



**REMOTE MONITORING OF MEMORY
DATA STRUCTURES FOR MALWARE
DETECTION IN A TALOS II
ARCHITECTURE**

THESIS

Robert A. Willburn, Second Lieutenant, USAF
AFIT-ENG-MS-21-M-094

**DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY**

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

The views expressed in this document are those of the author and do not reflect the official policy or position of the United States Air Force, the United States Department of Defense or the United States Government. This material is declared a work of the U.S. Government and is not subject to copyright protection in the United States.

AFIT-ENG-MS-21-M-094

REMOTE MONITORING OF MEMORY DATA STRUCTURES FOR MALWARE
DETECTION IN A TALOS II ARCHITECTURE

THESIS

Presented to the Faculty
Department of Electrical and Computer Engineering
Graduate School of Engineering and Management
Air Force Institute of Technology
Air University
Air Education and Training Command
in Partial Fulfillment of the Requirements for the
Degree of Master of Science in Computer Science

Robert A. Willburn, B.S.

Second Lieutenant, USAF

March 2021

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-21-M-094

REMOTE MONITORING OF MEMORY DATA STRUCTURES FOR MALWARE
DETECTION IN A TALOS II ARCHITECTURE

THESIS

Robert A. Willburn, B.S.
Second Lieutenant, USAF

Committee Membership:

Scott R. Graham, Ph.D.
Chair

Barry E. Mullins, Ph.D., P.E.
Member

Stephen J. Dunlap, M.S.
Member

Abstract

New forms of malware, namely fileless malware and rootkits, pose a threat to traditional anti-malware. Rootkits in particular have the capacity to obscure the present state of memory from the user space of a target machine. If this happens, anti-malware running in the user space of an affected machine cannot be trusted to operate properly.

To combat this threat, this research proposes the remote monitoring of memory from a second, secure machine. This research uses the Talos II platform. It assumes a secure processor running OpenBMC serving as a baseboard management controller for a POWER9 processor, which is assumed to be vulnerable to exploitation. Although the baseboard management controller is hardened against exploitation, and therefore restricted in its functionality, it includes an application called *pdbg*, used for debugging POWER9 processors. This application allows for both reading and writing to registers and system memory of the POWER9 processor from the baseboard management controller directly via the field replaceable unit support interface bus.

This research developed a program to run on the baseboard management controller which utilizes *pdbg* to traverse the process tree active in the memory of the POWER9 processor. By traversing this data structure, it demonstrates the capability to view the entire process tree remotely, thereby providing a check on whether information in memory is being hidden from user space on the POWER9.

Acknowledgements

I would like to thank my advisor Dr. Graham for his advice in every stage of the research process, from experimental design, to writing and review.

I would also like to thank my committee members, Mr. Stephen Dunlap for his providing insight into the architecture used in the research and experimental design, and Dr. Barry Mullins for his flexibility in the review process.

Robert A. Willburn

Table of Contents

	Page
Abstract	iv
Acknowledgements	v
List of Figures	ix
List of Tables	xi
List of Acronyms	xii
I. Introduction	1
1.1 Background and Motivation	1
1.2 Problem Statement	2
1.3 Research Objectives	2
1.4 Hypothesis	3
1.5 Approach	3
1.6 Contributions	3
1.7 Organization	4
II. Background and Related Work	5
2.1 Overview	5
2.2 Anti-malware	5
2.3 Servers	6
2.4 Talos II Architecture	7
2.5 On Chip Controller (OCC)	9
2.6 <i>pdbg</i>	9
2.7 Fileless Malware	9
2.8 Rootkit and Kernel Modules	10
2.9 Process Management in Linux	12
2.10 Endianness	13
2.11 Summary	14
III. Case Study	15
3.1 Overview	15
3.2 Motivation	15
3.3 Use Case	15
3.4 Limitations	16
3.5 Requirements and Assumptions	17
3.6 Case Study Methodology:	17
3.6.1 Process Tree	17

	Page
3.6.2 Creating the Program	20
3.6.3 Testing	21
3.7 Conclusion	26
3.8 Summary	27
IV. Experimental Scenarios	28
4.1 Overview	28
4.2 Experimental Objective	28
4.3 Design	29
4.4 Assumptions	29
4.4.1 Control Variables	30
4.4.2 Independent Variables	30
4.4.3 Dependant Variables	30
4.5 Application Scenarios	31
4.5.1 Scenario 1	31
4.5.2 Scenario 2	31
4.6 Performance Evaluation	32
4.6.1 Scenario 1	32
4.6.2 Scenario 2	44
4.7 Summary	50
4.7.1 Scenario 1	50
4.7.2 Scenario 2	51
V. Observations and Analysis	52
5.1 Overview	52
5.2 Analysis	52
5.2.1 Single Read	52
5.2.2 Variable reads	54
5.2.3 Proposed Equation	55
5.3 Application Scenario - Feasibility of Detecting Malware Processes Using <i>pdbg</i>	56
5.4 Memory Read Using a Microcontroller	58
5.5 Summary	59
VI. Conclusion	61
6.1 Overview	61
6.2 Summary	61
6.3 Contributions	63
6.4 Future Work	63
6.5 Conclusion	64

	Page
Appendix A. Process Tree Traversal Script	65
Bibliography	71

List of Figures

Figure		Page
1	Talos II Architecture	8
2	Process Tree Structure	13
3	Processes Retrieved from P9 Memory	22
4	Latency of <i>pdbg</i> Reads from 8 to 800 Bytes	34
5	Standard Deviation and Ratio to Average Duration for Trials Between 8 and 800 Bytes	35
6	Latency of <i>pdbg</i> Reads from 808 to 1016 Bytes	36
7	Standard Deviation and Ratio to Average Duration for Trials Between 808 and 1016 Bytes	37
8	Latency of <i>pdbg</i> Reads from 1152 to 3584 Bytes	38
9	Standard Deviation and Ratio to Average Duration for Trials Between 1152 and 3584 Bytes	39
10	Latency of <i>pdbg</i> Reads from 8 to 3584 Bytes	40
11	Standard Deviation and Ratio to Average Duration for Trials Between 8 and 3584 Bytes	41
12	Latency of <i>pdbg</i> Reads from 4096 to 524288 Bytes	42
13	Standard Deviation of Trials Between 4096 to 524288 Bytes	43
14	Ratio of Standard Deviation to Average Duration for Trials Between 4096 to 524288 Bytes	43
15	Variable Reads Results 128 and 256 Bytes	46
16	Variable Reads Results 1024 - 4096 Bytes	47
17	Standard Deviation and Ratio for Variable Reads at Data Read Size 128	48
18	Standard Deviation and Ratio for Variable Reads at Data Read Size 256	48

Figure		Page
19	Standard Deviation and Ratio for Variable Reads at Data Read Size 1024	49
20	Standard Deviation and Ratio for Variable Reads at Data Read Size 2048	49
21	Standard Deviation and Ratio for Variable Reads at Data Read Size 4096	50

List of Tables

Table		Page
1	Baseline Process Test	23
2	Dummy Processes	24
3	Malware Detection	26
4	Variable Number of Reads for 128 Bytes	44
5	Variable Number of Reads for 256 Bytes	45
6	Variable Number of Reads for 1024 Bytes	45
7	Variable Number of Reads for 2048 Bytes	45
8	Variable Number of Reads for 4096 Bytes	46

List of Acronyms

BIOS	Basic Input/Output System
BMC	Baseboard Management Controller
DHCP	Domain Host Control Protocol
DNS	Domain Name System
FSI	Flexible Service Interface
GDB	GNU Debugger
GPT	GUID Partition Table
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IPMC	Intelligent Platform Management Controller
LKM	Loadable Kernel Module
MBR	Master Boot Record
OCC	On Chip Controller
OS	Operating System
PCB	Process Control Block
PID	Process ID
TGID	Thread Group ID
UEFI	Unified Extensible Firmware Interface

REMOTE MONITORING OF MEMORY DATA STRUCTURES FOR MALWARE DETECTION IN A TALOS II ARCHITECTURE

I. Introduction

1.1 Background and Motivation

Malware is a persistent problem for cyber security professionals, and anti-malware software plays a prominent role in combating its spread. Anti-malware solutions typically work by analyzing the signatures of entries in the filesystem. New forms of malware, such as fileless malware, present challenges to this model. Fileless malware operates by loading a script directly to memory without ever writing to disk, thereby frustrating anti-malware's ability to detect this class of malware [1]. In response, companies have implemented memory scanning capabilities into anti-malware software [2]. Additionally, the proper functioning of anti-malware software is threatened by rootkits, which are malware that gain root permissions to a system. Once root permissions are gained, the malware can compromise the kernel via a Loadable Kernel Module (LKM) [3]. An LKM can compromise the ability of anti-malware software to detect malware in memory or in the file system. Specifically, an LKM can alter kernel data structures. For example, it could overwrite a function that lists running processes such that the malware processes are omitted from the list.

One potential way to mitigate the ability of a rootkit to hide information is to gain direct access to memory resources from an external entity that can be trusted. Given a case where a rootkit is able to hide information in memory from an operating system it has compromised, circumventing this compromised operating system entirely could

provide a direct look into what actually exists in memory. Comparing this information to information gathered from the user space of the potentially compromised machine could expose that the target machine was compromised.

1.2 Problem Statement

Can direct access to memory provide a means to check for the ability of anti-malware software to fully access data in memory? The Talos II architecture provides for the direct access of a Baseboard Management Controller (BMC) to the memory of a POWER9 processor on the rack via the Flexible Service Interface (FSI) bus. This method of direct external access to memory provides an opportunity to remotely view the memory contents of a potentially compromised system. By exploring the process tree data structure in memory, this research measures the possibility and practicality of using remote memory access to augment the operation of anti-malware software.

