6-28-2019

Centralized and Distributed Detection of Compromised Smart Grid Devices using Machine Learning and Convolution Techniques

Cengiz Kaygusuz
ckayg001@fiu.edu

FLORIDA INTERNATIONAL UNIVERSITY

Miami, Florida

CENTRALIZED AND DISTRIBUTED DETECTION OF COMPROMISED SMART GRID DEVICES USING MACHINE LEARNING AND CONVOLUTION TECHNIQUES

A thesis submitted in partial fulfillment of the

requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Cengiz Kaygusuz

2019

To: John L. Volakis
College of Engineering and Computing

This thesis, written by Cengiz Kaygusuz, and entitled Centralized and Distributed Detection of Compromised Smart Grid Devices using Machine Learning and Convolution Techniques, having been approved in respect to style and intellectual content, is referred to you for judgment.

We have read this thesis and recommend that it be approved.

Kemal Akkaya

Alexander Perez Pons

A. Selcuk Uluagac, Major Professor

Date of Defense: June 28, 2018

The thesis of Cengiz Kaygusuz is approved.

John L. Volakis
College of Engineering and Computing

Andres G. Gil
Vice President for Research and Economic Development and Dean of the University Graduate School

Florida International University, 2019

ACKNOWLEDGMENTS

First and foremost, I would like to thank my mother, father, and big brother in supporting me throughout all phases of my life, which has led to the creation of this work. Nothing would have been possible without their unwavering love and support.

I would like to thank my advisor, Professor A. Selcuk Uluagac, for providing me the opportunity to accomplish this work. His influence shall continue long after my graduation.

I would like to thank Dr. Hidayet Aksu for his intellectual guidance. Working with him was a privilege that I shall treasure ever after.

I would like to thank all of my colleagues in both CSL and ADWISE labs for all their help and companionship.

Lastly, I would like to acknowledge this thesis has been possible by the generous support of the U.S. Department of Energy with their award numbered DE-OE0000779.

ABSTRACT OF THE THESIS

CENTRALIZED AND DISTRIBUTED DETECTION OF COMPROMISED SMART GRID DEVICES USING MACHINE LEARNING AND CONVOLUTION TECHNIQUES

by

Cengiz Kaygusuz

Florida International University, 2019

Miami, Florida

Professor A. Selcuk Uluagac, Major Professor

The smart grid concept has further transformed the traditional power grid into a massive cyber-physical system that depends on advanced two-way communication infrastructure. While the introduction of cyber components has improved the grid, it has also broadened the attack surface. In particular, the threat stemming from compromised devices pose a significant danger: An attacker can control the devices to change the behavior of the grid and can impact the measurements or damage the grid equipment. In this thesis, to detect such malicious smart grid devices, we propose a novel machine learning and convolution-based framework, named PowerWatch, that can run in centralized and distributed settings. After gathering library and system calls, the framework is able to identify how close the observed device is behaving with respect to its normal operations, with mispredictions implying compromise. We evaluated the framework through a state-machine-based computational model of the smart grid devices that explore a wide variety of possible cases that may occur in grid operations: attaining 95.1% accuracy at 0.03% false positive rate over 37500 experiments. The framework was then further tested on a realistic smart grid testbed, where it was able to successfully detect the compromised device in every attack scenario considered in the threat model.

TABLE OF CONTENTS

LIST OF FIGURES

FIGURE PAGE

CHAPTER 1

INTRODUCTION

The ability to sense and react to what is happening in the power grid by smart devices in real time has revolutionized the power industry; by measuring the grid parameters, smart grid devices can control the electrical grid much more safely and efficiently than ever before [FMXY12, YQST13]. Indeed, the introduction of smart devices into the power grid has been a good step in the name of modernization. However, it has also brought challenging security problems that threaten the availability of such a critical infrastructure [WL13].

One of the most vital security problems in the power domain involves compromised smart grid devices [MKB+12]. A compromised device is a device operating in a smart grid environment, the behavior of which is altered in an unauthorized fashion. The damage a compromised device may inflict can be summarized as *an unauthorized modification* causing *undefined behavior*, inflicted by either an insider or outsider party. Within the context of the smart grid, undefined behavior directly implies a severe risk of unavailability of electricity, a threat which must be addressed.

Dangers of compromised devices are best exemplified through a malicious actor. Attacks conducted by such entities to the grid devices may yield dire consequences: A compromised device with sensors that measure the behavior of the power grid may send false information that may cause the control device to raise the voltage, possibly overloading and knocking the grid out in case of a coordinated attack. Similarly, a malicious activity on a control device may accomplish the same hazard directly, again, making electricity unavailable [GBG+11, BAU17]. To ensure a healthy supply of such a critical resource, the devices must be ensured to behave as expected.

On the other hand, compromised devices may operate in the grid even without explicitly malicious actions: a counterfeit device swapped in place of an original one

may also be considered a compromised device. Since this device has not been verified for correct behavior, it may behave differently in certain situations compared to its authentic counterpart. In more extreme cases, the device may even fail altogether.

The modern computation takes place over an operating system (OS). The OS as a whole provides an abstraction layer which presents a uniform interface to the applications and programs, allowing the same logic to be run across multiple devices with zero to minimal modification. This abstraction is great in breadth as any meaningful action such as communication over a network, or requesting space on memory must be accomplished through the OS. The procedures requested from the OS is named as *system calls*. In a similar vein, a program logic often calls procedures from another corpus of logic called *libraries* for arbitrary computation to allow for more efficient development. The use of individual pieces of logic in another library is named as *library call*. Indeed, the oft use of the library and system calls enables one to characterize a program; utilizing this data for intrusion detection has been a subject of research since Kosoresow's initial work [KH97].

To address the problem of compromised devices using the insight provided by call lists, in this paper, we propose a novel framework, named PowerWatch, to detect compromised devices in a smart grid environment. After collecting the system and library call lists from a monitored device, the framework attempts to predict the next call using calls preceding it. The result of this process is then convoluted by an overlapping integral kernel of a predefined size, a technique inspired by convolutional neural networks [Sch15]. The convolution step yields a value named as *activity signal*, intuitively indicating how close the observed device is behaving to the ground truth. The signal is then merely subjected to a configurable threshold: if the signal exceeds the threshold at any given point, then it is decided that the device has been compromised.

We tested and evaluated the framework through a finite-state machine based computational model of the smart grid devices. This model is established with the critical observation that smart grid devices operate in repetitive behavior that is triggered periodically or events such as measurement report or actions to control the grid. The identified state machine is then utilized to generate call lists that are representative of the behavior of grid devices. A total of 37500 experiments were conducted with different parameters such as variables dealing with framework configuration, and ones that represent scenarios that may arise during the operations of the smart grid devices. This corpus of data was both used to analyze how the framework's performance are varying with respect to its configurable values. The results of the extensive evaluation have yielded 95.1% accuracy at 0.03% false positive rate. The framework was further tested to detect attack scenarios on a smart grid testbed where it was successful in detecting different attacks while being able to pinpoint when the devices have become compromised.

To account for the different use cases that may arise in the monitoring of the smart grid, the framework is proposed to be deployed in three distinct configurations. In *stand-alone* mode, the detection framework runs on the kernelspace of the device itself, suitable for monitoring singular devices. In *centralized* mode, collected call lists are sent to a dedicated, centralized cloud environment, offering high reliability and low overhead on grid devices. In *distributed* mode, devices with low computation capacity send their call lists to devices with high computation capacity, suitable for grid devices that have extra computational power, enabling PowerWatch to be operational without any additional hardware costs.

CHAPTER 2

RELATED WORK

The related work chapter is divided into three sections: smart grid security, host-based anomaly detection, and distributed systems. The layout of this section is also representative of the s

The current research concerning the safety of the smart grid focused on protection against specific threats. Proposed solutions mostly focus on the mitigation of false data injections, though there are also a few studies conducted on the behavioral analysis of smart grids.

The general theme of detecting false data injections is through conducting anomaly analysis over data that is sent back and forth between auxiliary data-collecting units and central decision-making devices. Liang et al. [LZL+17] has made a review of such attacks. In [SG14], authors examine how an automatic generation control (AGC) unit can be overloaded through erroneous data, and propose a solution that operates by measuring the divergence from the forecast. In [KT13], authors consider an adversary model that is based on misleading the control center into operating in a different grid topology than what is deployed, through man-in-the-middle attacks; it is mitigated by comparing and contrasting data obtained from various devices in the grid. In [LED+14], a solution based on sparse matrix optimization is proposed, acting on the insight that individual measurements tend to correlate, whereas malicious ones stand on their own. In [KP11], rather than proposing a detection scheme, authors show that it is possible to make the grid immune to data injection attacks if a particular subset of measurement units are guaranteed to send authentic data. Moghaddass et al. conducts anomaly analysis through distribution fitting over the data that comes from smart grid infrastructure [MW18]. Matthews et al. propose a large scale analysis of phase measurements using MapReduce techniques [ML18].

Manandhar et al. detected false data injections into the grid using Kalman filters [MCHL14]. In [CBLM17], a network-based anomaly detection scheme, is proposed using off-the-shelf IoT devices in the grid. Wang et al. focus solely on detecting anomalies arising from PMU's using statistical methods [WLX+17].

We acknowledge that protection against false data injections is of utmost importance, though it is a partial answer to the compromised device problem. An adversary can conduct a false data injection by altering the behavior of the measurement units, which can only detect compromised measurement devices, leaving central devices out. The compromise of the central devices is a non-negligible possibility, and it is self-evident that they cannot be detected by inspecting the data they receive.

As mentioned before, a multitude of studies has been conducted using behavioral analysis. An artificial neural network based solution is proposed to detect malicious acts of voltage control [Kos16] by using photovoltaic power production and weather data, though the corpus of data and the techniques used are specific to solar power production plants. In [SGLL13], researchers employ a rule-based detection mechanism, where rules are determined manually by humans, which is an erroneous source that can be eliminated. Berthier et al. use a particular device's specifications to construct a detector [BS11], which is a solution that does not easily generalize into different devices. The solution proposed by Hong et al. [HLG14] utilizes host-based information in addition to the network, though the host-based information depends on events that could be recorded on the device and, again, cannot be easily generalized into different classes of devices.

