worth considering as the amount of noise tends to increase with the distance from the source. At the same time, the sampling rate of the side channel is very important. As our model describes the *side channel resolution* has a significant impact on the success rate.

- **Side Channel Preprocessing**: In some cases, it may be possible to filter the side channel information. When performing such analysis, a set of system operations are selected. These operations should be representative of the system/program's activity. When possible, we recommend that the number of these representative system operations be kept as small as possible. For example, in our traffic analysis of Internet of Things work, we filter connections of various protocols since they contribute no value for our mission. Unfortunately, this may not always be possible, as with the high performance computing platform power analysis study.

8.4 Closing Remarks

Regardless of the application and intent, side channel information provides valuable insight into the activity of a system. This information should be considered as both an opportunity and a threat and should be explored from both a theoretical and practical viewpoint. As an opportunity, side channel information presents a non-intrusive, reliable, technology agnostic method for monitoring systems. Unlike other approaches, side channel information can provide a holistic view of a heterogeneous ecosystem. Other beneficial applications of side channel analysis include the potential to disclose information regarding both software and hardware activity. As a threat, this dissertation highlights the extent of information provided by side channels and the severity of the impact on security and privacy.

References

[1] P. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS and other systems," in *Advances in Cryptology*, vol. 96, 1996, p. 104113.

[2] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, vol. 48, no. 5, pp. 701–716, 2005.

[3] T. S. Messerges, E. A. Dabbish, and R. H. Sloan, "Power analysis attacks of modular exponentiation in smartcards," in *Proceedings of the International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 1999, pp. 144–157.

[4] R. M. Avanzi, "Side channel attacks on implementations of curve-based cryptographic primitives." *International Association for Cryptologic Research ePrint Archive*, vol. 2005, p. 17, 2005.

[5] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton, "Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 332–346.

[6] F. Mollers, S. Seitz, A. Hellmann, and C. Sorge, "Short paper: Extrapolation and prediction of user behaviour from wireless home automation communication," in *Proceedings of the 2014 ACM Conference on Security and Privacy in Wireless Mobile Networks*. ACM, 2014, pp. 195–200. [Online]. Available: http://doi.acm.org/10.1145/2627393.2627407

[7] H. Cheng and R. Avnur, "Traffic analysis of SSL encrypted web browsing," 1998.

[8] D. Wagner and B. Schneier, "Analysis of the SSL 3.0 protocol," in *Proceedings of the 2nd USENIX Workshop on Electronic Commerce Proceedings*, 1996, pp. 29–40.

[9] K. Ali, A. X. Liu, W. Wang, and M. Shahzad, "Keystroke recognition using WiFi signals," in *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*. ACM, 2015, pp. 90–102.

[10] M. Backes, M. Dürmuth, S. Gerling, M. Pinkal, and C. Sporleder, "Acoustic side-channel attacks on printers." in *Proceedings of the 19th USENIX Security Symposium*, 2010, pp. 307–322.

[11] V. Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

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

[13] Y. Carmeli, "On bugs and ciphers: New techniques in cryptanalysis," Ph.D. dissertation, Technion, 2015.

[14] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the importance of checking cryptographic protocols for faults," in *Proceedings of the International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 1997, pp. 37–51.

[15] G. W. Hart, "Nonintrusive appliance load monitoring," in *Proceedings of the IEEE*, vol. 80, no. 12. IEEE, 1992, pp. 1870–1891.

[16] T. Zia, D. Bruckner, and A. Zaidi, "A hidden Markov model based procedure for identifying household electric loads," in *Proceedings of the 37th Annual Conference on IEEE Industrial Electronics (IECON)*. IEEE, 2011, pp. 3218–3223.

[17] M. Zhong, N. Goddard, and C. Sutton, "Interleaved factorial non-homogeneous hidden Markov models for energy disaggregation," *arXiv*, 2014.

[18] K. Anderson, A. Ocneanu, D. Benitez, D. Carlson, A. Rowe, and M. Berges, "BLUED: A fully labeled public dataset for event-based non-intrusive load monitoring research," in *Proceedings of the 2nd KDD workshop on data mining applications in sustainability (SustKDD)*, 2012.

[19] J. Gao, S. Giri, E. C. Kara, and M. Bergés, "PLAID: A public dataset of high-resoultion electrical appliance measurements for load identification research: Demo abstract," in *Proceedings of the 1st ACM Conference on Embedded*

Systems for Energy-Efficient Buildings, 2014, pp. 198–199. [Online]. Available: http://doi.acm.org/10.1145/2674061.2675032

[20] N. Batra, J. Kelly, O. Parson, H. Dutta, W. Knottenbelt, A. Rogers, A. Singh, and M. Srivastava, "NILMTK: An open source toolkit for non-intrusive load monitoring," in *Proceedings of the 5th International Conference on Future Energy Systems*, 2014, pp. 265–276. [Online]. Available: http://doi.acm.org/10.1145/2602044.2602051

[21] G. Bauer, K. Stockinger, and P. Lukowicz, "Recognizing the use-mode of kitchen appliances from their current consumption." in *Proceedings of the European Conference on Smart Sensing and Context (EuroSSC)*. Springer, 2009, pp. 163–176.