1.3 Research Objectives

As part of this effort, the following research objectives were identified:

- Identify malware that can compromise the ability of anti-malware software to access information in memory
- Utilize *pdbg* to identify when information is being hidden by malware from the user space of a compromised machine
- Evaluate the feasibility of deploying *pdbg* for this task

The following questions were posed and answered in this research in order to satisfy the previous research objectives:

- Can directly accessing memory via *pdbg* glean sufficient information to create a meaningful contribution to anti-malware efforts?

- Can *pdbg* access memory fast enough for information gleaned to be feasibly deployed in a real-world scenario? Both of these questions were answered in the affirmative in this research.

1.4 Hypothesis

The hypothesis of this research is that *pdbg* can be used to traverse data structures in memory which will provide information that can expose when a complete view of memory is being obfuscated from user space on a potentially compromised POWER9 machine. The test case performed in Chapter III confirms this hypothesis. Bounds are placed on the probability of finding a hidden malware process in memory by Equation 10 in Chapter V.

1.5 Approach

The approach consists of the creation of a program to automate the execution of *pdbg* to read the process tree data structure. Once *pdbg* has transferred the relevant information to the BMC from the POWER9, the BMC can analyze said information for data hidden from the user space of the POWER9. This approach demonstrates both the possibility and utility of monitoring active data structures in memory for assessing the ability of anti-malware software to fully access data in memory.

1.6 Contributions

- Performance Analysis: An analysis of the operation of *pdbg* for different data sizes and numbers of invocations of *pdbg* provides a model of computational complexity. This analysis is used to predict the efficacy of memory monitoring solutions using *pdbg*, and will be of utility to any seeking to explore this topic.

- Memory scanner program: This program can be used to retrieve and reconstruct key components of the process tree of a remote machine. This approach could also be used to read the *dentry* file system tree in memory by slightly modifying the program. This program demonstrates the feasibility and utility of monitoring active data structures in memory for signs of malware.

1.7 Organization

Chapter II introduces key concepts of the current state of malware and anti-malware, details the low level operations of the Talos II architecture, and discusses the inner workings of various Linux kernel data structures. It outlines the reasons for monitoring active processes by anti-malware software, how the compromise of the kernel can frustrate efforts to do so, and how the monitoring of memory from a separate machine deemed secure counteracts these threats. It defines terminology that is used throughout the rest of the research. Chapter III details a test case whereby running processes can be directly monitored in memory. It demonstrates the ability to find all running processes using *pdbg*, including processes that have been hidden from the user space of the target machine by malware. This in turn demonstrates the feasibility of monitoring processes directly in memory. Chapter IV details experiments that measure the latency of *pdbg*. It demonstrates the trends in the time elapsed for retrieving various amounts of data from memory. Chapter V analyzes the results of the experiments carried out in Chapter IV. It provides a model by which to predict the cost of the utilization of *pdbg* in various scenarios. From this data, a piecewise linear equation was generated that predicts the time to complete various actions utilizing *pdbg*. Chapter V analyzes the feasibility the deployment of the solution proposed in Chapter III in light of the findings of Chapter IV.

II. Background and Related Work

2.1 Overview

This chapter provides background information relevant to the state of current anti-malware solutions and the architecture used in this research.

2.2 Anti-malware

Anti-malware software is intended to detect and remove malware from a targeted system. There are different kinds of anti-malware. Anti-malware can be hosted on the client machine itself or operate remotely from a centralized administrator machine. The former model is more prevalent in consumer machines, whereas the centralized model is more often used by network administrators on a commercial network. Anti-malware solutions typically scan the filesystem for file signatures. These signatures are hash values calculated from various attributes of the file [4]. This hash can be compared against an allowlist or a disallowlist. Some systems may have special procedures for dealing with a file whose hash is neither found in the allow or disallow lists. Such approaches may be referred to as greylisting. Each of these represent a general approach to dealing with authorization of programs. In an allowlist model, also known as whitelist, the only files permitted to run are those that are present on the allowlist. If a file is not on the allowlist, it could be barred from execution, quarantined or deleted. This is the most restrictive approach. This will most often be applied in a commercial setting where group policy for permitted software is set from a network administrator [5]. To prevent malware from affecting workplace systems, employees are often restricted from installing new software on their work machines. Allowlists are one way this kind of policy is enforced. In the disallowlist model, also known as blacklist, file signatures are compared to those of known malware. If

a file signature matches a known malware signature, then it would be barred from execution, quarantined or deleted. This is the most permissive approach. It is most common in personal machines. In a greylist model, which is a hybrid between the previous two, signatures which are not explicitly disallowed, nor explicitly allowed, may be subject to additional measures or scrutiny. Perhaps they would be temporarily quarantined, or perhaps only allowed execution in a sandbox. They could be sent to a third party for additional examination before being executed, or perhaps require user or administrator intervention.

2.3 Servers

Servers comprise one half of the client-server model, providing specific services to client computers. Domain Host Control Protocol (DHCP), Domain Name System (DNS), and website hosting are all examples of services provided by servers. For example, when a client computer searches for a webpage, that search is translated into a request to a DNS server, which looks up the IP address associated with the domain name and returns it to the client, which then sends an Hypertext Transfer Protocol (HTTP) or Hypertext Transfer Protocol Secure (HTTPS) *GET* request to a web server with the given address, which in turn provides the client with the requested webpage. The components of modern servers are superficially similar to personal computers, but vary in scale. Servers are commonly grouped in a rack. Every server contains multiple processors, each with multiple cores. Each one of these processors runs its own Operating System (OS) and communicate with the outside world through a network connection.

A separate system called the Intelligent Platform Management Controller (IPMC) is used in some servers to manage the disparate processors from a centralized location [6]. The main processor on the IPMC is the BMC. This processor has the ability

to power on and off the various processors for which it is responsible. It can also update firmware. As long as the server is connected to a power source, the BMC remains on even if the server processors are powered off. The BMC possesses its own local IP address and can be directly accessed from an external machine over a network. Administrators often use this capability to remotely manage the server. Additionally, if a processor for which the BMC is responsible has issues with networking, it can reroute traffic through the BMC [7].

While the BMC sends the signal for a processor to turn on, it is still up to the Basic Input/Output System (BIOS) to boot the processor and then hand over control to the processor's OS. BIOS is firmware, meaning very low-level software that directly interfaces with hardware. In recent years, BIOS has begun to be replaced with Unified Extensible Firmware Interface (UEFI). UEFI is superior in that it can boot from much larger drives than BIOS. BIOS only supports the Master Boot Record (MBR) partitioning scheme. This scheme sets the limit for the number of bits available to represent the number of logical sectors at 32, which limits the size of the disk from which BIOS can boot to 2.1 TB. The GUID Partition Table (GPT) partitioning scheme, a scheme which UEFI is capable of implementing, can manage disk space up to 9.4 ZB. This additional disk space is often needed for modern servers. UEFI is implemented in a 32-bit or 64-bit architecture, whereas BIOS is implemented in a 16-bit architecture. UEFI also possesses the secure boot feature, which aims to prevent rootkit attacks [8].

2.4 Talos II Architecture

Talos II is a server architecture which uses POWER9 processors for the servers and has a BMC. In this research, the BMC runs the *openBMC* operating system while the POWER9 runs a version of the Debian operating system, with Linux kernel

version 4.18.0-rc1+. The POWER9 is a product of the *OpenPOWER* foundation, which, among other things, creates processors with open source hardware such as the POWER9 [9]. The firmware for the BMC is open source. The Talos II board, depicted in Figure 1 [10], also supports running an FSI bus from the BMC to one or more POWER9s. This bus is intended to allow for the servicing of all processors on the board through a common serial interface [11]. In Figure 1, the key components to keep in mind going forward are the BMC, the OCC, and the POWER9.

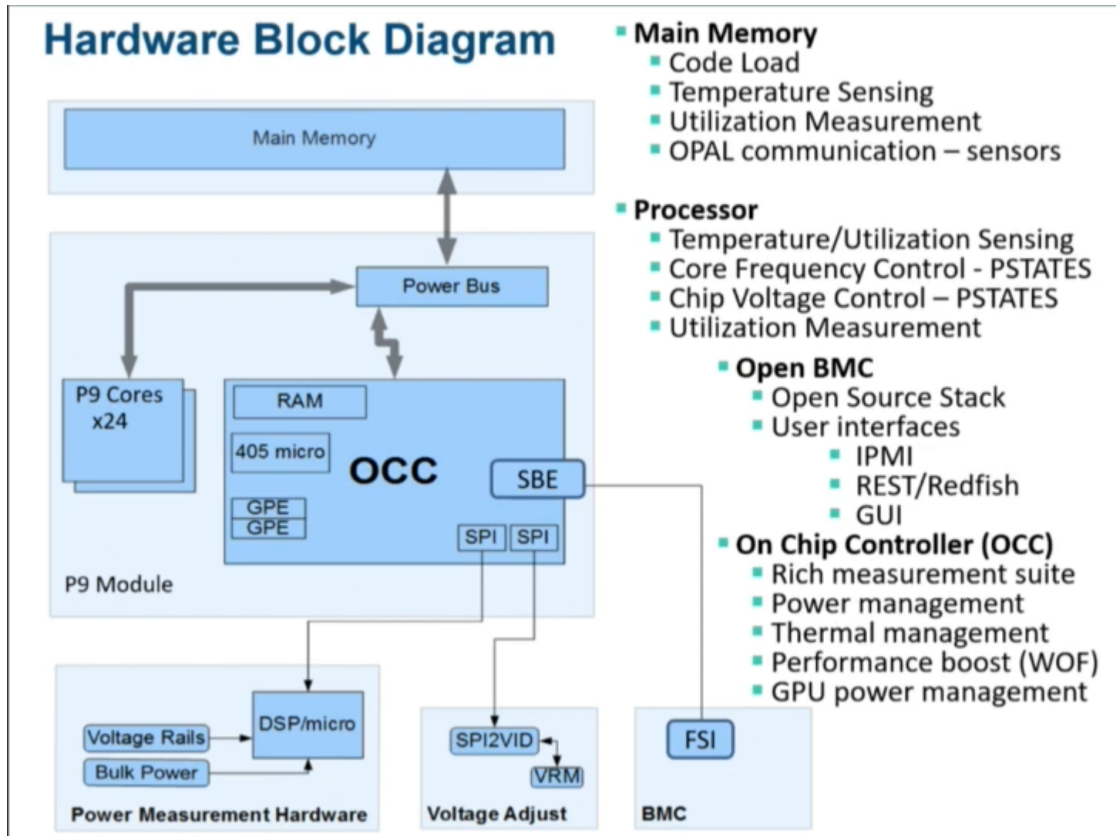


Figure 1. Talos II Architecture

2.5 OCC

The OCC is an embedded processor that monitors the operation of POWER9 processors and provides an interface for accessing memory. At set intervals, it checks the temperature at which the POWER9 is operating to ensure it does not overheat. It monitors the rate at which the POWER9 consumes energy [12]. When accessing memory, the OCC reads 8 bytes to a cache at a time from memory and subsequently returns this data over the FSI bus to the BMC