Differences from existing work: To the best of our knowledge, the proposed framework, PowerWatch is the first work to tackle the compromised device problem in smart grids in a holistic manner focusing on the behavior of the devices with

system call sequencing, ML, and convolution techniques. Some of the prior art on the smart grid security touches this problem without identifying it; as a result, their solutions do not generalize to a wide variety of circumstances that are considered in this study. In addition to the generalization advantage, our proposed framework utilizes machine learning algorithms and convolution mechanisms in a novel way.

CHAPTER 3

PRELIMINARIES

This chapter provides the background information about the problem domain, *smart grid*, the adversary model, and the approach to remedy them along with its assumptions.

3.1 Smart Grid

The term *Smart Grid* refers to the enhanced version of the electrical grid that improves on its efficiency flexibility. It does not refer to a new invention that replaced the existing traditional grid in a short timespan, but rather smoothly evolved from the previous mechanisms that were in place. Table 3.1 gives a good overview of the differences between the traditional and the form of a grid that is dubbed as *smart*.

Table 3.1: A comparison of traditional and smart grid. [Far10]

Concept	**Traditional Grid**	**Smart Grid**
Control	Electro-mechanical	Digital
Communication	One-Way Communication	Two-Way Communication
Power Generation	Centralized Generation	Distributed Generation
Structure	Hierarchical	Network
Sensing	Few Sensors	Sensors Throughout
Awareness	Blind	Self-Monitoring
Recovery	Manual Restoration	Self-Healing
Flexibility	Failures and Blackouts	Adaptive and Islanding
Testing	Manual Check/Test	Remote Check/Test
Customer Options	Few Customer Choices	Many Customer Choices

The advent of the electrical grid predates computation devices by more than half a century. Before the availability of such devices, the grid was managed by electro-mechanical means. Each entity within the grid was operating independently and blindly. Any anomaly that happened within the grid affected a large portion of it, required human intervention, which is slow compared to electronic devices.

Many of the improvements to the grid that are listed in the later entries in the Table 3.1 has been possible as a result of the improvements in the former entries; namely, the digitization, two-way communication structure made possible by the introduction of the computer networks. However, it also introduced previously known security problems associated with the digital devices into the grid. While this trade-off has been to the benefit of the grid, the introduced security problems must be appropriately addressed to ensure the grid is always operational.

The scholarly investigation of smart grid weaknesses has been a busy topic, so that novel attacks are discovered continuously. In [TSS+18], authors discovered a novel attack that a handful of strategically manipulated measurement results could drastically alter electricity pricing. Chung et al. [CLY+18] propose an attack that combines both cyber and physical aspects of the smart grid to mask line outages and alter the grid topology information in the central systems. A particular attack proposed in 2014 [LCZ+14] makes use of multiple compromised circuit switches to cascade a local failure in the grid.

3.2 Adversary Model

We consider an adversary that can perform unauthorized modification of a device. The space of all such threats are vast enough not to permit individually addressing them; however, we hypothesize that all of them, by, definition, involves the modification of the logic in execution, hence could be discovered by spotting anomalies in the execution. To serve as an example, the space of all threats can be grouped into three categories:

- *Direct grid control with specific commands:* A compromised command and control device such as an IED (Intelligent Electronic Device), may allow the

attacker - internal or external - to issue commands directly to affect the state of the grid. Hijacking the control of an IED is an example of such an attack.

- *Indirect grid control via fake measurements:* A compromised measurement unit may send fake measurements to exert control over the smart grid indirectly. Poisoning measurements are one way to accomplish this kind of effect.
- *Surveillance of sensitive data:* A compromised device may allow an insider or external third-party to gather sensitive, confidential data; namely, leak information about the state of the grid by using the communication capabilities of the devices.

The adversary may modify a device by the following means:

- *Online*: The adversary may remotely connect to a device and modify it.
- *Offline* The adversary may physically interact with the device to modify it.

These examples illustrate what an adversary may achieve by unauthorized modification. We ultimately assume that any *unauthorized* modification is potentially malicious and compromises the smart grid device.

3.3 Device Model

A key observation about smart grid devices is that certain parts of the programs they are running are repetitively executed [TBAC11, BAU17]: They respond to the events they receive in the smart grid in a deterministic fashion. Figure 3.1 gives an example of the operations of an IED represented as a state machine. The figure exemplifies the fact that the device stays idle for most of the time: on detecting a frequency anomaly, the device conducts necessary computation to trip the circuit

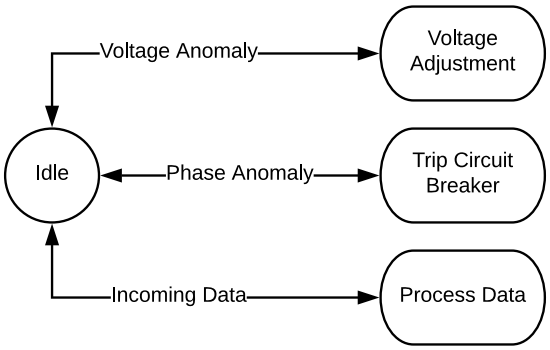


Figure 3.1: An example of operation of smart grid devices.

breaker. Likewise, on detecting higher than nominal voltage, the device again does the computation to lower the voltage; essentially behaving in a "reactive" fashion. Such a reactive nature is inherent in all kinds of devices capable of computation that operate for extended periods and can be generalized into other smart grid devices as well. Figure 3.2 presents a generalization of these smart grid devices as a state machine. The number of states represents a different kind of actions a device can take. This abstract model of smart grid devices is used both in the call trace analysis and evaluation extensively.

Another observation is that in ordinary operation, the device strictly operates in known deterministic states; in other words, the capabilities and tasks of a smart grid device are well known a priori. If a smart grid device is compromised, a previously unknown state is added to the device.

In the first sight, it is not evident that previously unknown behavior - however it is measured - would imply the device had been compromised. Unknown behavior implies a compromised device in a specific context. Such an event is similar to a user running a previously unknown program on her personal computer. The same logic cannot be applied in such a situation to detect whether the user's computer is compromised; *the main difference lies in the fact that a user may run an arbitrary*

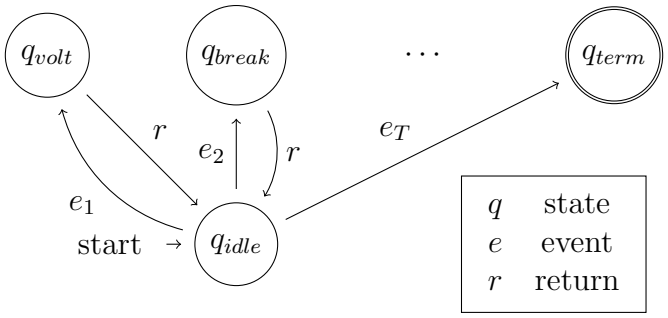


Figure 3.2: The generalization of smart grid devices as a state machine. Two states for voltage and circuit breaking is given as an example. Many more states may be present in a grid device.

program in her computer, while the smart grid environment does not offer such a feature. In the case of smart grid devices, since the logic executed in smart grid devices are well known before they are operational, and this logic is strictly controlled by the utility or an authorized device vendor, any code execution profile that diverges from the known implies the device has been compromised.

The framework also assumes that the *known* behavior is deterministic: It translates well into the context of this study as, known states output known call lists. The determinism is one of the fundamental concepts that enable a device to be engineered in the first place: if the device would respond chaotically to stimuli, then it cannot be used to achieve a task, e.g., controlling the voltage level.

Non-determinism in computer logic is nearly always the result of multithreading, and in rare cases, the presence of quantum phenomena [LJL+10] which can be safely disregarded as smart grid devices does not exhibit quantum behavior. The multithreaded non-determinism must be considered as grid devices may be implemented in such a fashion, in which case the obtained call lists will be out-of-order and lose most of its characteristic information. Fortunately, this problem can easily be circumvented if we consider individual threads, which are strictly deterministic.

Process Control	CreateProcess()	fork()
	ExitProcess()	exit()
	WaitForSingleObject()	wait()
File Manipulation	CreateFile()	open()
	ReadFile()	read()
	WriteFile()	write()
	CloseHandle()	close()
Device Manipulation	SetConsoleMode()	ioctl()
	ReadConsole()	read()
	WriteConsole()	write()
Information Maintenance	GetCurrentProcessID()	getpid()
	SetTimer()	alarm()
	Sleep()	sleep()
Communication	CreatePipe()	pipe()
	CreateFileMapping()	shmget()
	MapViewOfFile()	mmap()
Protection	SetFileSecurity()	chmod()
	InitlializeSecurityDescriptor()	umask()
	SetSecurityDescriptorGroup()	chown()

Figure 3.3: A selection of system calls for Windows and Linux operating systems.

3.4 System and Library Calls

The modern computation takes place through an operating system (OS). An OS is a gigantic abstraction layer that orchestrates a device's operation and provides a unified and standardized interface for utilizing device resources, such as disk, network, memory, and many others. Figure 3.3 gives a list of example system calls and their functionality for two operating systems.

It is unfeasible to not use system calls, as making any kind of action other than pure computation such as reading a file from disk, allocating-deallocating additional memory, communicating with other devices on the network, and using abstractions such as memory sharing, communicating with other processes etc. require a system call to be made. It is essential for authentic programs as well as adversarial actors to use system calls; a program that is disallowed to use them are necessarily only

capable of performing arithmetic operations at the machine level. Such programs could be safely ignored as the connected nature of the smart grid at least requires a device to use its networking capabilities, which mandates the use of system calls.

Library calls, as its name suggests, are function calls that are made to use the functionality of a *shared* library, a collection of commonly used operations. The most commonly used shared library is the standard *C* library, which provides functionalities like comparing byte segments, mathematical functions, and operations such as sine, cosine, and square root, and the means of receiving input and giving output. Conceptually, a program may not use any shared library to operate. In practice, however, shared libraries are one of the staple practices that help to deliver fast and maintainable software. Besides, it is also a standard practice to use the standard *C* library to invoke system calls as for many of them; numerous data structures must be instantiated, which is tedious and error-prone. The shared libraries are used widely in practice, and the mandating the use of them bear no constraints on the viability of library calls.

The system and library call lists (collectively to be referred to as *call lists* hereafter) are artifacts that could be utilized to map a program to the abstract state machine depicted in Figure 3.2. The critical observation to be made here is that, since system and library calls have to be utilized in order for a device to do any meaningful computation, the states can be identified by the calls they make. An example of a trace of library calls belonging to a program implemented in *libiec61850* is given in Figure 3.4. It can be seen that after the initialization period, the program executes `strlen`, `sendto`, and `usleep` repeatedly in a recognizable pattern.