[22] M. A. Lisovich, D. K. Mulligan, and S. B. Wicker, "Inferring personal information from demand-response systems," in *Proceedings of the IEEE Symposium on Security & Privacy*, vol. 8, no. 1. IEEE, 2010, pp. 11–20.

[23] S. S. Clark, H. Mustafa, B. Ransford, J. Sorber, K. Fu, and W. Xu, "Current events: Identifying webpages by tapping the electrical outlet," pp. 700–717, 2013.

[24] C. Isci and M. Martonosi, "Identifying program power phase behavior using power vectors," in *Proceedings of the IEEE International Workshop on Workload Characterization*. IEEE, 2003, pp. 108–118.

[25] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine, "Privacy vulnerabilities in encrypted HTTP streams," *Lecture Notes in Computer Science*, p. 1, 2006.

[26] A. Hintz, "Fingerprinting websites using traffic analysis," in *Privacy Enhancing Technologies*. Springer, 2003, pp. 171–178.

[27] M. Liberatore and B. N. Levine, "Inferring the source of encrypted HTTP connections," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*. ACM, 2006, pp. 255–263.

[28] Q. Sun, D. R. Simon, Y.-M. Wang, W. Russell, V. N. Padmanabhan, and L. Qiu, "Statistical identification of encrypted web browsing traffic," in *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 2002, pp. 19–30.

[29] D. Herrmann, R. Wendolsky, and H. Federrath, "Website fingerprinting: attacking popular privacy enhancing technologies with the multinomial naïve-Bayes classifier," in *Proceedings of the 2009 ACM Workshop on Cloud Computing Security*. ACM, 2009, pp. 31–42.

[30] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel, "Website fingerprinting in onion routing based anonymization networks," in *Proceedings of the 10th annual ACM workshop on Privacy in the Electronic Society*. ACM, 2011, pp. 103–114.

[31] X. Luo, P. Zhou, E. W. Chan, W. Lee, R. K. Chang, and R. Perdisci, "HTTPOS: Sealing information leaks with browser-side obfuscation of encrypted flows." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2011.

[32] C. V. Wright, S. E. Coull, and F. Monrose, "Traffic morphing: An efficient defense against statistical traffic analysis." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2009.

[33] X. Fu, B. Graham, R. Bettati, W. Zhao, and D. Xuan, "Analytical and empirical analysis of countermeasures to traffic analysis attacks," in *Proceedings of the International Conference on Parallel Processing*. IEEE, 2003, pp. 483–492.

[34] M. Backes, G. Doychev, and B. Köpf, "Preventing side-channel leaks in web traffic: A formal approach." in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

[35] M. Dusi, M. Crotti, F. Gringoli, and L. Salgarelli, "Tunnel hunter: Detecting application-layer tunnels with statistical fingerprinting," *Computer Networks*, vol. 53, no. 1, pp. 81–97, 2009.

[36] T. Kohno, A. Broido, and K. C. Claffy, "Remote physical device fingerprinting," in *Proceedings of the IEEE Transactions on Dependable and Secure Computing*, vol. 2, no. 2. IEEE, 2005, pp. 93–108.

[37] T. Xin, B. Guo, Z. Wang, M. Li, and Z. Yu, "Freesense: Indoor human identification with WiFi signals," *arXiv*, 2016.

[38] Y. Liu, R. Gunasekaran, X. Ma, and S. S. Vazhkudai, "Automatic identification of application I/O signatures from noisy server-side traces," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014, pp. 213–228.

[39] H. Luu, M. Winslett, W. Gropp, R. Ross, P. Carns, K. Harms, M. Prabhat, S. Byna, and Y. Yao, "A multiplatform study of I/O behavior on petascale supercomputers," in *Proceedings of the 24th International Symposium on High-Performance Parallel and Distributed Computing*. ACM, 2015, pp. 33–44.

[40] S. Peisert, "Fingerprinting Communication and Computation on HPC Machines," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-3483E, June 2010.

[41] S. Whalen, S. Engle, S. Peisert, and M. Bishop, "Network-theoretic classification of parallel computation patterns," *International Journal of High Performance Computing Applications*, vol. 26, no. 2, pp. 159–169, 2012.

[42] S. Whalen, S. Peisert, and M. Bishop, "Multiclass classification of distributed memory parallel computations," *Pattern Recognition Letters*, vol. 34, no. 3, pp. 322–329, 2013.