2.6 *pdbg*

pdbg is an application that runs on the BMC in a Talos II architecture. It allows for the BMC to directly access the POWER9 processor’s resources, namely various registers and system memory, using the FSI bus. This allows for the circumvention of all layers of the hierarchical protection domains of the POWER9 including the kernel. The *pdbg* application can interact with specific processors, cores and threads to execute commands. While there is only one processor on the Talos II board used in this research, *pdbg* could be used to target the resources of any of a number of processors on a board. The *getmem* option for *pdbg* can access the system memory of the selected processor using two arguments, an offset and a length in bytes. When invoked, *pdbg* will read the number of bytes from physical memory starting from the offset over the FSI bus. In order to gain access to memory, *pdbg* sends a request to the OCC to retrieve the desired memory.

2.7 Fileless Malware

Fileless malware is a class of malware that loads executable scripts directly into memory, thereby avoiding the need to write an executable file to disk. To accomplish

this, it typically utilizes programs already present on the target machine to execute its payload. A popular method for the execution of fileless malware is the invocation of *PowerShell* to execute the payload. This particular method of payload execution was utilized in the hack of the Democratic National Committee email server hack in 2016 [13]. By utilizing a trusted program on the target machine to execute the payload, the attacker is able to obscure their activity as legitimate in the eyes of anti-malware software, and avoid an automatic scan by anti-malware software that would typically be triggered by a write to disk.

2.8 Rootkit and Kernel Modules

Rootkit malware is malware that gains root permissions to a device, and often modifies the kernel itself. Rootkits are extremely dangerous as they are able to leverage their high level of permissions to hide themselves from detection by modifying system binaries. For example, the *t0rnkit* rootkit replaced the system binaries *find*, *login*, *top*, *ifconfig*, *ps*, *du*, *ls* and *netstat* [14] with its own modified versions which allowed an attacker to evade detection. The standard *ps* command enumerates running processes but the modified version would output all processes except those the attacker intended to hide. This is a blunt technique to hide processes, and there are other more subtle means to achieve the intended effect. LKMs can modify the system call table in order to execute a malicious file without replacing any existing files. In Linux, system calls allow user applications to invoke kernel functions. In order to call a kernel function, a system call looks up its address in the system call table. If a rootkit were to modify the system call table by changing the address for a function call, it would have the capability to intercept the invocation of the kernel function in favor of a function designated by the rootkit [15].

This provides a confounding scenario for anti-malware. If a rootkit were to hook

into the system call table, it could intercept the kernel calls used by anti-malware software on the machine to detect said malware. This provides an advantage over the tactic of directly modifying utility binaries because it both works on any software seeking to invoke chosen kernel functions, but also does not leave itself open to detection by changing a known utility. This can lead to detection as the hash of these files are known and could be detected by anti-malware software. Additionally, if the function redirected to by the modified system call table is loaded directly into memory and never written to disk, anti-malware software without the ability to scan memory would never discover it. This presents a dangerous scenario that both demonstrates the necessity of scanning memory and ensuring that when one is scanning memory, that program scanning the memory is not relying on potentially compromised utilities or system table.

Rootkit detection is difficult. Efforts often take the form of side channel analysis to profile the behavior of rootkits, rather than directly detecting their presence in the filesystem [16]. Detection and removal of rootkits is resource intensive and typically only initiated when cause for suspicion of infection is present.

LKM, also called external kernel modules, are files loaded into the kernel as needed. This differentiates them from internal kernel modules that are built into the kernel source tree, and are generated when building the kernel package [17]. One use for LKMs is interacting with kernel data structures without having to recompile the kernel. The Linux kernel is written in the programming language C. In order to build a module, one must have access to the configuration and header files used in building the kernel in question. Within the source code for an LKM, a function is called that loads code in the module into the kernel. The file must be compiled and loaded as root. The LKM can then be removed from the kernel with a bash command.

Because LKMs can run in the kernel without recompilation, they are an effective

form for rootkits to take. Loading code into a running kernel is much easier and less noticeable than recompiling a kernel and rebooting. LKMs do have some limitations. For instance, manipulating active data structures, such as removing a process from the process tree without killing the process while the kernel is running would be extremely challenging.

2.9 Process Management in Linux

Process Control Blocks (PCBs) are part of a data structure defined by the kernel that keeps track of a running process. In Linux, PCBs are represented by the structure *task_struct*. This structure, defined in the kernel source code, possesses a number of fields that contain information about the corresponding processes. The Thread Group ID (TGID), is functionally the same as the Process ID (PID) as seen from user space. Another field in *task_struct* is PID. When a process is created, the PID and TGID field are the same. However, when that process spawns a new thread, the new thread has a new PID but the same TGID. In fact, when the system call *getpid()* is called from user space, it is the TGID of the *task_struct* associated with the process that *getpid()* returns. Another field, *comm*, is the name of the process. The fields *children* and *siblings* are *list_head* structs that point to a list of the process' children and the list of its parent's children respectively. Each is a linearly linked list, so to traverse the PCBs of the children of a PCB, one must first traverse to the child pointed to by *children*, and then proceed down the list of PCBs pointed to by the child's *sibling* field. These data fields represent a few of the many fields available. A representation of the process tree can be found in Figure 2.

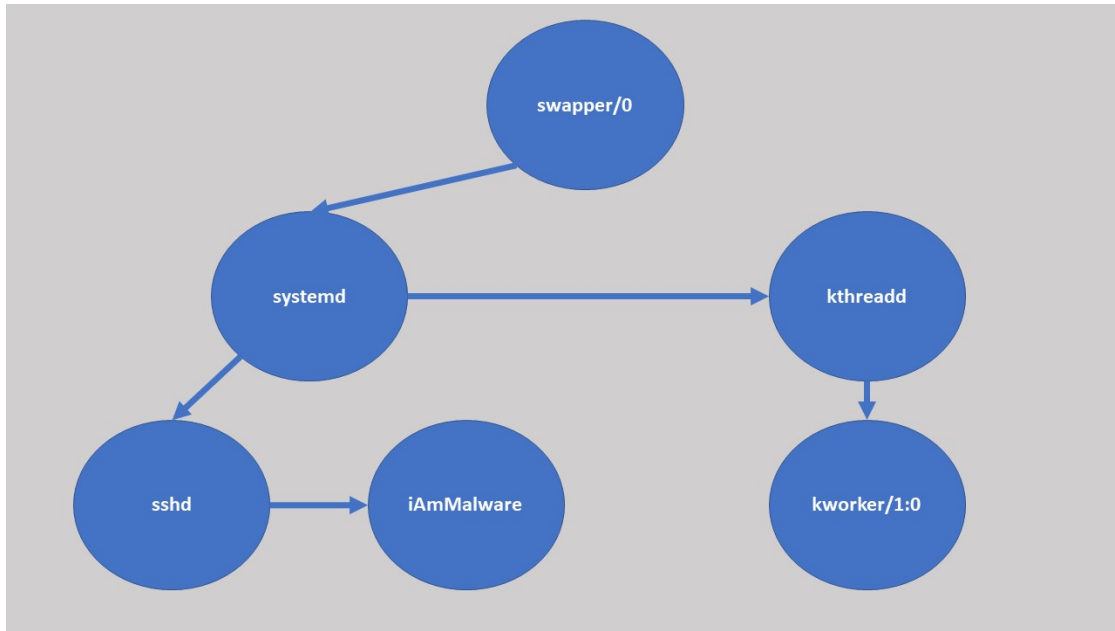


Figure 2. Process Tree Structure

Each node is a representation of a single *task_struct*. Each arrow is a link from a *task_struct* to either a child or sibling node. Each of the names of the *task_structs* in the image are processes that ran during the test case. All of these processes will be discussed in Chapter III.

2.10 Endianness

Endianness refers to the ordering of bytes in memory in a given architecture. Memory is organized into words, which are fixed sized pieces of data. In a big endian architecture, the bytes of each word are stored first byte first. In a little endian architecture, bytes are stored first byte last. Each byte is represented by two hexadecimal numbers. Hexadecimal numbers are base 16, ranging from 0 to F, where F is 15. This means each digit is equal to $X * 16^Y$, where X is the digit and Y is the digit's placement from right. Each hexadecimal character represents four bits. Together,

two hexadecimal numbers represent eight bits, or one byte. If the decimal value 100 was represented in hexadecimal value, it would be represented as 01100100 in binary, a base 2 numbering system, and 64 in hexadecimal. Sixty-four in hexadecimal translates to 100 in the regular base 10 numbering system, $4 * 16^0 + 6 * 16^1 = 100$. In a 32-bit word, with a big endian architecture the value would be stored as 00 00 00 64, and in a little endian architecture it would be stored as 64 00 00 00.

2.11 Summary

This chapter presented background concepts related to the use of direct memory inspection for malware detection. Chapter III will present the case study examining the feasibility of such an approach.

III. Case Study

3.1 Overview

This chapter will detail the process-reading test case. This test case demonstrates the ability of a program running on the BMC to retrieve and reconstruct key elements of the process tree from the memory used by the POWER9. It also demonstrates the ability to find processes hidden from the POWER9's user space by malware.