The states need not be uniquely identifiable as it is not the ultimate goal of this study. Instead, we are interested in whether the device has operated in a state that has not been present before, and it is assumed that the state is potentially

```
09:43:38 strncpy(0x7ffd5bde7450, "lo", 16)                                                  = 0x7ffd5bde7450
09:43:38 ioctl(3, 35123, 0x7ffd5bde7450)                                                   = 0
09:43:38 ioctl(3, 35091, 0x7ffd5bde7450)                                                   = 0
09:43:38 ioctl(3, 35092, 0x7ffd5bde7450)                                                   = 0
09:43:38 memset(0x55b6681af844, '\0', 8)                                                   = 0x55b6681af844
09:43:38 memcpy(0x55b6681af844, "\001\f\315\001\0\001", 6)                                 = 0x55b6681af844
09:43:38 malloc(1518)                                                                      = 0x55b6681af860
09:43:38 memcpy(0x55b6681af860, "\001\f\315\001\0\001", 6)                                 = 0x55b6681af860
09:43:38 gettimeofday(0x7ffd5bde74f0, 0)                                                   = 0
09:43:38 calloc(1, 21)                                                                     = 0x55b6681afe60
09:43:38 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:38 malloc(42)                                                                        = 0x55b6681afe80
09:43:38 memcpy(0x55b6681afe80, "simpleIOGenericIO/LLN0$GO$gcbAna"..., 42)                 = 0x55b6681afe80
09:43:38 strlen("simpleIOGenericIO/LLN0$AnalogVal"...)                                     = 35
09:43:38 malloc(36)                                                                        = 0x55b6681afec0
09:43:38 memcpy(0x55b6681afec0, "simpleIOGenericIO/LLN0$AnalogVal"..., 36)                 = 0x55b6681afec0
09:43:38 usleep(1000000)                                                                   = <void>
09:43:39 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:39 strlen("simpleIOGenericIO/LLN0$AnalogVal"...)                                     = 35
09:43:39 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:39 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:39 strlen("simpleIOGenericIO/LLN0$AnalogVal"...)                                     = 35
09:43:39 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:39 sendto(3, 0x55b6681af860, 201, 0)                                                 = 201
09:43:39 usleep(1000000)                                                                   = <void>
09:43:40 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:40 strlen("simpleIOGenericIO/LLN0$AnalogVal"...)                                     = 35
09:43:40 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:40 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:40 strlen("simpleIOGenericIO/LLN0$AnalogVal"...)                                     = 35
09:43:40 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:40 sendto(3, 0x55b6681af860, 201, 0)                                                 = 201
09:43:40 usleep(1000000)                                                                   = <void>
09:43:41 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:41 strlen("simpleIOGenericIO/LLN0$AnalogVal"...)                                     = 35
09:43:41 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:41 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:41 strlen("simpleIOGenericIO/LLN0$AnalogVal"...)                                     = 35
09:43:41 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)                                     = 41
09:43:41 sendto(3, 0x55b6681af860, 201, 0)                                                 = 201
```

Figure 3.4: An example of library calls taken by: strlen, sendto, malloc, and more.

malicious. At this point, it is natural to ask what happens if the malicious state outputs the same type of call lists. In this case, the malicious state is strictly constrained to imitate one of the constrained states. For example, in the example given in Figure 3.4, an adversarial agent cannot establish a connection to a remote server to receive commands, which requires a `socket` or equivalent system call to be made, which would be detected right away.

CHAPTER 4

INVESTIGATION OF CALL LIST PATTERNS

Two call lists are finite sequence of symbols which could be compared element-by-element to establish their equivalence. However, direct equivalence is of little use since a device's computation history may differ from observation to another a multitude of factors, such as time of observation, device's initial state, and events within the real world. However, if we measure the same device unaltered, there are some similarities in the patterns of computation.

Even though exact equality cannot be used, the categories of equality can be used to extract statistical information to form a basis of *similarity* between two call lists. The equality of call lists can be broken down into three categories: *type, number*, and *order* of calls. In this chapter, we aim to formally define the metrics of type, number, and order of call lists, and reason how they can be utilized in detecting anomalies in them.

4.1 Definitions

$$SD\left(\begin{vmatrix} malloc \\ malloc \\ free \\ free \end{vmatrix}, \begin{vmatrix} malloc \\ free \end{vmatrix}\right) = 0. \tag{4.1}$$

Set Distance $SD(L_1, L_2)$ is the measure of how two call lists are different from each other according to the type of calls they inhibit. Let A be the set of calls in L_1, and B the set of calls in L_2. The function $SD(L_1, L_2)$ is simply the number of *unique* elements (cardinality) that is contained in A, but not in B. Formally stated, $SD(L_1, L_2) = |A - B|$. Note that $SD(L_1, L_2) \neq SD(L_2, L_1)$. Equation 4.1 gives an example where $SD = 0$.

$$LD\left(\begin{vmatrix} malloc \\ malloc \\ malloc \end{vmatrix}, \begin{vmatrix} free \\ free \\ free \end{vmatrix}\right) = 0. \tag{4.2}$$

Length Distance $LD(L_1, L_2)$ is simply the difference of number of system calls contained by two call lists. $LD = 0$ indicates two call lists are of the same length, while $LD \neq 0$ indicates one list is longer than another by given amount, without specifying which one it is. Equation 4.2 gives an example where $LD = 0$. Note that $LD(L_1, L_2) = LD(L_2, L_1)$.

$$ED\left(\begin{vmatrix} malloc \\ malloc \\ free \\ free \end{vmatrix}, \begin{vmatrix} malloc \\ free \\ malloc \\ free \end{vmatrix}\right) = 0. \tag{4.3}$$

Euclidean Distance $ED(L_1, L_2)$ is a measurement unit that intermixes both type and length difference between two call lists. v_{L_i} is an N dimensional vector where each dimension is mapped to total number of calls made to that particular system or library function belonging to call list L_i. With this definition, $ED(L_1, L_2)$ is simply equal to $|v_{L_1} - v_{L_2}|$, or $ED(L_1, L_2) = |v_{L_1} - v_{L_2}|$. Equation 4.3 gives an example where $ED = 0$.

$$HD\left(\begin{vmatrix} malloc \\ malloc \\ free \\ free \end{vmatrix}, \begin{vmatrix} malloc \\ free \\ malloc \\ free \end{vmatrix}\right) = 2. \tag{4.4}$$

Hamming Distance $HD(L_1, L_2)$ is simply the number of operations required to be undertaken in order to make two call lists identical. Note that $HD(L_1, L_2) = 0$ implies two lists are identical. Equation 4.4 gives an example where $HD = 2$.

4.2 Direct Use of Distance Metrics

An obvious insight is to use these metrics directly to spot differences in the call lists; however, this is not possible due to problems with device determinism. While the devices themselves are strictly deterministic, it is not possible to predict all of the real world situations in a deterministic fashion. For example, it is well defined what an IED will do if it spots a phase anomaly, or receives a data packet from a particular sensor, but it is not feasible to calculate ahead-of-time the order of these events happening (if they happen at all), and the direct use of these metrics require a rigid determinism as in the example.

The state machine given in Figure 3.2 can be used to clearly illustrate the problems with direct use. Suppose that the machine operates under the following terms:

- The machine operates in *turns*.
- At each turn, the machine transitions to one of the states except the initial and the terminal state.
- After the transition, the state machine outputs a *call list*.
- On performing a certain number of turns, the machine halts by transitioning to the terminal state.

To permit direct use, one must possess two pieces of critical information: the total amount of turns the machine is going to operate, and all of the states which the machine is going to transition at each turn, both of which are not possible to predict reliably.

4.3 Statistical Use of Distance Metrics

The effect of the individual computing states exerts a bias in the call list produced by the machine that could be measured using statistical methods. For example, if we consider two states, one of which produces double the amount of one of the particular call, and the activation probabilities of those two states are the same, then the state machine which has the mentioned state will contain the call more on average. Following is a list of difference metrics and their explanations:

- $SD(L_M, L_C) \neq 0$: A call list contains a type of call that is not contained in the other call list.
- $LD(L_C, L_M) \neq 0$: Two call lists inhibit the same type of calls, but differing in length.
- $ED(L_C, L_M) \neq 0$: Two call lists are of the same length and contain the same type of calls, but their internal distributions are different.
- $HD(L_C, L_M) \neq 0$ Two call lists are of the same length, contain the same type of calls, their internal distributions are the same, but their orders are not identical.

Two call lists created by the states differing by the first three different cases can be differentiated by calculating the average of the call counts for each type of call that may happen. The last case, however, eludes this scheme, as in this case, two call lists differ only in the order of the calls and harbor no statistical difference.

4.4 Completeness of Hamming Distance

Indeed, the Hamming Distance is *complete*, in a sense that this metric can calculate a difference that may be spotted by all of the other methods, in addition to the

differences that could only be spotted by this metric. We will now provide proof for this claim.

Assume that L is the set containing all possible non-empty finite call lists, with $L_i, i \in \mathbb{N}$ indicating the ith call list of L. Note that for the purposes of this proof, we are only interested in the fact that $\forall i, j \in \mathbb{N} : i \neq j \implies L_i \neq L_j$.

Lemma 4.4.1 *For all pairs of call lists, if their HD is zero, then all other metrics are also zero:*

$$\forall i, j \in \mathbb{N} \quad HD(L_i, L_j) = 0 \implies SD(L_i, L_j) = 0, \tag{4.5a}$$

$$\forall i, j \in \mathbb{N} \quad HD(L_i, L_j) = 0 \implies LD(L_i, L_j) = 0, \tag{4.5b}$$

$$\forall i, j \in \mathbb{N} \quad HD(L_i, L_j) = 0 \implies ED(L_i, L_j) = 0. \tag{4.5c}$$

Proof. It is sufficient to simply make the following statement:

$$\forall i, j \in \mathbb{N} \quad HD(L_i, L_j) = 0 \iff i = j. \tag{4.6}$$

It means the hamming distance is zero only when two lists are equal. Since the distance of a call list with itself is zero by all metrics, the propositions hold true. □

Lemma 4.4.2 *For all metrics other than HD (namely, ED, SD, LD) there exist pairs of call lists such that their distances are 0 under that metric, and greater than 0 in HD.*

$$\exists i, j \in \mathbb{N} : SD(L_i, L_j) = 0 \ \wedge \ HD(L_i, L_j) > 0. \tag{4.7a}$$

$$\exists i, j \in \mathbb{N} : LD(L_i, L_j) = 0 \ \wedge \ HD(L_i, L_j) > 0. \tag{4.7b}$$

$$\exists i, j \in \mathbb{N} : ED(L_i, L_j) = 0 \ \wedge \ HD(L_i, L_j) > 0. \tag{4.7c}$$

Proof. We shall prove each proposition individually, with a generic approach where we will show in each set whose elements are zero with respect to a metric other than HD, there exist pairs of call lists which their distance are non-zero under HD.