[43] K. Asanovic *et al.*, "The landscape of parallel computing research: A view from Berkeley," 2006.

[44] O. DeMasi, T. Samak, and D. H. Bailey, "Identifying HPC codes via performance logs and machine learning," in *Proceedings of the First Workshop on Changing Landscapes in HPC Security*. ACM, 2013, pp. 23–30.

[45] J. K. Millen, "Covert channel capacity," in *Proceedings of the IEEE Symposium on Security and Privacy*. Oakland, CA, 1987, pp. 1540–7993.

[46] O. L. Costich and I. S. Moskowitz, "Analysis of a storage channel in the two phase commit protocol," in *Proceedings of the Computer Security Foundations Workshop IV*. IEEE, 1991, pp. 201–208.

[47] S.-P. Shieh *et al.*, "Estimating and measuring covert channel bandwidth in multilevel secure operating systems," *Journal of Information Science and Engineering*, vol. 15, no. 1, pp. 91–106, 1999.

[48] R. Gay, H. Mantel, and H. Sudbrock, "An empirical bandwidth analysis of interrupt-related covert channels," *International Journal of Secure Software Engineering (IJSSE)*, vol. 6, no. 2, pp. 1–22, 2015.

[49] M. H. Kang and I. S. Moskowitz, "A pump for rapid, reliable, secure communication," in *Proceedings of the 1st ACM Conference on Computer and Communications Security*. ACM, 1993, pp. 119–129.

[50] P. Chapman and D. Evans, "Automated black-box detection of side-channel vulnerabilities in web applications," in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 263–274.

[51] S. Micali and L. Reyzin, "Physically observable cryptography," in *Proceedings of the Theory of Cryptography Conference*. Springer, 2004, pp. 278–296.

[52] F.-X. Standaert, T. G. Malkin, and M. Yung, "A formal practice-oriented model for the analysis of side-channel attacks," *International Association for Cryptologic Research*, vol. 134, no. 2006, p. 2, 2006.

[53] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the IEEE Symposium on Security and Privacy*. Los Alamitos, CA, USA: IEEE Computer Society, 1982, p. 11.

[54] D. Sutherland, "A model of information," in *Proceedings of the 9th National Computer Security Conference*. DTIC Document, 1986, pp. 175–183.

[55] C.-K. Luk *et al.*, "Pin: Building customized program analysis tools with dynamic instrumentation," vol. 40, no. 6, pp. 190–200, Jun. 2005. [Online]. Available: http://doi.acm.org/10.1145/1064978.1065034

[56] DARPA, "Cyber grand challenge binaries," https://github.com/CyberGrandChallenge/samples, 2014–2015.

[57] T. Trader, "US researcher caught mining for bitcoins on NSF iron," HPC Wire. [Online]. Available: https://www.hpcwire.com/2014/06/09/us-researcher-caught-mining-bitcoins-nsf-iron/

[58] P. Colella, "Defining software requirements for scientific computing," 2004.

[59] A. M. et. al., "PQube phasor measurement unit," Power Standards Lab. [Online]. Available: http://pqubepmu.com/

[60] C. King, "stress-ng." [Online]. Available: http://kernel.ubuntu.com/~cking/stress-ng/

[61] D. H. Bailey *et al.*, "The NAS parallel benchmarks," *International Journal of High Performance Computing Applications*, vol. 5, no. 3, pp. 63–73, 1991.

[62] NERSC, "NERSC-8 trinity procurement benchmarks." [Online]. Available: http://www.nersc.gov/users/computational-systems/cori/nersc-8-procurement/

[63] P. Welch, "The use of fast Fourier transform for the estimation of power spectra: a method based on time averaging over short, modified periodograms," in *Proceedings of the IEEE Transactions on Audio and Electroacoustics*, vol. 15, no. 2. IEEE, 1967, pp. 70–73.

[64] E. Jones, T. Oliphant, P. Peterson *et al.*, "SciPy: Open source scientific tools for Python," 2001–. [Online]. Available: http://www.scipy.org/

[65] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[66] P. Carns, R. Latham, R. Ross, K. Iskra, S. Lang, and K. Riley, "24/7 characterization of petascale I/O workloads," in *Proceedings of the IEEE International Conference on Cluster Computing and Workshops*. IEEE, 2009, pp. 1–10.

[67] "Nest Labs: Home automation company," https://nest.com, 2017.

[68] "Alphabet Inc." https://abc.xyz/, 2017.

[69] "Nest Thermostat," https://nest.com/thermostat/meet-nest-thermostat/, 2017.

[70] "Nest Protect: Smoke and CO Alarm," https://nest.com/smoke-co-alarm/meet-nest-protect/, 2017.

[71] E. Grochowski and M. Annavaram, "Energy per instruction trends in Intel microprocessors," *Technology@Intel Magazine*, vol. 4, no. 3, pp. 1–8, 2006.