3.2 Motivation

Rootkits have the potential to obscure memory from the view of anti-malware software running in the user space of a target machine. One way it could do this is by replacing the binary for a utility, such as *ps*, with a malicious program that hides processes spawned by malware. Another technique available is hooking into the call table of the target machine in order to intercept calls by the anti-malware software to kernel functions that provide access to processes running in memory. Whatever the method used, if a malware is able to deny user space applications access to an accurate view of running processes, detection and mitigation efforts of anti-malware software running on the target machine will be fruitless. However, if a second, secure machine is able to directly access the memory used by the compromised machine, and traverse the data structure being obscured from the target machine, this information could be used to demonstrate that the target machine has its view of memory obscured. This in turn would point to likely scenario that the target machine has been compromised.

3.3 Use Case

Consider the example of a Talos II system where the BMC has the capability to read the POWER9's main memory directly using *pdbg*. If the POWER9 is compro-

mised by a rootkit, it could obscure the existence of processes spawned by malware from anti-malware running on the POWER9 by either modifying system utilities or hooking the system call table. However, if all processes in memory can be traversed by a program running on the BMC, said program could provide an accounting of running processes to compare to what is seen from user space in the POWER9. If said program could identify when processes were being hid from user space of the POWER9, it would confirm the hypothesis of this research.

Specifically, consider the case of the aforementioned *tOrnkit* rootkit. If an anti-malware program running on the POWER9 attempted to use the *ps* system command to list out running processes with the intention of analyzing whether they were malicious, it would invoke the replacement *ps* program. This malicious replacement could give legitimate output with the exception of the malware it intends to hide. However, if a program running on the BMC were to directly examine the process tree in memory, the manipulated system commands inserted by the rootkit could not hide the malicious processes, as this program is not running in the user space of the affected machine. This is also the case for changing the system call table. In no way does this program interact with the compromised system except to examine its memory. It does not rely upon the compromised OS, and neither of the previously mentioned obscuring activities actually change the content of the process tree. For that reason, from the prospective of this program, a rootkit replacing a utility binary and hooking the system call table is equally irrelevant to the efficacy of the program.

3.4 Limitations

In the aforementioned use case, if the rootkit was able to decouple the running process from the process tree, it could evade detection by the proposed method. However, this would be a much harder task on the part of the attacker, as they would

have to figure out how to manipulate the process scheduler and other aspects of the kernel to interact with their process without reference to kernel data structures used for those purposes. The malware could also manipulate where the initial process is loaded in memory, although that would be easy to detect.

3.5 Requirements and Assumptions

This program requires an FSI bus linking a BMC to a POWER9, with *pdbg* installed on the BMC. This program assumes that while the POWER9 may be infected with a rootkit, the BMC is not compromised. It is also assumed that the POWER9 does not have the capacity to block the OCC's ability to access memory.

3.6 Case Study Methodology:

3.6.1 Process Tree

To demonstrate the feasibility of identification of running processes in the memory of the POWER9, the first focus was on finding the location of PCBs in memory. The program running on the BMC needs to be able to find the location of the first PCB in memory to give an address to the first *pdbg* execution. Scanning all 8 GB of memory used by the POWER9 was not feasible as the *pdbg* is only able to read about 3 KB/s according to cursory testing of large data sizes. Therefore, the determination of whether the first node of the process tree was loaded in the same location in memory was paramount. The steps were to first find the location of the process tree from the user space of the POWER9 and then access that location using *pdbg* on the BMC.

There were a number of attempts made to access the memory of the POWER9 from the user space of the POWER9. First, the file */dev/mem* was examined. This is a character file that is an image of main memory. When one prints data from this file, that data is a direct copy of what exists in main memory. However, since

Linux 2.6.26, a kernel configuration has restricted the areas that can be accessed by `/dev/mem`. Only the first 1MB is accessible in kernel versions made after 2016, with that space being left accessible for the purpose of backwards compatibility with Windows[18]. In this experiment, only 65 KB could be read. This could be due to further restriction of the range of memory accessible to `/dev/mem` in this kernel version.

Another attempt used `/proc/kcore`. This file represents the physical memory of the system and is stored in an ELF file format. The first approach consisted of dumping memory and examining it for clues as to the location of the PCBs. User processes were created and executed in order to serve as targets for a search of memory, but scans of the dumped memory did not locate them.

The next attempt was to use GNU Debugger (GDB) with `/proc/kcore` and `vmlinux` to attempt to examine the values of various kernel data structures in real time. `Vmlinux` is an uncompressed executable file containing the kernel in ELF file format. On this system it is a symbolic link to `vmlinux-4.18.0-rc1+`, which corresponds to the running kernel. With `vmlinux` as the executable and `/proc/kcore` serving as the core file, GDB should be able to run as it runs on a regular executable. Unfortunately, this `vmlinux` file was stripped when the kernel was compiled, meaning it had none of the symbolic information necessary to examine its data structures through debugging.

A further attempt was to write a kernel module to directly interact with the process tree. There is a macro `for_each_process` that iterates through the process tree. For kernel modules, `init_task` is a macro that references the root `task_struct` in the process tree. On each iteration of `for_each_process`, a `task_struct` pointer is initialized to point to the current `task_struct` at the given point in the traversal. With the pointer to a given `task_struct`, the kernel module is able to print out its virtual address, its name, `pid` etc. All of the logged virtual addresses had a consistent offset

of 0xc000000000000000 from their address in physical memory. Additionally, it was found that the root of the PCB tree was a *task_struct* called *init_task*, which is always loaded at the same address, 0x1070A00. This was tested repeatedly by rebooting the system. The initial process is always loaded at the same address. If this was not the case, memory would have to be scanned to find the location of the initial process every time the system was rebooted. This could be done by using the same kernel module used to find the location the first time, which would not necessarily be a large hurdle. That being said, its much more convenient that it is always loaded in the same location.

The *task_struct* object is a struct that holds everything the operating system needs to manage each process. The five fields targeted in this approach are *comm*, *tgid*, *children*, *siblings* and *real_start*. *Comm* is short for command, which is the name of the process. The *tgid* field is an integer that holds the thread group id. This is essentially the same as the PID when viewed from the perspective of user space. The *children* field is a *list_head struct*. This *struct* holds two pointers, the first of which is to the *task_struct* spawned by the current process. Like the *children* field, the *siblings* field is a *list_head struct*. It holds a pointer to the next *task_struct* spawned by the parent process. Rather than a traditional tree structure, a tree with *list_head* nodes have parent nodes that link to the first element of a linearly linked list of children nodes. The children nodes then have the *siblings* field which links to the next node in said linked list. The time field has a value equal to the number of nanoseconds after the system booted that the process began. Every field is at a constant offset from the beginning of the *task_struct* in every *task_struct*.

3.6.2 Creating the Program

With the location of the beginning of the process tree discovered, a program could be created to run on the BMC to traverse it. The program, found in Appendix A, is written in Python, which is an interpreted language, meaning the code itself is not compiled before execution.

The program begins by reading the first *task_struct* in the process tree. To do this, it invokes a system call to *pdbg* to read 2000 bytes of memory staring at the address 0x1070A00, which is the address of the first *task_struct*. The result is then piped to *hexdump* in two stages, the second of which, `hexdump -ve ' 1/1 "%.2x"',` formats the output as hexadecimal characters. The result is stored in a 4000 character long character array. Each element in the array is a hex character, each of which contains half a byte of information. The program parses these characters to assign the proper characters to a variable that correspond to each of the *task_struct* fields identified above. The POWER9 is little endian, so the bytes have to be rearranged. For all variables except the addresses, once the bytes are rearranged, a Python function is called that interprets the character array as a hexadecimal value to be cast to either an integer or character variable.

The program constructs a tree structure to represent the process tree. The tree structure is organized in a traditional manner where each child node is stored in a list of child nodes held by the parent node. Once a new node has been constructed and initialized with the values read in with *pdbg*, the program proceeds with a depth-first search algorithm to fill the tree. The specific address pointed to by the *children* field is the address of the *children* field in the first entry in the *listhead* struct linearly linked list, if any children exist. If not, the *children* field stores its own address. The program subtracts an offset of 0x5E0 from this address. The program then compares this address to the address of the current PCB. If it is not the same, the program

is recursively called with the new address. Once the bottom of a branch of the tree has been reached, the program traverses to the siblings of the current PCB. Like the *children* field, the *siblings* field points to the address of the *siblings* field of the sibling PCB. The program subtracts an offset of 0x5F0 from this address. This is then compared to the address of the current PCB. If they are not the same, the program traverses to the sibling node. Otherwise, the program ends at this level of recursion and continues from the parent node. Once this has finished, the program prints out a representation of the process tree approximating the *pstree* utility. For further reference, see Appendix A.

3.6.3 Testing

Testing had four stages, which were

- Demonstrate the program traversing the process tree
- Demonstrate that the program has detected all running processes
- Show the program continues to operate properly when an arbitrary number of user processes are executed and
- Demonstrate that the program can find processes in the process tree that have been hidden from the user space of the POWER9

The first stage test of the program was performed while the only user processes running were a bash shell spawned from a *PuTTY*-created session, *sudo* spawned from that shell, and a shell spawned from *sudo*. With this as the environment, the program traverses the entire process tree, creates a tree to represent it, and then prints out the representation. A portion of the result can be seen in Figure 3.