To prove 4.7a, let S^i be the set of all possible non-empty call lists, with S^i_j is the jth call list in the set, where members of the set S^i have a set distance of 0 to L_i:

$$S^i = \{S^i_j : \forall j \in \mathbb{N} \quad SD(S^i_j, L_i) = 0\}. \tag{4.8}$$

The set of S_i is of infinite size. Recall that it contains *all* possible call lists, of which the only constraint is that they contain at least one type of call that is present in L_i. Since there is not an upper limit on the number of a specific call, it follows that the set is infinite. Among these call lists of S_i, by Equation 4.6 there is only one call list that HD is zero, and for all others, it yields a nonzero value; hence Proposition 4.7a holds.

For Proposition 4.7b, we follow a similar procedure. Let G_i be the set of all possible non-empty call lists, with G^i_j is the jth call list of G^i, where members of the set G^i has a length distance of 0 to L_i:

$$G^i = \{G^i_j : \forall j \in \mathbb{N} \quad LD(G^i_j, L_i) = 0\}. \tag{4.9}$$

Let $T_i \in \mathbb{N}$ be the length of L_i, and $O_i \in \mathbb{N}$ be the number of types of calls of L_i. The following equation

$$|G^i| = {T_i}^{O_i}, \tag{4.10}$$

where $|.|$ describes the size of the set G^i. Since both $T_i > 1$ and $G_i > 1$, the value $|G^i| = {T_i}^{O_i} > 0$. By Equation 4.6, for only one member of the set G_i yields $HD = 0$; hence the equation 4.7b holds.

For Equation 4.7c, let E^i be the set of all possible call lists with E^i_j is the jth call list of E^i, where all members of the set E^i have an euclidean distance of zero to L_i:

$$E^i = \{E^i_j : \forall j \in \mathbb{N} \quad ED(E^i_j, L_i) = 0\} \tag{4.11}$$

Recall that $v_i \in \mathbb{N}^{O_i}$, where each dimension indicates the total amount of calls made by that particular type of call. The size of the set E^i can be described as a well-known combination problem, concerned with the number of all possible orderings of a total of $\sum_i v_{Li}$ elements, with each element belonging to an element class, calculated as shown in the following equation.

$$|E^i| = \frac{(\sum_i v_{Li})!}{\prod_i (v_{Li}!)} \tag{4.12}$$

It is easy to see $|E^i| > 1$ barring trivial cases. By Equation 4.6, only one member of the set yields $HD = 0$; thus, the proposition holds. $\square$

To summarize the two lemmas: the first lemma indicates if the hamming distance is zero (i.e., two call lists are of the same pattern), then, it implies they are equal and cannot be distinguished by any metric. The second lemma indicates there exist call lists that could only be distinguished by hamming distance, i.e., pattern analysis. These two indicate that a pattern analysis algorithm can distinguish between every pair of call lists where such differentiation is possible.

In light of this information, in the next chapter, we describe the detection framework.

CHAPTER 5

THE ARCHITECTURE OF POWERWATCH FRAMEWORK

This chapter describes the PowerWatch, a framework that processes call traces with the purpose of identifying compromised devices. The core trace analysis algorithm is described first, followed by the operation modes of PowerWatch, developed to address different infrastructure requirements.

5.1 Trace Analysis Algorithm

5.1.1 Overview

The general architecture of the framework is given in Figure 5.1. Two principal phases of the algorithm are described below, followed by the detailed description of the individual steps.

Training

In this phase, the device is run while being guaranteed not to be compromised, and its call lists are harvested. These call lists are then pre-processed by a procedure called *bucketing*. Using this data, the machine learning model is then trained. Another training step is then conducted to compute the activity index threshold, where the entire algorithm is run against an authentic device, and the maximum value of the activity signal is picked as the threshold.

Monitoring

In this phase, the device is deployed into the field and is being monitored through its call lists. After an adequate amount of calls are harvested, it is first pre-processed to be fed into the machine learning model. *The model is principally used to predict*

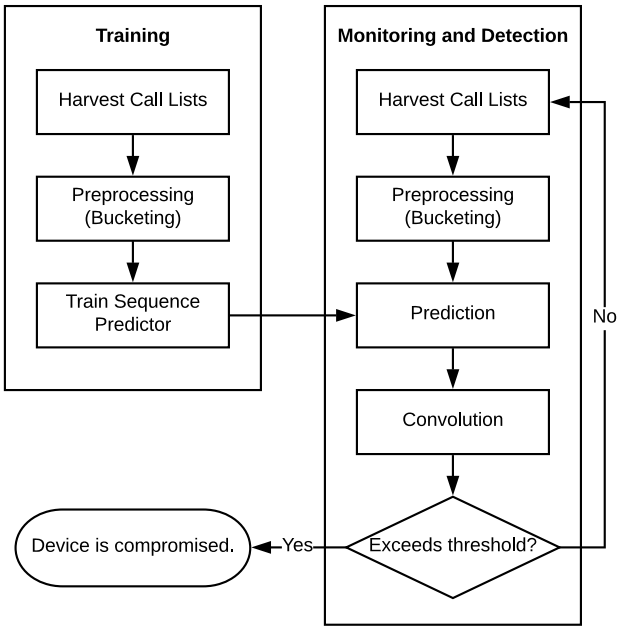


Figure 5.1: Flowchart for the call list analysis algorithm.

a call in the list by looking at calls immediately preceding to it. The results of the predictions are then analyzed; for each call, if the prediction was successful, it is marked 0, if not, it is marked 1. This array of binary values are then convoluted by an integral kernel, which sums its values within the convolution window of interest. If there is a value within the convoluted signal that exceeds the threshold, it is decided that the device has been compromised.

5.1.2 Detailed Steps

In this section, we detail the major steps given in Figure 5.1.

Harvesting Call Traces

As its name suggests, library and system call lists are harvested at the first stage. Since the framework records no information other than the type of the call, a call list can be thought as a sequence of symbols. An extensive study on collecting call

```
1557698613.409616 malloc(48)
1557698613.409994 malloc(1518)
1557698613.410198 malloc(16)
1557698613.410398 printf("Set interface id: %s\n", "lo"Set interface id: lo
)                                          = 21
1557698613.411163 strlen("lo")
1557698613.411434 malloc(3)
1557698613.411690 memcpy(0x56029b6eccd0, "lo\0", 3)
1557698613.411949 calloc(1, 96)
1557698613.412481 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)
1557698613.413341 malloc(42)
1557698613.413546 memcpy(0x56029b6ecd60, "simpleIOGenericIO/LLN0$GO$gcbAna"..., 42)
1557698613.414033 strlen("simpleIOGenericIO/LLN0$GO$gcbAna"...)
1557698613.415228 calloc(1, 21)
1557698613.415618 malloc(16)
1557698613.415882 malloc(32)
1557698613.415971 pthread_create(0x56029b6ecdf0, 0, 0x56029b4494b0, 0x56029b6ec260)
1557698613.416206 signal(SIGINT, 0x56029b4444fc)
1557698613.416361 signal(SIGUSR1, 0x56029b444679)
1557698613.416496 usleep(1000000)
1557698614.417212 usleep(1000000)
1557698615.418233 usleep(1000000)
1557698616.419125 usleep(1000000)
```

Figure 5.2: An example of ltrace output.

lists can be found in Lopez's work [LBAU17]. In this study, we opted to use *ltrace*, one of the standard programs when it comes to collecting system or library calls in the Linux operating system. An example of the ltrace output can be found in Figure 5.2.

Both system and library calls are traced by *ltrace* with the invocation of a particular system call, `ptrace`, which is a mnemonic for *process tracing*. This system call is used by *ltrace* to attach itself to the monitored process, which enables *ltrace* to insert a special instruction that temporarily halts the program and notifies an external process (this instruction is `0xCC` in case of Intel architectures). On the notification that the monitored process has been halted, ltrace examines the memory contents of the process, records the system and library call, and lets the operating system resume the process.

Algorithm 1: Bucketing (Preprocessing) Stage

```
   input  : system or library call list
   output: bucketed call list
1  begin
2      S_bucket ← configure()
3      S_window ← configure()
4      T ← configure()
5      for i ← 1 to input.length - S_bucket do
6          for j ← 1 to S_bucket - 1 do
7              R[i][j] ← input[i + j - 1]
8          R[i].last ← input[i + S_bucket - 1]
9      return R
```

Bucketing (Pre-processing)

After the framework obtains the raw data which is a sequence of symbols, they are arranged in a way that the sequence predictor accepts. The number of calls to be inspected, i.e., the look-behind value is named as *bucket size*. In the preprocessed dataset, each row has *bucket size* columns, where the last column is the target call, and the previous columns are previous calls, respectively. The optimal bucket size value is determined experimentally in the evaluation section. Algorithm 1 outlines this procedure.

Prediction

At this point, the machine learning model is fed with the preprocessed data, which then begins to predict each call by looking at previous calls. For each call, if the prediction was successful, it emits 0, if it fails, 1 is emitted. This process is described in Algorithm 2.

The choice of specific numbers – 0 for *valid* predictions, and 1 for *invalid* predictions – is not intuitive at first sight, since traditionally 0 and 1 represent *false* and *true*, respectively. In our case, PowerWatch essentially measures the diver-

Algorithm 2: Prediction Stage

```
   input  : Bucketed Call List
   output: Raw Prediction Signal
 1 begin
 2     S_bucket ← configure()
 3     T ← configure()
 4     for i ← 1 to R.length do
 5         C_target ← R[i].last
 6         C_predict ← predict(R[i])
 7         if C_predict = C_target then
 8             P[i] ← 0
 9         else
10             P[i] ← 1
11     return P
```

gence of the program logic execution, with the aim of higher values representing greater divergence. The numerical value is obtained in the convolution step, which its explanation will help in understanding this design decision better.