[72] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," DTIC Document, Tech. Rep., 1973.

[73] D. Von Oheimb, "Information flow control revisited: Noninfluence= noninterference+ nonleakage," in *Proceedings of the European Symposium on Research in Computer Security*. Springer, 2004, pp. 225–243.

[74] J. Rushby, *Noninterference, transitivity, and channel-control security policies*. SRI International, Computer Science Laboratory, 1992.

[75] J. A. Ambrose, R. G. Ragel *et al.*, "RIJID: random code injection to mask power analysis based side channel attacks," in *Proceedings of the 44th ACM/IEEE Conference on Design Automation*. IEEE, 2007, pp. 489–492.

[76] P. Grabher, J. Grobschadl, and D. Page, "Non-deterministic processors: FPGA-based analysis of area, performance and security," in *Proceedings of the 4th Workshop on Embedded Systems Security*. New York, NY, USA: ACM, 2009, pp. 1:1–1:10. [Online]. Available: http://doi.acm.org/10.1145/1631716.1631717

[77] J. A. Ambrose, R. G. Ragel, S. Parameswaran, and A. Ignjatovic, "Multiprocessor information concealment architecture to prevent power analysis-based side channel attacks," *IET Computers & Digital Techniques*, vol. 5, no. 1, pp. 1–15, 2011.

[78] A. Kurakin, I. J. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," in *Proceedings of the ACM Computing Research Repository (CoRR)*, vol. 1607.02533, 2016.

Appendix A

HPC I/O Analysis: Darshan Features

Below we list the features, with the corresponding index as depicted in the violin plots, used for the Darshan I/O analysis:

0 POSIX_SIZE_READ_100K_1M
1 POSIX_SIZE_READ_10M_100M
2 POSIX_CONSEC_READS/READS
3 POSIX_SIZE_WRITE_0_100
4 POSIX_SIZE_READ_10K_100K
5 POSIX_STRIDE3_STRIDE
6 POSIX_SIZE_READ_4M_10M
7 POSIX_SIZE_READ_100M_1G
8 BYTES/WRITE
9 POSIX_STRIDE4_STRIDE
10 POSIX_ACCESS1_ACCESS
11 POSIX_ACCESS4_ACCESS
12 POSIX_RW_SWITCHES/TOTAL
13 READ_MAX/READ_BYTES
14 POSIX_SIZE_WRITE_100K_1M
15 POSIX_SIZE_READ_1K_10K
16 POSIX_SIZE_READ_1M_4M
17 SEEKS
18 POSIX_CONSEC_WRITES/WRITES
19 POSIX_STRIDE2_STRIDE
20 OPENS
21 POSIX_FSYNCS
22 POSIX_SIZE_READ_1G_PLUS
23 POSIX_STRIDE2_COUNT
24 POSIX_ACCESS2_ACCESS
25 POSIX_SIZE_READ_100_1K
26 POSIX_STRIDE1_STRIDE

```
27 POSIX_STRIDE1_COUNT
28 POSIX_ACCESS3_COUNT
29 POSIX_SEQ_WRITES/WRITES
30 POSIX_SIZE_WRITE_1G_PLUS
31 POSIX_MMAPS
32 POSIX_SEQ_READS/READS
33 POSIX_SIZE_WRITE_100_1K
34 POSIX_STRIDE4_COUNT
35 POSIX_SIZE_WRITE_1M_4M
36 POSIX_SIZE_WRITE_10K_100K
37 POSIX_SIZE_READ_0_100
38 POSIX_STATS
39 WRITE_MAX/WRITE_BYTES
40 POSIX_ACCESS3_ACCESS
41 WRITES
42 POSIX_ACCESS2_COUNT
43 POSIX_STRIDE3_COUNT
44 POSIX_SIZE_WRITE_10M_100M
45 POSIX_SIZE_WRITE_4M_10M
46 READS
47 POSIX_ACCESS1_COUNT
48 POSIX_SIZE_WRITE_100M_1G
49 POSIX_FDSYNCS
50 BYTES/READ
51 POSIX_SIZE_WRITE_1K_10K
52 POSIX_ACCESS4_COUNT
```

Appendix B

Side Channel Theory

B.1 Turing Machine Binary Counter Program

The pseudo-code below represents a Turing machine implementation of a binary counter program:

```
main() {
  no_digits = 0
  while True:
    move_head_right()
    no_digits += 1

  carry = 0
  counter = 0
  while True:
    if counter != 0 and (carry == 0 or counter > no_digits):
      break

    // read input digit
    digit = read()
    if counter == 0:
      // first digit (no carry)
      result = (digit + 1) mod 10
```

```
    else:
      result = (digit + carry) mod 10

    if result == 0:
      carry = 1
    else:
      carry = 0

    write(result)
    move_head_left()
    counter  += 1
}
```