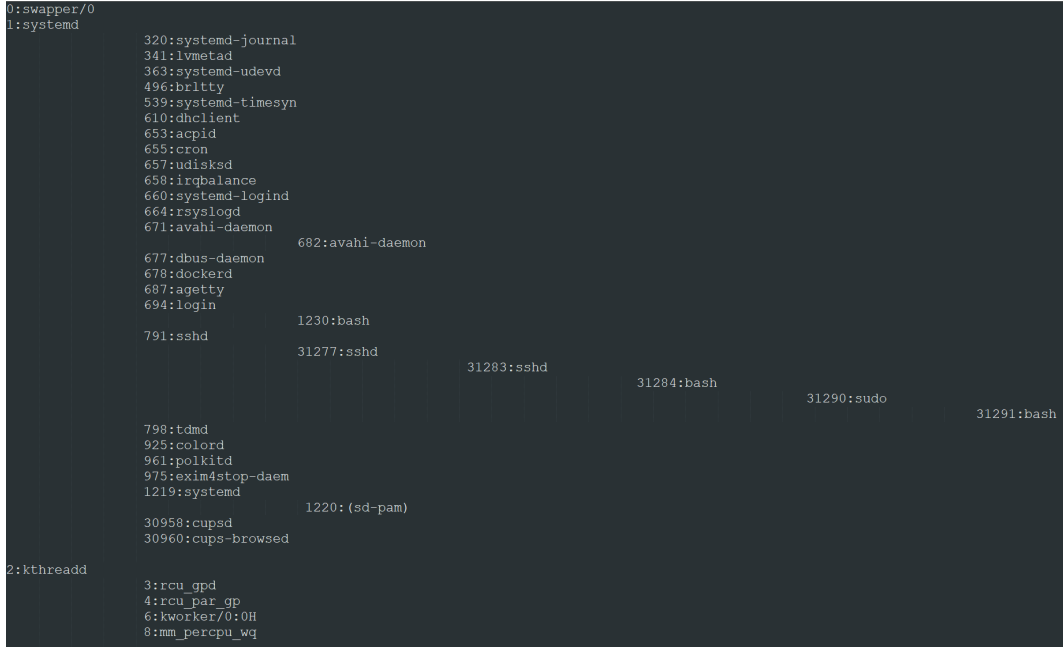


Figure 3. Processes Retrieved from P9 Memory

This result demonstrated that the program had the ability to traverse the process tree. When the system boots, the first process created is *swapper/0*. In turn, it spawns *systemd* and *kthreadd*. The “d” at the end of these processes stands for daemon. This is true of many processes running on the system. *kthreadd* handles kernel processes. All other processes fall under *systemd*. *sshd* is the ssh daemon. It spawns a child process which is also from the ssh daemon, which in turn spawns another ssh daemon process. This process spawns a bash shell. From this shell, the user opened a *sudo* session. The command *sudo -s* spawns a bash shell with root permissions. In Figure 3, this hierarchy of these processes is shown. This provides a concrete example of process hierarchy, as well as an example of how this hierarchy is preserved and reproduced by the program on the BMC.

The next stage consisted of running the *ps* command on the POWER9 before executing the program from the BMC. The number of processes identified by each

are then compared. If a process is found by one and not the other, the reason for this is investigated. At this point in the tests, no processes are being hidden from the user space of the POWER9. This process was repeated 5 times, and the results are listed in Table 1.

Table 1. Baseline Process Test

Number of Processes Output from Power9	Number of Processes Read with pdbg
208	204
205	204
205	204
206	204
205	204

The first column shows the number of processes read from user space of the POWER9. The second column shows the number of processes found by the program running on the BMC. The number of processes read with the program is lower than the number of processes found from user space on the POWER9 because in each case between 1 and 4 kernel thread processes terminated before the program traversed to them. All other programs found by executing *ps* on the POWER9 were accounted for by the program, and vice versa. This stage of the test also showed the background number of processes running on the POWER9 varied by only a few processes.

For the third stage of the test, two programs were created to run on the POWER9 that slept in a loop and ran in the background. These two programs would be executed in pairs between 8 and 200 times. This means that the pairs would be executed 8 times, each of the two programs would be executed 8 times, creating 16 processes. This was done for the purpose of demonstrating that the program running on the BMC works when an arbitrary number of user spawned processes are running on the POWER9. Additionally, the processes retrieved by the program were checked against processes

found from user space on the POWER9 using a more systematic approach. The *ps* command with the options “-ef pid,lstart,cmd” was executed on the POWER9 immediately before the program was ran on the BMC. This command outputs every running process, their PIDs as well as when each process began. These results were stored to a file and moved to the BMC. The program then reads in the data from this file and constructs a representation of the process tree as is seen from user space of the POWER9. Next the program traverses the process tree using *pdbg* and reconstructs a tree structure to represent the present state of the process tree as it exists in memory. The program then checks to make sure that each entry in the process tree constructed with the data read with *pdbg* is in the output of the *ps* command executed on the POWER9. The intention is to check for processes being hidden from the user space of the POWER9, although none are being hidden at this point. When a process is read in by the program using *pdbg* but is not present in the results from the *ps* command, the process is marked as flagged. Flagged processes have their starting times examined to determine whether the process could have spawned while the program was running. The program notes the latest time of a process detected by the *ps* command to start and compares that to the flagged processes. If a flagged process began before a process included in the *ps* results but does not show up in the results of the *ps* command, the process is identified as a hidden process. The results of this stage of the test are displayed in Table 2.

Table 2. Dummy Processes

Processes Output from Power9	Processes Read with pdbg	Dummy Process Pairs Spawned	Flagged	Hidden
215	216	6	3	0
446	445	121	0	0
232	232	14	0	0
244	241	19	0	0

In each of these executions, there were no hidden processes. The number of processes output by the *ps* command were indicated by the first column, and the number read in by the program utilizing *pdbg* were indicated by the second column. The discrepancy between the number of processes found by output of the POWER9 and the program is accounted for by kernel threads spawning and terminating before and during the execution of the program. Other than these kernel threads, every process output by *ps* was found by the program.

The last stage of the test demonstrated the detection of “hidden processes”. To illustrate this point, a modified version of the *ps* command was crafted to run on the POWER9 that would otherwise run normally, but omit the name of a chosen malicious process. This was done by modifying the source code of a number of files in the *procps* family of utilities. This is essentially a compendium of files that when compiled together produce a number of system utilities such as *ps*, *proc* and *top*. After modifying several files, a new version of *ps* was compiled and moved to the */bin* folder. This is intended to imitate a rootkit such as *t0rnkit* that compiles an alternative version of *ps* and replaces this normal utility. The name of the designated malicious program was *iAmMalware*. A program with that name was created and executed in the background a variable number of times. The malicious *ps* command was then ran. The same procedure of spawning dummy processes as the third stage was also followed. Procedures for flagging and identifying hidden processes was the same as for stage 3, with the output from user space of the POWER9 coming from the modified *ps* command. In every runthrough, all malware processes were found and correctly identified as hidden. All processes that were flagged but not hidden were again the result of kernel thread processes spawning after the execution of the *ps* command. These results are reported in Table 3.

Table 3. Malware Detection

Processes Output from Power9	Processes Read with pdbg	Dummy Process Pairs Spawned	Malware Spawned	Flagged Processes	Hidden Processes
247	254	21	9	10	9
543	548	109	6	9	6
290	298	43	8	9	8
222	231	9	10	10	10
544	546	170	2	5	2

3.7 Conclusion

First, the identification of the location of the process tree in memory and the discovery that it loads at the same location after every boot made the implementation of the program defined in this test case possible. The PCBs have their fields located at constant offsets from the beginning of the *struct*, which allows for a program to jump from PCB to PCB by translating data at a constant offset from the beginning of the *struct* into the address of the next PCB. Every process that runs has an entry in this tree. This test case demonstrated the possibility of reading over the entire process tree using *pdbg*. It consistently found every process in the process tree with the exception of those processes that terminated before they could be reached. Additionally, it was shown that processes being hidden from user space on the POWER9 have no bearing on the ability of the program to traverse the entire process tree, and then identify processes being hidden from user space on the POWER9. This capability would have the application of allowing for real-time monitoring of running processes on a potentially compromised machine from a separate system to ensure the correct operation of process detection of anti-malware software. As demonstrated in this test case, a system utility replaced has no impact on the ability of this program to correctly identify every process in the process tree. This would also be the case with a rootkit that was able to compromise the system call table.

The main drawback in terms of feasibility is the time to complete an entire read of the system. *pdbg* has a slow data read rate, the specifics of which are discussed in the next chapter. Processes both complete and begin in the time it takes for the program to complete. This means that malware that spawns multiple quick running processes would be hard to detect, as its processes may exit the process tree before the detection program reaches them. However, malware that spawns processes that persist over longer periods could still be detected. The time it takes to complete a full scan of the process tree would limit the frequency at which this capability could be deployed. However, if hardware with faster access times to memory was utilized, these drawbacks could be significantly minimized, and real world application could become more feasible.

This confirms the hypothesis to the extent that a program running on the BMC can find processes hidden from the user space of the POWER9, as long as the program traverses to the hidden process before the process terminates. The experiment in Chapter IV discusses the bounds in which this approach can find hidden processes before they terminate.

3.8 Summary

This chapter presented a test case demonstrating the ability of utilizing *pdbg* in a Talos II architecture to traverse, thus confirming the hypothesis within a limited scope. Chapter IV will present an experiment examining the bounds in which this approach is effective.

IV. Experimental Scenarios

4.1 Overview

This chapter outlines experiments relating to the functioning of *pdbg* in the Talos II architecture. To characterize the time complexity of *pdbg* in different conditions, an experiment, divided into two scenarios, was created. The scenarios were carried out by creating a program that made repeated calls to *pdbg*, and recorded the time to complete each *pdbg* execution. The first scenario measures the rates at which data is read from the POWER9's memory for various sizes of data. The second scenario isolates the overhead of each *pdbg* call by using a variable number of reads for a given data size. The purpose of gathering this information is to generate a predictive equation to analyze the feasibility of various memory monitoring techniques which utilize *pdbg*.

4.2 Experimental Objective

The objective of this experiment is to derive an equation that predicts the cost in time of executing *pdbg* to read a given size of data. With this equation, the estimated time to execute a program that utilizes *pdbg* can be calculated. A specific example of is using this equation to predict the time to execute the program from Chapter III, which can be found in AppendixA, when a given number of processes are running on the POWER9. It would also be able to predict the performance of modifications to the program, such as if multiple targeted *pdbg* executions were used to read the relevant fields from a PCB rather than a single execution that reads the entire PCB.