Convolution

Algorithm 3: Convolution Stage

```
   input  : Raw Prediction Signal
   output: Activity Signal
 1 begin
 2     T_A ← input
 3     for i ← 1 to T_A.length − S_window do
 4         T_B[i] ← sum from T_A[i] to T_A[i + S_window]
 5     return T_B
```

As the last step before the decision, the binary signal obtained from the prediction stage is sum-convoluted by an overlapping integral kernel of size called *window size*. In other words, the binary signal is iterated through a sliding window of size window size, wherein each iteration, every element within the window is summed. Result of this operation is named as *activity signal*, and each element in the signal

is called *activity index*. The mentioned index can also be thought of as an index of corruption and has a simple and useful property that having a high index indicates suspicious activity. The optimal window size value is determined experimentally in the evaluation section. Algorithm 3 outlines this procedure.

Thresholding

The last step is straightforward: if any element within the activity signal exceeds a threshold, then it is decided that the device is compromised. The threshold is picked by running the framework on a corpus of call lists that are known to be coming from an authentic device, but not used to train the sequence predictor.

5.2 Operational Modes

This section discusses the different methods of how PowerWatch may be used. We propose three different setups: stand-alone, centralized, and decentralized deployment. Different deployment cases are proposed in order to satisfy possible monitoring requirements efficiently, listed as follows:

- *Stand-Alone Mode*: In this mode, all the tracing steps - from call list harvesting to thresholding - is done on a device. This is the simplest use case and effective for monitoring single devices, however, the tracing logic induces an overhead to the device, and each monitored device is needed to be managed separately.
- *Cloud-Assisted Mode*: The harvested data is sent to the cloud for processing, dedicated to this task. This mode induces the least amount of overhead to a given device, as gathering data and sending it over the network has been found to be efficient. In addition, a dedicated server cluster ensures the availability of processing power for the monitoring task, increasing PowerWatch's reliability.

Figure 5.3: Standalone deployment scheme of PowerWatch.

These perks come at the cost of increased economic expenses for the cloud service.

- *Distributed Mode*: This mode utilizes the binary classification of smart grid devices, namely *resource-rich* and *resource-limited* devices (referred as *rich* and *limited* devices hereafter). The devices which are incapable of running the tracing logic (limited) send their call lists to *rich* devices, which possess adequate resources to handle such a task. As this mode utilizes already existing devices within the network to conduct the analysis, no additional hardware expenses are required, however, depending on the resource utilization of the grid devices, it may not be feasible to use this mode without diminishing the capabilities of the grid devices.

5.2.1 Stand-Alone Mode

Figure 5.3 outlines the stand-alone operation mode. In this scheme, the module that conducts the call trace analysis is placed on the device that is being monitored. Elements of this scheme is explained as follows:

- *Monitored Process*: As the name clearly indicates, this is the program that is being monitored. It runs in the userspace, without superuser privileges.

- *Tracer*: This module records both the system and library calls from a device. This module must be implemented at the operating system level (kernelspace) as the operating system is fool-proof in terms of tracing system calls: no program without compromising the operating system itself may conceal the system calls it is making.
- *Call List Analyzer*: This module executes the call list analysis algorithm described in Chapter 5.

Since the analyzing module is placed on the same device that its integrity is in question, it is natural to ask why such a setup is employable. Recall that the operating system integrity was one of the assumptions that were made in Chapter 3. The breach of the operating system is much more complicated than modifying userspace programs, and once done, it requires hardware-assisted detection methods [WL13] that requires replacement of the working devices, which our solution is trying to avoid. Within a device, a small, trusted segment that is capable of computation is adequate to monitor the rest of the system, which in our case, this segment is the operating system.

5.2.2 Cloud-Assisted Mode

In this setup, the gathered data is sent from a monitored device to a centralized cloud system for processing. The details of this scheme is given in Figure 5.4. The main advantage of the cloud setup is reliability and availability by dedicating a cluster of hosts specifically for analyzing the call list traces. Its design follows service-oriented architecture (SOA) [EAA+04] patterns. It can be described as a computation cluster that is accessible behind a reverse-proxy server. The entities present in the Figure 5.4 are explained as follows:

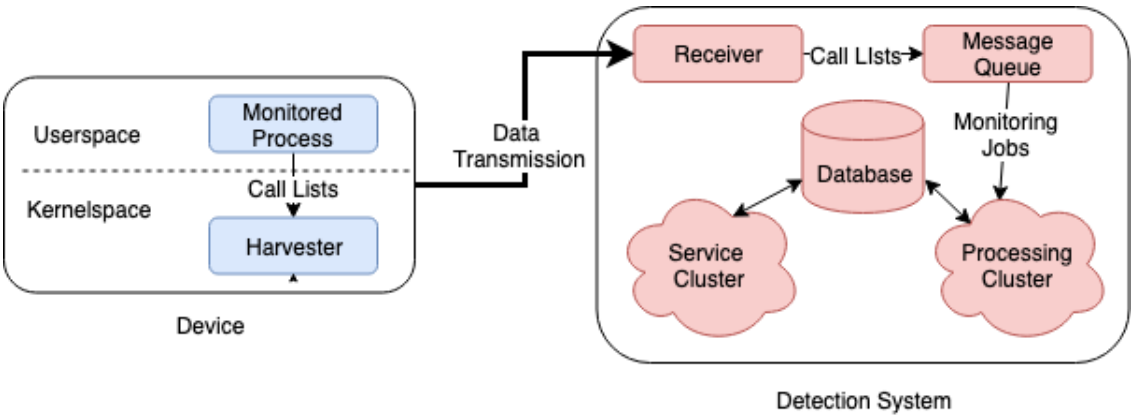


Figure 5.4: Cloud-assisted deployment scheme of PowerWatch.

- *Device*: This is the device that its monitoring is of interest. Differently, from the stand-alone deployment, it does not processes the call list traces on itself but sends it to a remote server.
- *Detection Cluster*: This is a collection of hosts that conducts the call list analysis.
- *Receiver*: The reverse-proxy server that is intended as an entry-point to the analysis system. It can be thought as an application-layer router that forwards the incoming connection to the relevant subsystem (only the relationship with the message queue is shown for brevity) or rejects the connection altogether.
- *Message Queue*: This entity basically distributes the analysis jobs (messages in terminology) to the processing cluster, and allows for asynchronous processing for various tasks which improves the responsivity of the system and is a typical in service-oriented software architectures [EAA+04].
- *Processing Cluster*: Contains the hosts that are dedicated to the call list analysis, and other logic that need to be executed periodically or asynchronously.

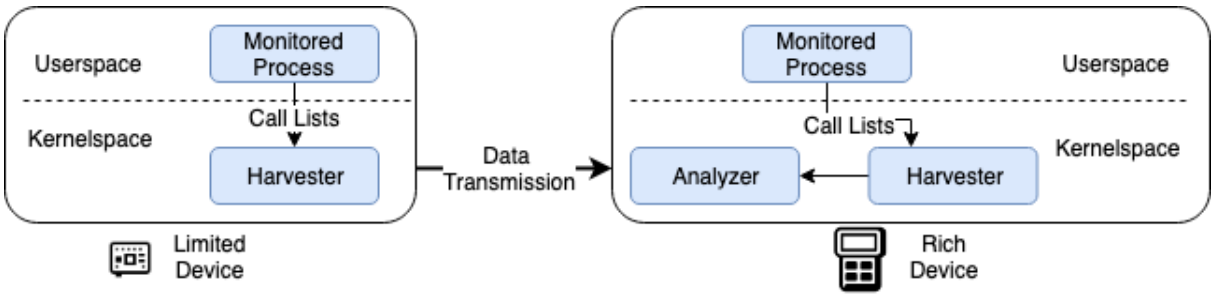


Figure 5.5: Information flow in distributed deployment scheme of PowerWatch.

- *Service Cluster*: This cluster is dedicated for human access to the state of the monitoring system. Its resources are completely isolated from the processing cluster to ensure the system state can be accessed at all times.
- *Database*: As its name indicates, it holds and persists all the information relevant to the analysis task, and ordinary application data.

The two clustered parts – service and processing – are done so in order to ensure availability of computation power. For example, if there is not enough capacity for call list trace analysis, the near-real-time property of the system is lost, and a compromised device may be noted much later after its occurrence. In a similar fashion, if the service cluster is not available, the computed activity index values may not be able to read from the system, making its purpose nullified.

5.2.3 Distributed Mode

The cloud-assisted deployment scheme requires additional hardware to be functional. The distributed deployment scheme aims to evade dealing with extra hardware by utilizing an existing device's computation power to conduct call trace analysis. More specifically, the devices are classified into two: *resource-rich* and *resource-limited* (to be referred as *rich* and *limited* devices hereafter), with the distinction that rich devices are capable enough to monitor limited devices. Figure 5.5 illustrates the

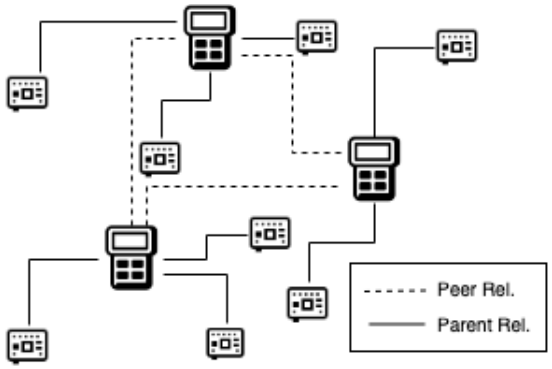


Figure 5.6: In decentralized deployment, rich devices monitor limited devices.

information flow in both between the devices and within the device itself. Figure 5.6 illustrates the idea of rich devices monitoring limited devices.

The distributed scheme requires each device to be able to function independently (e.g., a rich device should be able to monitor itself and any other limited device connected to itself even though it knows nothing about other devices), however, the system still needs to be configurable. To achieve this goal, we intend any rich device to serve as an entry point to PowerWatch, holding all relevant information to access the other rich devices in the network.