4.3 Design

This experiment is designed to model the performance of *pdbg* under varying conditions, with the ultimate goal of determining feasibility of using *pdbg* to monitor active data structures in memory. The characteristics of *pdbg* to be determined are

- latency of memory accesses
- overhead of *pdbg* execution
- variability

To this end, two experimental scenarios were conducted to isolate these individual metrics. Each scenario consists of trials, sets, and blocks. Each trial consists of a single call to *pdbg* to read a variable amount of data. Trials are organized into sets. A set consists of running through a number of trials for a defined range of data sizes. These sets are organized into blocks, where a block is the repetition of a set in order to find the average data retrieval rates and variance.

The first scenario was intended to isolate the data retrieval rate. To accomplish this, it measures the latency of *pdbg* for a various data read sizes.

The second scenario isolates the overhead of *pdbg*. To accomplish this, it measures the duration of reading a fixed data size with a variable number of *pdbg* reads. Each trial consists of a number of *pdbg* executions to read a given amount of data. A set consists of a number of trials each reading the same amount of data with a variable number of reads. A block consists of a number of sets at a given data size. Different blocks have different data sizes.

4.4 Assumptions

This research assumes that accessing any address in memory will have the same latency as accessing any other. For this reason, each trial starts at the same address.

4.4.1 Control Variables

The control variables are the system environment and location of the starting read point. Before beginning testing for each new block, the BMC was restarted. No user-created processes were running during this execution, so the processor workload was expected to be relatively stable from trial to trial. As was shown in Chapter III, the number of baseline processes running on the POWER9 fluctuates from about 203 processes to 208 processes, the discrepancy arising from kernel threads being initiated and terminated by the OS. Each data read is a multiple of 8 bytes because the OCC retrieves 8 bytes from memory at a time. If a single execution of *pdbg* is not divisible by 8, the final read would be less than 8 bytes, thus increasing the average read time and skewing the results of especially the smaller read sizes. This was the case for both Scenario 1 and Scenario 2.

4.4.2 Independent Variables

The independent variable for the first scenario is the data size for each trial. In this scenario, each trial reads a different amount of data. The independent variable for the second trial is the number of times *pdbg* is executed to read a fixed amount of data.

4.4.3 Dependant Variables

The dependant variables were the latency of each trial and the variance for all trials that either read the same amount of data for scenario 1, or all trials that read the same amount of data and execute *pdbg* the same number of times.

4.5 Application Scenarios

4.5.1 Scenario 1

The first scenario set out to measure the data retrieval rate. For the first block, each set consisted of 100 trials starting at 8 bytes and increasing by 8 bytes for each trial. This means the individual reads ranged from 8 bytes to 800 bytes for each set. The sets were repeated 50 times and averages for each data size and the standard deviation of each trial size were calculated. For each set, the trials were performed back to back. The POWER9 was rebooted between blocks.

For the second block, each set consisted of 23 trials starting at 808 bytes and increasing by 8 bytes for each trial, giving a range from 808 bytes to 1016 bytes. The sets were repeated 50 times each and averages for each set and the standard deviation of each trial size were calculated.

For the third block, each set consisted of 20 trials starting at 1152 bytes and increasing by 128 bytes for each trial with the largest value being 3584. The sets were repeated 50 times each and averages for each set and the standard deviation of each trial size were calculated.

For the last block, each set consisted of 8 trials starting at 4096 bytes doubling for each trial with the largest value being 524288. The sets were repeated 50 times each and averages for each set and the standard deviation of each trial size were calculated.

4.5.2 Scenario 2

The second scenario set out to isolate the overhead of each *pdbg* execution. Like the first scenario, the second scenario is organized into blocks, each consisting of sets which in turn consist of multiple trials. For example, as shown by Table 4, the first block consists of each trial reading 128 bytes of data. In each set, trials were conducted that had read sizes for individual calls to *pdbg* of 8, 16, 32, 64, and 128

bytes, whereas the total amount of data read was always 128 bytes. The trial with the *pdbg* execution size of 128 only executed *pdbg* once, whereas the trial with the *pdbg* execution size of 8 executed *pdbg* 16 times. The amount of data read for each trial is a power of 2. The reasons for this is each data read size is divisible by 8, and the total amount of data read for each block is divisible by the data read sizes of each of the trials. This allows for the same data read sizes for each block to allow for easier comparison. The data for each of the five blocks is shown in Tables 4-8.

4.6 Performance Evaluation

4.6.1 Scenario 1

The results for each block were plotted on a graph and various regression equations were used on the data to determine whether the data was linear, exponential etc. For each block, a graph with the average duration of each trial size is included. Each graph includes a regression equation that best models the trends in the data, as well as the R^2 value for the regression equation. An R^2 value is a goodness-of-fit measurement that measures how closely the dependent variable is determined by the independent variable for the given regression equation. An R^2 value of 0 indicates no correlation, and an R^2 value of 1 indicates perfect correlation. All R^2 values fall between 0 and 1. All regression equations for the average durations were solidly linear. That means the data is described better by a linear regression equation than other forms of equations, and that the R^2 value was high. For these experiments, a high R^2 value is defined as a value above 0.95. A linear equation is an equation in the form $y = mx + c$, where y is the dependent variable, m is the slope, x is the dependent variable, and c is a constant. Additionally, graphs are included that show the standard deviations for each trial size, and the ratio of each average standard deviation to the total duration. For each measurement, regression equations are used to describe the trends in the

data. A standard deviation is a value for a data set that when added to or subtracted from the mean, should give upper and lower bounds in which 68.2% of data should fall. Ideally, this research should analyze the normality of the data, the degree to which the data fit a normal distribution curve, but for the purpose of a high level overview of variance, only the standard deviation for each trial size was calculated. The ratio of standard deviation to mean gives a ratio of variance. For example, if the mean for a data set is 1, the standard deviation is 0.05, and the data is normal, 68.2% of data will fall within 5% of the mean. Both the standard deviations and their ratio to the corresponding averages were logarithmic for Scenario 1, but linear for Scenario 2. Logarithmic equations take the form $y = b * \ln(x) + c$, where y is the dependent variable, b is the coefficient for the independent variable, $\ln(x)$ is the natural log of the independent variable, and c is a constant. A natural log is a logarithm with the base of e . The R^2 values of these regression equations varied to a great extent, with the data in first block being much more consistently logarithmic than the latter blocks. Figure 4 shows the results for data sizes 8 through 800 bytes.

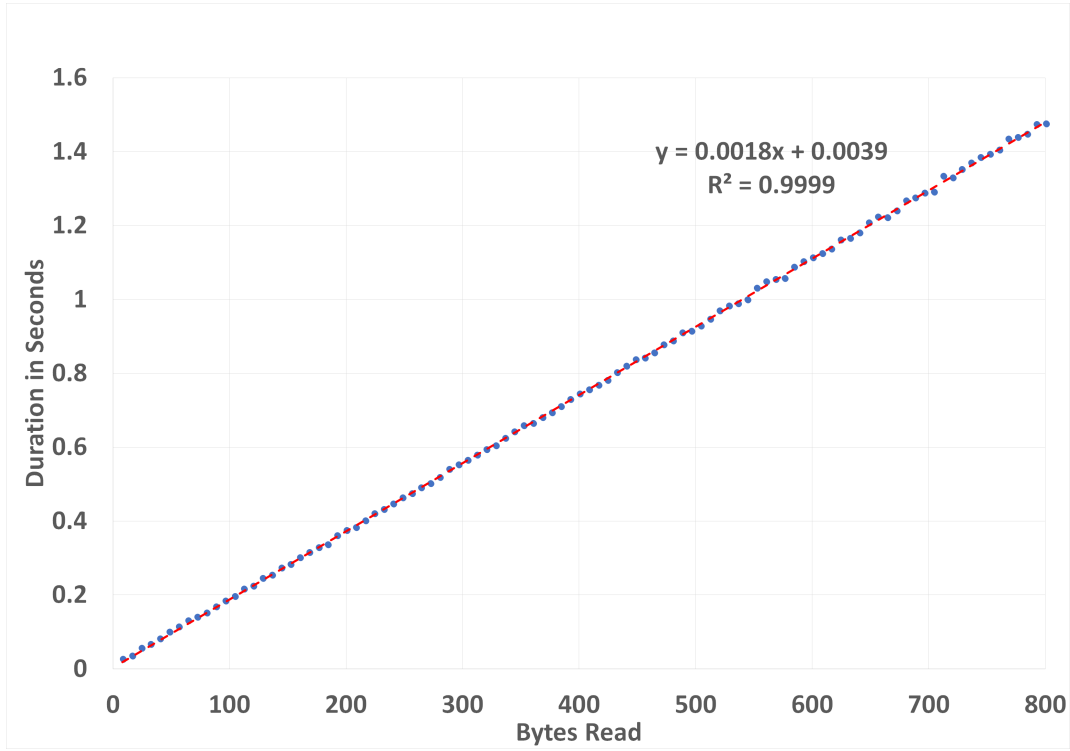


Figure 4. Latency of *pdbg* Reads from 8 to 800 Bytes

The data are very clearly linear. The R^2 value of 0.9999 indicates these data are almost perfectly linear. The slope is 0.0018, meaning that the cost of each additional byte read is on average 1.8 milliseconds. The constant identified here is 0.39 milliseconds. Figure 5 shows the standard deviation of the trials for each data point in Figure 4 and the corresponding ratio of said standard deviation to each data point. For the purpose of comparing the standard deviation to its ratio in relation to the average duration of a given data read size, the y-axis remains unlabeled. For the standard deviation, the y-axis is the standard deviation for all trials at a given data read size in seconds. For the ratio, the y-axis is the standard deviation in seconds at a given data read size divided by the average duration of reads of that same read size in seconds, giving a percentage with no units. This is the case for all charts plotting this data in both scenarios unless stated otherwise.

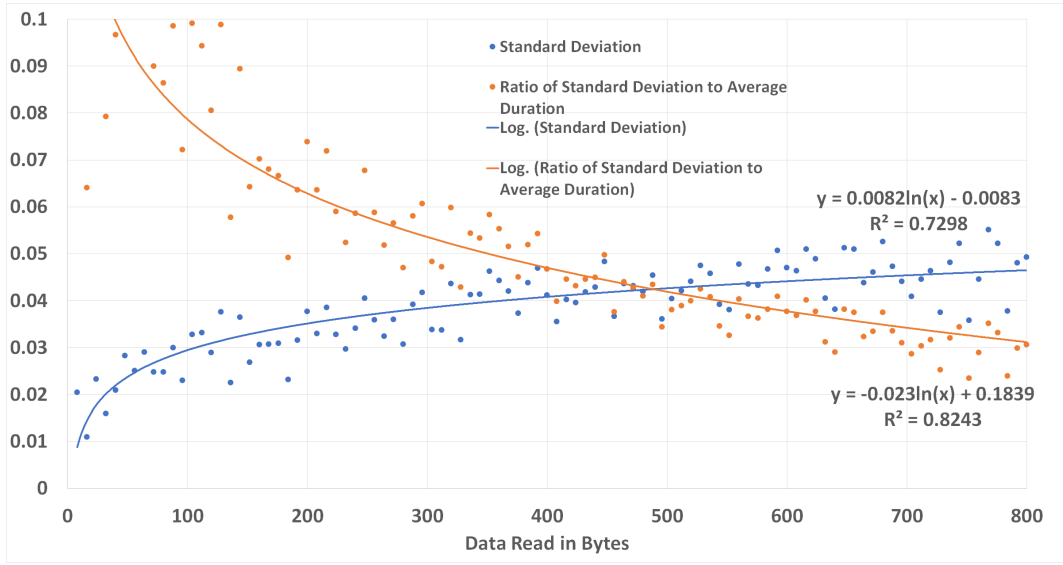


Figure 5. Standard Deviation and Ratio to Average Duration for Trials Between 8 and 800 Bytes

The standard deviations tend to rise logarithmically as total amount of data read increases. This trendline has an R^2 value of 0.7286, meaning that while the data points tend to follow the trendline there is a fair amount of randomness, the nature of which will be commented upon later. The ratio of standard deviation to average duration decreases logarithmically. The R^2 value for this data was 0.8243. This indicates that the ratio of standard deviation to data read size varies to a lesser degree than does the increase in standard deviation as data read sizes increase. By viewing the chart, it can be observed that the ratio of standard deviation to average duration fits the trendline better as data sizes increase.

The second block ranged from 808 to 1016 bytes. Like block one, each subsequent trial differed from the previous trial by 8 bytes. Figure 6 shows the results for this block.

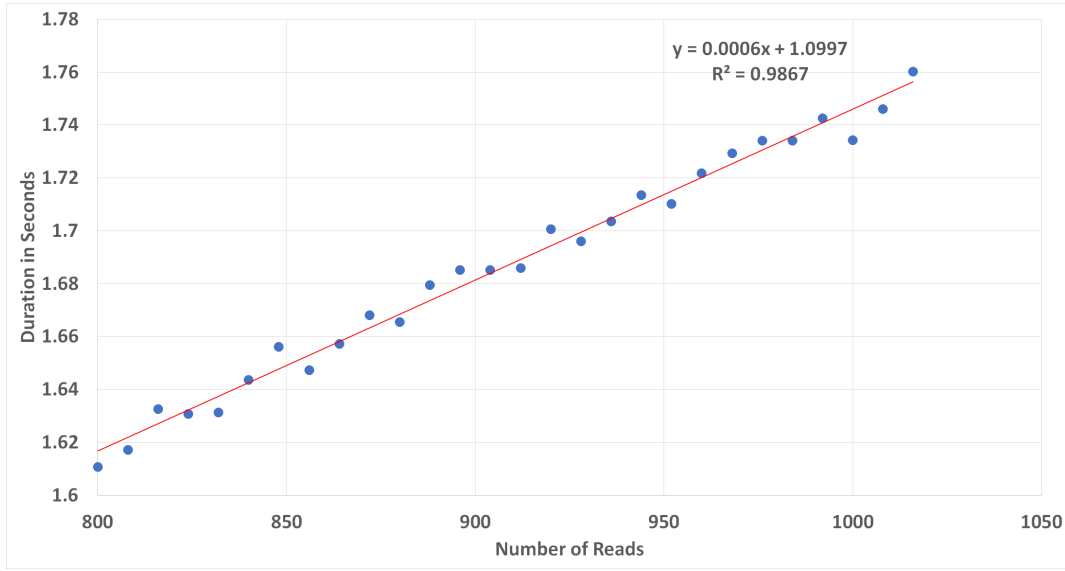


Figure 6. Latency of *pdbg* Reads from 808 to 1016 Bytes

From 800 to 1016 the data remains solidly linear but slightly less so. The slope is 0.6 milliseconds, a 67% decrease from the previous block. This indicates the overhead drops precipitously after the 800 bytes. Figure 7 shows the standard deviation of the trials for each data point in Figure 6.

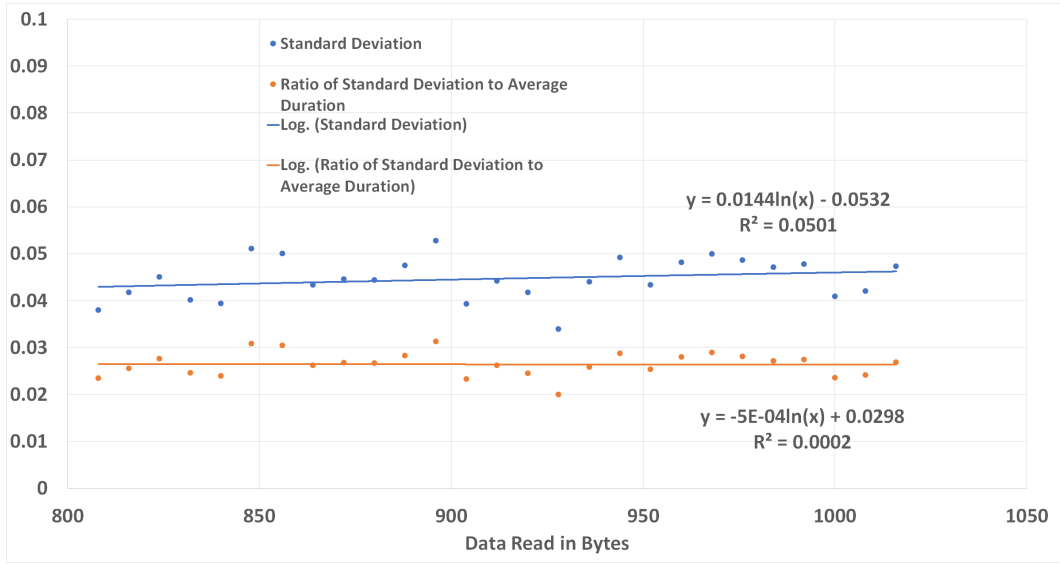


Figure 7. Standard Deviation and Ratio to Average Duration for Trials Between 808 and 1016 Bytes

The trends in the standard deviations became much less consistent. The regression equation describing the standard deviations was $0.0087\ln(x) - 0.0147$. The R^2 value for this data was only 0.0201. This indicates an incredibly weak correlation of the data to the trendline. The regression equation of the change in ratio of the standard deviation to the data read size was $-0.004\ln(x) + 0.0542$. The R^2 value for this data was only 0.0126. This is an even weaker correlation to the trendline.

The third block ranged from 1152 to 3584 bytes. The size of each read was separated by 128 bytes, meaning there were 20 trials in each set. Figure 8 shows the results from this block.

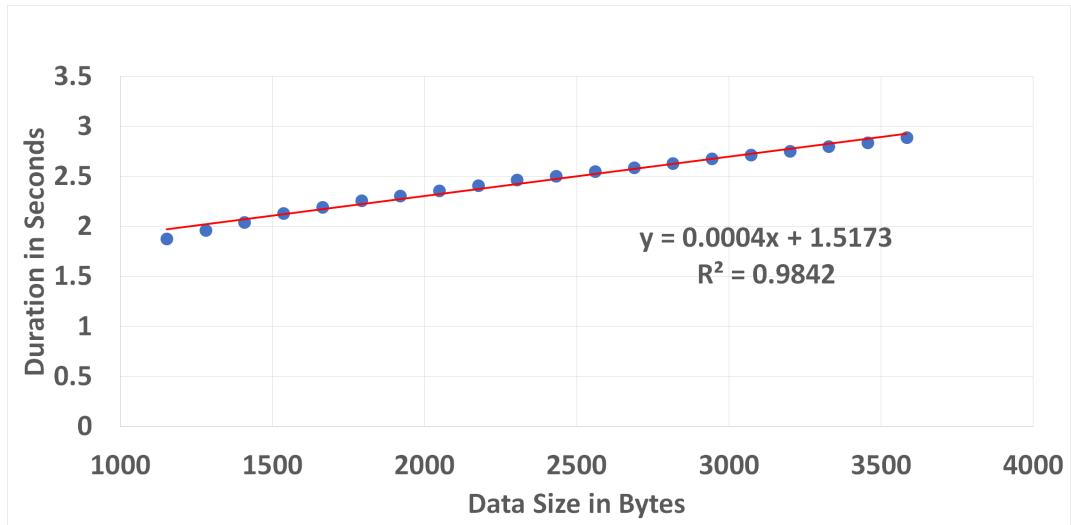


Figure 8. Latency of *pdbg* Reads from 1152 to 3584 Bytes

The data was strongly linear. The slope decreased to 0.0004, a 33% decrease from the previous trial. The R^2 value of the trendline was 0.9842. Figure 9 shows the standard deviation of the trials for each data point in Figure 8.