Before revealing any more information about the details, a shortlist of terminology is given below:

- *Resource-Rich Devices*: Abbreviated as *rich devices*, these are computationally capable devices that are able to carry the monitoring load. Examples of these devices include IED's and remote terminal units (RTU's).
- *Resource-Limited Devices*: Abbreviated as *limited* devices, these devices do not have the computational capability to monitor its own processes and must send their call lists to a rich device. Simple sensors and logging devices are examples of limited devices, such as phase measurement units and smart meters.

- *Registry*: The distributed database that holds which rich devices are within the network. A complete copy of this database is shared by all of the rich devices.
- *Peer Server*: The peer server is the default server which rich devices synchronize with by default.
- *Parent*: Every limited device (child) chooses a suitable rich device (parent) so that they send their call lists to their parent, to have them analyzed for an anomaly.
- *Parent Discovery*: Refers to the procedure of finding a suitable parent.
- *Synchronization*: Refers to the synchronization procedure of *rich* devices.
- *Merge*: In the case of partitioning, two or more networks are merged back together the network problem is solved.

The actions to be taken within the network are as follows:

Peer Discovery refers to the mechanism that enables rich and limited devices to be aware of its surrounding peers. It is the collection of the two procedures used by rich and limited devices respectively: *synchronization* and *parent discovery*. The objective of the former is for a rich device to join the rich device network so that limited devices may discover and choose it as its parent, and for the latter, to find a parent to send the call lists.

Algorithm 4 describes the selection of a suitable peer for synchronization. A rich device needs to be synchronized with the rest of the network before it can operate PowerWatch's logic, hence synchronization continues as an endless loop until its successful, as can be seen on line 2. The device attempts to synchronize with the peer server with lines 3 and 4. On line 5, the procedure is terminated if synchronization were successful. If not, the device attempts to obtain a list of rich devices that are

Algorithm 4: Synchronization for Rich Devices

```
   input  : none
   output: successful synchronization
 1 begin
 2     while no synchronization do
 3         peer_server ← default_peer_server
 4         synchronize(peer_server)
 5         if synchronization successful then
 6             return
 7         local_rich_peer_list ← discoveryBroadcast()
 8         rich_peer ← randomly select a peer from peer list synchronize(rich_peer)
 9         if synchronization successful then
10             return
11         exponentialBackoff()
```

within the same broadcast domain on line 7, randomly selects one of them with line 8, and attempts to synchronize with it on line 9, terminating if the synchronization is successful on line 10. In case the device is still not synchronized with the Power-Watch deployment, it waits for a period that is exponentially increasing with each failed synchronization attempt, as shown with line 11.

The exponential backoff algorithm is the standard algorithm for waiting for a predefined amount of time after failure until the attempted operation succeeds. After each failed operation, the device waits for a time period that increases exponentially with respect to the number of failures. For this study, it is recommended to wait for 2^n seconds where n is the total number of failures. For example, a device failing to connect for the 3rd, 4th, and 5th times would wait for 8, 16, and 32 seconds, respectively, before attempting for connection again. The maximum value for n is also recommended to be 7 $(2^7 = 128)$, which limits the maximum waiting time to a little more than two minutes.

Algorithm 5 describes the procedure for the selection of a suitable parent. Limited devices first seek a parent within the same broadcast domain as choosing the

Algorithm 5: Parent Discovery for Limited Devices

```
   input  : none
   output: chosen parent
1  begin
2      while a parent is not found do
3          rich_peer_list ← initializeEmptyList()
4          rich_peer_list.append(localRichDiscovery())
5          rich_peer_list.append(localLimitedParentDiscovery())
6          rich_peer_list.append(peerListFromDefaultServer())
7          for i ← 0 to rich_peer_list.length do
8              rich_peer ← rich_peer_list[i]
9              chooseParent(rich_peer)
10             if parent choosing successful then
11                 return
12         exponentialBackoff()
13     return parent
```

closest parent would put the smallest overhead to the network. If no parent could be found, the device then asks the parents of limited devices within the same broadcast domain. If this also fails, it attempts to communicate with the default peer server. The limited device exponentially backoffs before attempting to begin the same procedure again.

Algorithm 5 describes the procedure for the selection of a suitable parent. The device first collects the list of suitable parents in three sources: from devices within the broadcast domain, the parents of the limited devices within the broadcast domain, and finally, from the peer list obtained from the default server, as seen in lines from 4 through 6. From line 7 to 11, the device goes through the rich device list in the order they are obtained, and attempts to choose one as a parent. If the selection is successful, the procedure terminates, and the device begins sending the call lists to the parent device. In case to selection happens, the device idles as can be seen on line 12, for a time period that is exponentially increasing with every idling period, similar to the peer discovery procedure.

Synchronization specifically refers to the process that a new rich device inheriting the common database. The principal purpose of this database is to serve as a registry for rich devices. The specifics of this distributed database is left to the implementation, though in our tests we used *rqlite*, a distributed version of *sqlite* that uses Raft consensus protocol [OO14], which is found in most common distributed databases.

Merging Partitions is a need arises when a distributed system is divided due to network error and start to operate as two or more independent systems. After the error is fixed, the independent clusters need to be joined together to operate as one again: this process is named as *partition merging*.

Merger operation is rather straightforward: as the unreachable rich peers are eliminated from the registry, the remaining records in both clusters are simply joined together.

CHAPTER 6

EVALUATION OF POWERWATCH

Evaluation of the trace analysis framework was conducted in two parts: the first part is based on a state-machine-based model of smart grid devices. This scheme allowed us to test the framework over a total of 37500 experiments collectively representing different device and attack complexities that may appear during the operation of smart grid devices. Using data resulted from the state-machines, we also determined the optimal values for bucket and window sizes. We further implemented a realistic smart grid testbed conforming to IEC61850 standards and compromised a device in three distinct ways while attempting to detect it.

6.1 Methodology of State Machine-Based Experiments

The evaluation of the framework's performance was conducted on the finite state machine model of smart grid devices, as shown in Figure 3.2. The model machine is defined in more depth to allow for experimentation: It is comprised of an initial and a halting state, a bounded amount of states that were marked as benign, and optionally, a state that was marked as malicious.

The principles of the state machine operation are stated as follows: The machine starts at the initial (or idle) state. At each turn, the machine randomly transitions to any state, except the initial and halting state, with respect to the state transition probabilities and outputs a call list assigned to that state. After a certain amount of turns (detailed below), the machine transitions into the halting state, and halts without outputting any call lists. The benign states had a cumulative state transition probability of 99%, and the malicious state, if present, had the transition probability 1%. For example, if there were 4 benign states and 1 malicious state, each benign state has a transition probability of $\frac{99}{100} \cdot \frac{1}{4}$. Furthermore, if the malicious

state is present, it is ensured that the malicious state has been transitioned to at least once.

In each experiment, a total number of 45 runs were conducted: 15 runs for a benign machine the results of which were used for training the detector, additional 15 runs for the benign machine, the results of which were used solely for testing the detector, and final 15 runs for the malicious state machine, namely a state machine that contained all the states of the benign machine, in addition to a malicious state. Each run took a total of 500 turns, with a total turn count of $500 \times 45 = 22500$ turns.

The state machines were randomly generated with respect to 5 parameters which model a big portion of all possible device and attack complexity that may be realized during the operation of the smart grid devices and are described as follows:

- *Benign state count*: Represents how many uniquely identifiable computations a device may conduct in a non-compromised situation. The benign state machine inhibits as many states as this parameter indicates.
- *Benign state call count*: Represents how many calls a benign state may make. This parameter is sampled separately for every benign state.
- *Benign state call dictionary*: Represents the diversity in types of calls a benign state may make, e.g., a benign state call dictionary value of 3 indicates a benign state makes only 3 different types of calls. Note that this value was an upper bound: the unique number of calls may be lower than that since the list itself was also randomly generated.
- *Malicious state call count*: The same as its benign counterpart, but it was used solely for the malicious state.

- *Malicious state call dictionary*: Again, the same as its benign counterpart, but it was used solely for the malicious state.

Table 6.1 describes the parameters used in generating the state machine. Each experiment receives the range of every parameter which it should sample from to generate the state machine. For every possible combination of parameter ranges, an experiment was run. The total number of experiments conducted was 37500.

Table 6.1: Parameters for State Machine Generation

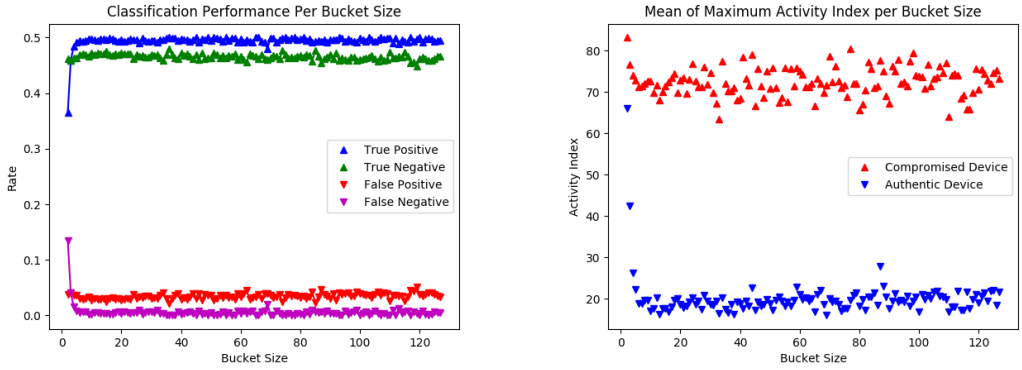
Parameter name	Ranges
Benign state count	[1, 5) [5, 10) [10, 15) [15, 20) [20, 25)
Benign state call count	[1, 5) [5, 10) [10, 15) [15, 20) [20, 25)
Bucket size	[2, 8) [8, 32) [32, 128)
Window size	[20, 40) [60, 80) [90, 100) [150, 200)
Benign state call vocabulary	[1, 5) [5, 10) [10, 15) [15, 20) [20, 25)
Malicious state call count	[1, 5) [5, 10) [10, 15) [15, 20) [20, 25)
Malicious state call vocabulary	[1, 5) [5, 10) [10, 15) [15, 20) [20, 25)

6.2 Results

This subsection is dedicated to interpreting the data obtained from state-machine-based smart grid devices. There are two questions to be answered: can the framework successfully detect compromised devices, and if so, what are the best values for bucket and window sizes?

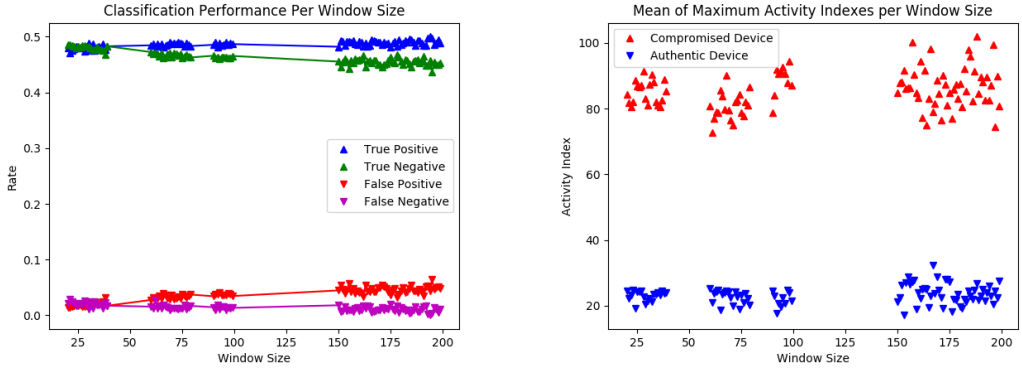
To assess whether the framework can accomplish its task, Figures 6.1(a) and 6.2(a) must be examined. In both cases, the framework consistently produces true positive and true negative values with low false positive and false negatives, which implies that the framework, indeed, can detect the compromised devices.

Figure 6.1 presents a collection of figures that show the performance of PowerWatch the by bucket size. Figure 6.1(b) clearly illustrates that for each bucket size



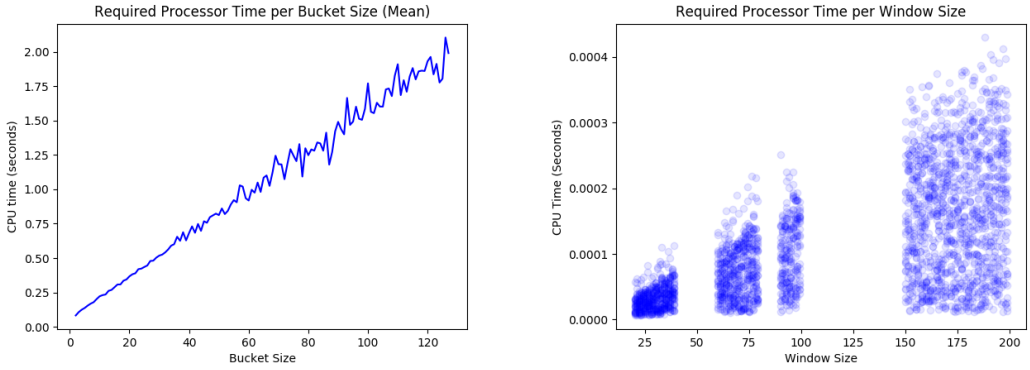
(a) (b)

Figure 6.1: PowerWatch's performance per bucket size.



(a) (b)

Figure 6.2: PowerWatch's performance per window size.



(a) (b)

Figure 6.3: Required processor times per bucket and window sizes.

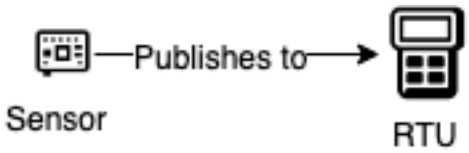


Figure 6.4: Topology of the testbed which attacks were conducted.

value, the framework produced higher activity signal peaks for the compromised devices (higher activity index implies compromise). Figure 6.1(a) presents the classification performance of the framework: the detection rates stay consistently very high for all the bucket size values except abnormally low ones. Figure 6.3(a) required computation power scales linearly with the bucket size. Since bigger bucket sizes require more time, and the detection rate does not change by bucket size except for very low values, the lowest possible bucket size that does not yield inferior performance must be selected, which in this case, we recommend the value of 10.

On the other hand, Figure 6.2 contains a collection of figures that present the performance of the framework by the window size. Figure 6.2(b) indicates that for every tested window size, the framework produced higher activity signal peaks for compromised devices. Figure 6.2(a) presents the classification performance: the detection rates slightly tank as the window size grows. Figure 6.3(b) shows the required computation power scales linearly with the window size in the worst case; though, even at its worst, it is negligibly low. Since higher window sizes imply inferior detection rate and increased computation power, a low window size value must be chosen: in this case, we recommend the value of 25.

6.3 Detecting Attacks in a Real Testbed

The state-machine-based testing and evaluation have allowed us to efficiently enumerate and run our framework against a wide variety of cases that can possibly

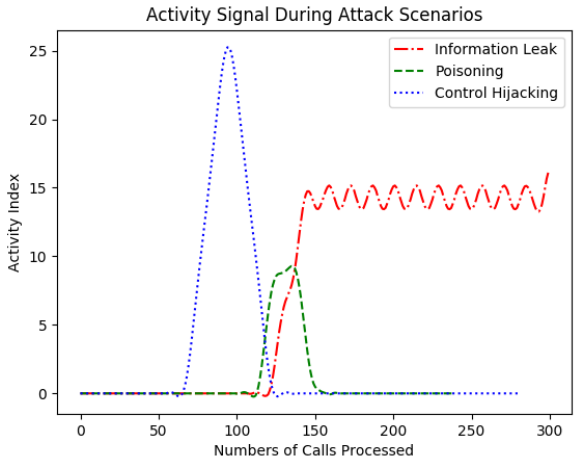


Figure 6.5: Activity signals generated by devices under various attacks.

appear during the operation of smart grid devices. We now present how well the framework performs under concrete attack scenarios.

This assessment was done under a representative testbed of smart grid devices which comprised of two devices: a command-and-control (C&C) device, and a measurement device in communication using GOOSE messages. An illustration of this setup is given in Figure 6.4. These devices were programmed so that the measurement unit supplied data to the (C&C) server, implemented by using the open-source *libiec61850* library which provided automatic IEC61850 conformance. For the command-and-control unit, a control hijacking attack was implemented. The measurement unit was compromised in two distinct ways: In the first case, the attacker altered the measurements, and in the second case, the attacker sent the measurement values to an external server, leaking the sensitive and valuable measurement information to outsiders.

The obtained results are shown in Figure 6.5 for both C&C and measurement units. In each figure, its activity indexes in both uncompromised and compromised states are given. As expected, when the device is not compromised, its activity

Figure 6.6: For overhead analysis, two rich devices were monitoring 40 limited devices.

index stays zero at all times. Compromising both C&C and measurement units in each attack scenario produced an activity index that is significantly higher than zero, indicating PowerWatch is successful in detecting compromised devices.

6.4 Overhead Analysis

The trace analysis algorithm is expected to increase both the computational load of the devices and the network utilization. It is necessary to quantify this increase in order to determine whether the algorithm is feasible to deploy. The overhead analysis is conducted in the decentralized mode as its results can be generalized into both stand-alone and centralized mode.

The illustration of the device topology for overhead analysis is given in Figure 6.6, where two rich devices were monitoring a grand total of 40 limited devices, each individually monitoring 20 devices. Both rich and limited devices were run under a container using Docker [Ber14]. The rich devices were allowed to use 2 CPU cores to its full extent, while limited devices were allowed to use 1 CPU core at 2% capacity. Both rich and limited devices were running a program implemented using *libiec61850*, with limited devices acting as a sensor unit, and publishing data

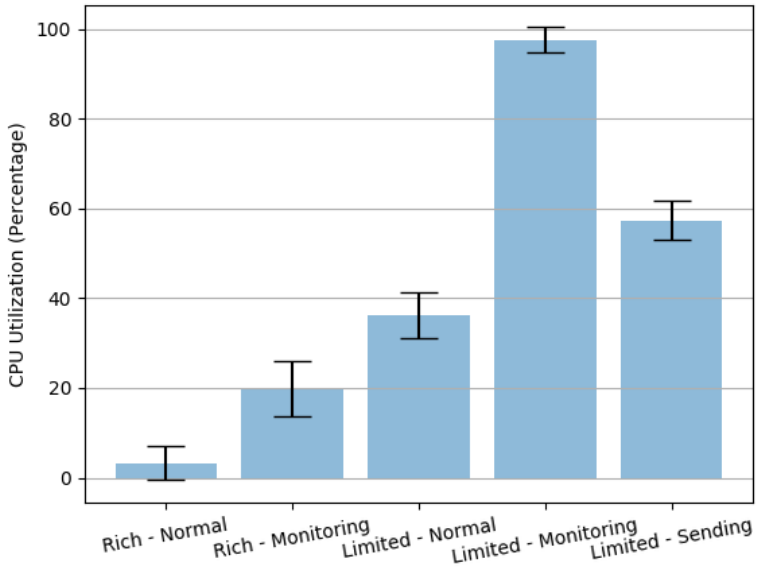


Figure 6.7: CPU utilization with respect to device and its state.

to one of the rich devices. Note that, in this test, limited devices send both GOOSE messages, which is related to the grid operations, and call list traces for monitoring.

6.4.1 CPU Overhead

Figure 6.7 summarizes the CPU usage percentages, sampled at one-second interval of the devices during their various configurations, explained below:

- *Rich - Normal*: Rich devices during their normal operation (no monitoring),
- *Rich - Monitoring*: Rich devices during monitoring 20 devices,
- *Limited - Normal*: Limited devices during their normal operation,
- *Limited - Monitoring*: Limited devices while they are monitoring themselves,
- *Limited - Sending*: Limited devices while they are sending call traces to rich devices.

From the figure, it can be seen that rich devices almost have no load when not monitoring, and have $\tilde{2}0\%$ CPU utilization when monitoring limited devices, which implies the device can easily monitor devices in addition to its normal duties.

Similarly, in the case of limited devices, their load on normal operations were measured to be around $\tilde{4}0\%$ utilization. When attempting to monitor themselves, the operation of its normal duties was disrupted, and unprocessed call lists were piling up. However, sending the call lists to a remote host induced an acceptable overhead while enabling the device to resume its duties without disruption.

6.4.2 Network Overhead

Network overhead strictly depends on the rate of calls that are being generated by the grid devices. In the implemented testbed, each device was producing approximately 6 calls per second. The application layer data size was calculated as 61 bytes. Multiplying it with 40 by device number yielded 2440 Byte/s. Considering the network infrastructure is comprised of gigabit ethernet, the proposed scheme consumes $1.95 \times 10^{-5}\%$ of the total bandwidth, which is a minuscule number. This figure shows a commodity network equipment can handle $10^5 = 10000$ devices until the network usage (approximately 2%) is noticeable.

CHAPTER 7

CONCLUSION

The transformation of smart grid to include digital devices has introduced the compromised devices problem, where an adversary may perform arbitrary modification on a device. We proposed to detect such devices by means of anomaly detection using system and library call lists. We have formally analyzed the capability of call list analysis for this task, proposed a novel algorithm that measures call list similarity, devised a Monte-Carlo experiment to assess how it behaves in different circumstances and implemented it to work in a real smart grid testbed. Our results show that the proposed algorithm is successful in detecting anomalies in call list patterns and also succeeded in detecting compromised devices in a concrete setting. Lastly, we described how the proposed method could be deployed in three different setups: stand-alone, where the detection algorithm is operated on the device, centralized, where the call lists are processed in a centralized server, and decentralized, where surrounding devices with high computation capacity monitors the devices with low computation capacity.

The vision of this thesis was to achieve a generalizable detection method for monitoring smart grids, and the experimental data implies this is accomplished with a very low false-positive ratio. On a theoretical basis, the detection potential of the methods presented here are superior to what has been reported in the literature, since it encompasses a wide variety of vulnerabilities instead of addressing individual security problems. The investigation of the impact of this theoretical reasoning into the practice can be accomplished by directly applying the methods of this thesis to the vulnerabilities that have been reported in the literature. Doing so requires a significant engineering effort, however, and it is the reason of its omission as we have

decided focus on the development of the method and its associated experiments.

7.1 Future Work

The concepts introduced within this study can be expanded by the following:

- *Utilizing the parameters of system and library calls*: Within the scope of this study, we considered only the name of the system or library call, and omitted the parameters. Even though the results are satisfactory, the usage of parameters may open new avenues in developing a better detector.
- *Data Sources*: PowerWatch is a data-driven framework. We utilized system and library calls to conduct detection; however, the data sources that could be utilized are not limited to these: any sequential data that is deterministically generated by the program execution logic may be used as a data source. A few examples of such data is given below:
  - `jmp` *instructions*: On the assembly level, system and library calls are implemented by changing the instruction pointer by either an interrupt or a `call` instruction. The `jmp` family of instructions work in a very similar fashion, differing only in being simpler as it does not affect the call stack or processor registers. Tracing jumped address values may offer higher resolution in tracing a program.
  - *Instruction Pointer*: The values of instruction pointers may be traced to obtain an even higher resolution data.
  - *Trace Injection*: A program may be injected with code that produces a specific symbol upon execution. The system and library calls produce a symbol only upon their invocation; however, the strategic placement of trace-generating code does not limit the data generation to the locations

of the system and library calls within the code; the relaxation of this constraint may be examined for positive impact on detection performance.

  - *System Resource Utilization*: The peripheral units of a device may be traced individually, e.g., by tracing the network interface and the display of the device separately. Logging discrete actions taken on these devices improve over system calls as they only contain partial information on how these devices are used exactly.

- *Static Analysis*: PowerWatch utilizes a dynamic analysis method - it *runs* the program - to obtain the call list patterns. A static analysis approach may be used to obtain a *formal grammar* of the call lists, which then could be used to analyze the call list traces.

BIBLIOGRAPHY

[BAU17] L. Babun, H. Aksu, and A. S. Uluagac. Identifying counterfeit smart grid devices: A lightweight system level framework. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–6, May 2017.

[Ber14] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[BS11] R. Berthier and W. H. Sanders. Specification-based intrusion detection for advanced metering infrastructures. In *IEEE 17th Pacific Rim International Symposium on Dependable Computing*, pages 184–193, Dec 2011.

[CBLM17] K. Candelario, C. Booth, A. S. Leger, and S. J. Matthews. Investigating a raspberry pi cluster for detecting anomalies in the smart grid. In *2017 IEEE MIT Undergraduate Research Technology Conference (URTC)*, pages 1–4, Nov 2017.

[CLY+18] Hwei-Ming Chung, Wen-Tai Li, Chau Yuen, Wei-Ho Chung, Yan Zhang, and Chao-Kai Wen. Local cyber-physical attack for masking line outage and topology attack in smart grid. *IEEE Transactions on Smart Grid*, 2018.

[EAA+04] Mark Endrei, Jenny Ang, Ali Arsanjani, Sook Chua, Philippe Comte, Pål Krogdahl, Min Luo, and Tony Newling. *Patterns: service-oriented architecture and web services*. IBM Corporation, International Technical Support Organization, 2004.

[Far10] Hassan Farhangi. The path of the smart grid. *IEEE power and energy magazine*, 8(1):18–28, 2010.

[FMXY12] X. Fang, S. Misra, G. Xue, and D. Yang. Smart grid – the new and improved power grid: A survey. *IEEE Communications Surveys Tutorials*, 14(4):944–980, Fourth 2012.

[GBG+11] A. Giani, E. Bitar, M. Garcia, M. McQueen, P. Khargonekar, and K. Poolla. Smart grid data integrity attacks: characterizations and countermeasures. In *IEEE International Conference on Smart Grid Communications (SmartGridComm)*, pages 232–237, Oct 2011.

[HLG14] J. Hong, C. Liu, and M. Govindarasu. Integrated anomaly detection for cyber security of the substations. *IEEE Transactions on Smart Grid*, 5(4):1643–1653, July 2014.

[KH97] A. P. Kosoresow and S. A. Hofmeyer. Intrusion detection via system call traces. *IEEE Software*, 14(5):35–42, Sep. 1997.

[Kos16] A. M. Kosek. Contextual anomaly detection for cyber-physical security in smart grids based on an artificial neural network model. In *Joint Workshop on Cyber- Physical Security and Resilience in Smart Grids (CPSR-SG)*, pages 1–6, April 2016.

[KP11] T. T. Kim and H. V. Poor. Strategic protection against data injection attacks on power grids. *IEEE Transactions on Smart Grid*, 2(2):326–333, June 2011.

[KT13] J. Kim and L. Tong. On topology attack of a smart grid: Undetectable attacks and countermeasures. *IEEE Journal on Selected Areas in Communications*, 31(7):1294–1305, July 2013.

[LBAU17] Juan Lopez, Leonardo Babun, Hidayet Aksu, and A. Selcuk Uluagac. A survey on function and system call hooking approaches. *Journal of Hardware and Systems Security*, 1(2):114–136, Jun 2017.

[LCZ+14] Shan Liu, Bo Chen, Takis Zourntos, Deepa Kundur, and Karen Butler-Purry. A coordinated multi-switch attack for cascading failures in smart grid. *IEEE Transactions on Smart Grid*, 5(3):1183–1195, 2014.

[LED+14] L. Liu, M. Esmalifalak, Q. Ding, V. A. Emesih, and Z. Han. Detecting false data injection attacks on power grid by sparse optimization. *IEEE Transactions on Smart Grid*, 5(2):612–621, March 2014.

[LJL+10] Thaddeus D Ladd, Fedor Jelezko, Raymond Laflamme, Yasunobu Nakamura, Christopher Monroe, and Jeremy Lloyd OBrien. Quantum computers. *Nature*, 464(7285):45, 2010.

[LZL+17] G. Liang, J. Zhao, F. Luo, S. R. Weller, and Z. Y. Dong. A review of false data injection attacks against modern power systems. *IEEE Transactions on Smart Grid*, 8(4):1630–1638, July 2017.

[MCHL14] K. Manandhar, X. Cao, F. Hu, and Y. Liu. Detection of faults and attacks including false data injection attack in smart grid using kalman

filter. *IEEE Transactions on Control of Network Systems*, 1(4):370–379, Dec 2014.

[MKB+12] Y. Mo, T. H. J. Kim, K. Brancik, D. Dickinson, H. Lee, A. Perrig, and B. Sinopoli. Cyber-physical security of a smart grid infrastructure. *Proceedings of the IEEE*, 100(1):195–209, Jan 2012.

[ML18] S. Matthews and A. St. Leger. Leveraging mapreduce and synchrophasors for real-time anomaly detection in the smart grid. *IEEE Transactions on Emerging Topics in Computing*, pages 1–1, 2018.

[MW18] R. Moghaddass and J. Wang. A hierarchical framework for smart grid anomaly detection using large-scale smart meter data. *IEEE Transactions on Smart Grid*, pages 1–1, 2018.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[Sch15] Jrgen Schmidhuber. Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117, 2015.

[SG14] S. Sridhar and M. Govindarasu. Model-based attack detection and mitigation for automatic generation control. *IEEE Transactions on Smart Grid*, 5(2):580–591, March 2014.

[SGLL13] Y. Sun, X. Guan, T. Liu, and Y. Liu. A cyber-physical monitoring system for attack detection in smart grid. In *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 33–34, April 2013.

[TBAC11] K. Turitsyn, S. Backhaus, M. Ananyev, and M. Chertkov. Smart finite state devices: A modeling framework for demand response technologies. In *50th IEEE Conference on Decision and Control and European Control Conference*, pages 7–14, Dec 2011.

[TSS+18] Song Tan, Wen-Zhan Song, Michael Stewart, Junjie Yang, and Lang Tong. Online data integrity attacks against real-time electrical market in smart grid. *IEEE Transactions on Smart Grid*, 9(1):313–322, 2018.

[WL13] Wenye Wang and Zhuo Lu. Cyber security in the smart grid: Survey and challenges. *Computer Networks*, 57(5):1344 – 1371, 2013.

[WLX+17] F. Wang, Q. Liu, F. Xiong, L. Guo, J. Feng, and Q. Wang. Data validation and anomaly detection techniques for smart substations. In *2017 IEEE Conference on Energy Internet and Energy System Integration (EI2)*, pages 1–6, Nov 2017.

[YQST13] Y. Yan, Y. Qian, H. Sharif, and D. Tipper. A survey on smart grid communication infrastructures: Motivations, requirements and challenges. *IEEE Communications Surveys Tutorials*, 15(1):5–20, First 2013.