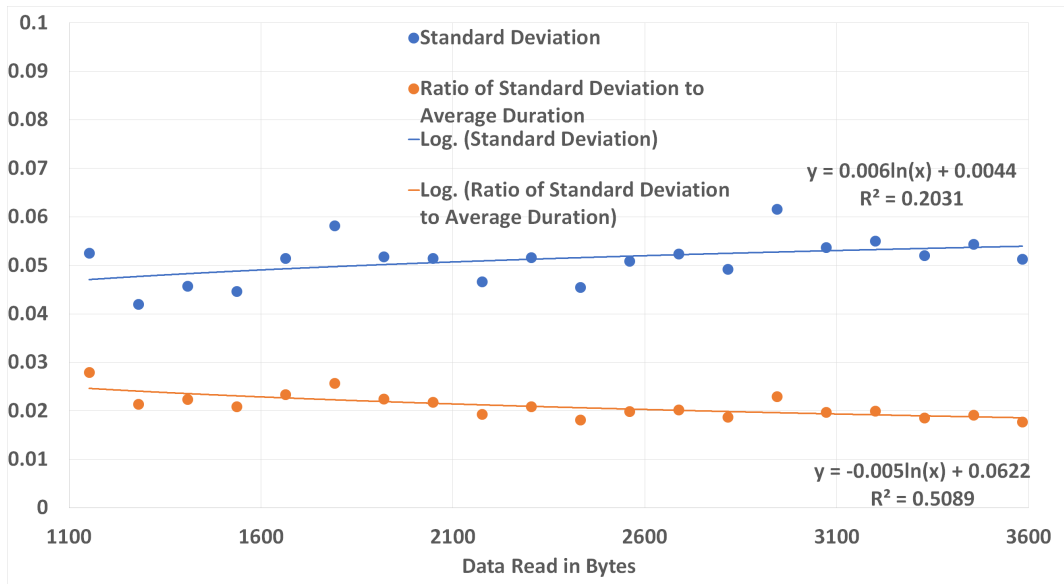


Figure 9. Standard Deviation and Ratio to Average Duration for Trials Between 1152 and 3584 Bytes

The increase in standard deviation was described by the trendline $0.0066\ln(x) + 0.0002$. Its R^2 was 0.0206, indicating that the standard deviations continued to vary widely. The ratio of the standard deviation to the total amount of data read was described by the trendline $-0.0005\ln(x) + 0.0612$. Its R^2 value was 0.4444. This is markedly greater than the value that this equation had in the previous block.

In order to more directly compare results of these three blocks, Figure 10 plots all three on one graph.

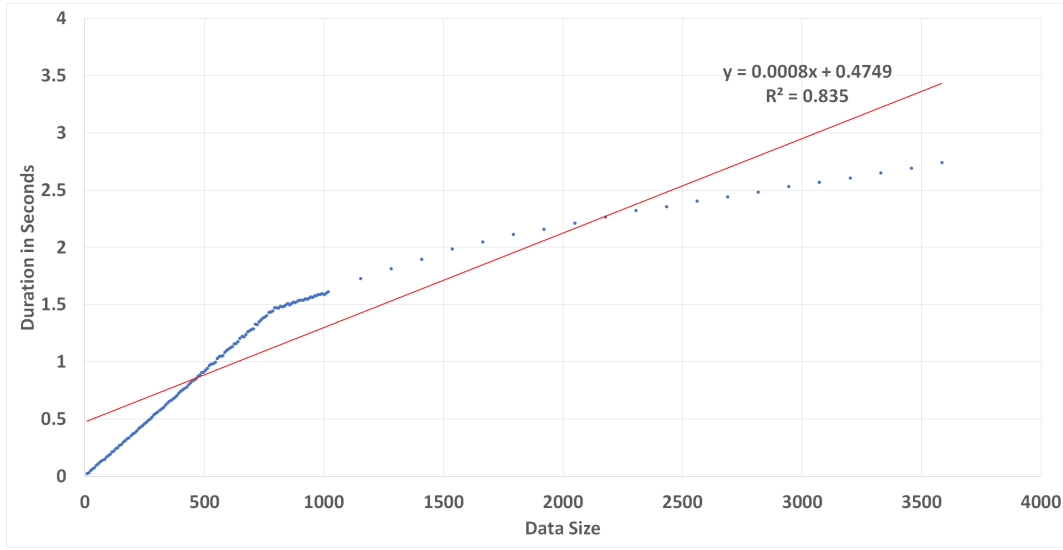


Figure 10. Latency of *pdbg* Reads from 8 to 3584 Bytes

The data shows a piecewise distribution of the data, where reads under 800 bytes increase at a solidly linear rate, whereas reads above this threshold increase linearly, albeit with a much smaller slope, which itself gradually decreases over time. A piecewise distribution of data indicates that different subsets of the data are governed by different equations. A linear trendline for this data has an equation of $0.0008x + 0.6114$. The slope of 0.0008 is close to an average of the three slopes 0.0018, 0.0006 and 0.0004.

Figure 11 plots all standard deviations and standard deviation ratios for the first three blocks.

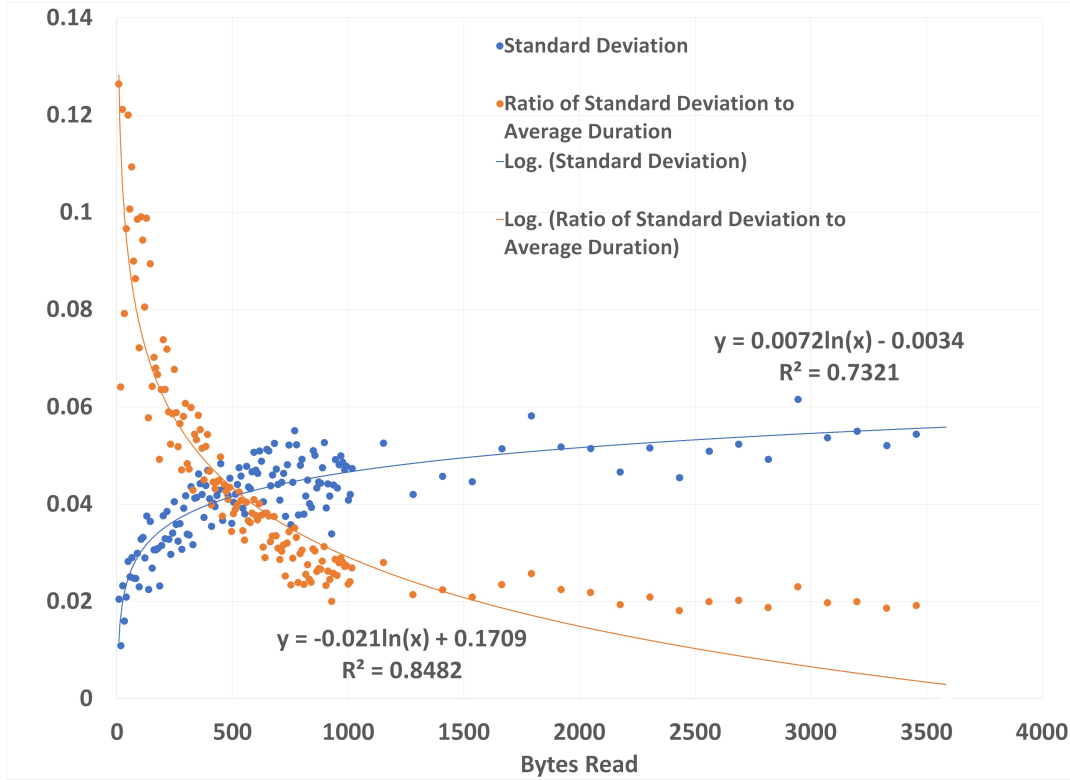


Figure 11. Standard Deviation and Ratio to Average Duration for Trials Between 8 and 3584 Bytes

The R^2 values for each equation indicate a moderate level of correlation between bytes read and the increase in total standard deviation size and decrease in ratio of standard deviation to average duration. Around 1500 bytes, the ratio of standard deviation to average read size stabilizes around 0.02, and oscillates at that level.

Figure 12 shows the last block. The last ranges in data read sizes from 4096 to 524288. Each value is doubled, which gives 8 reads for each set.

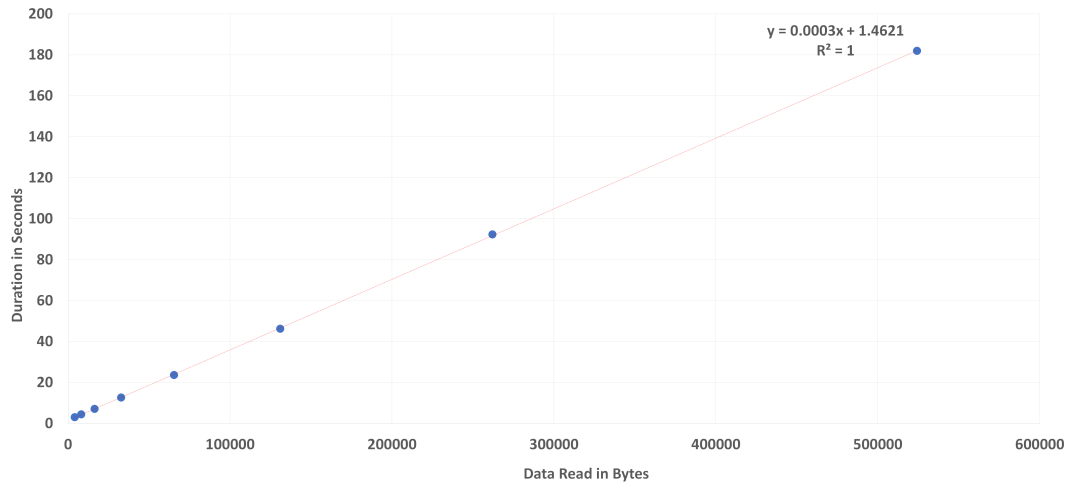


Figure 12. Latency of *pdbg* Reads from 4096 to 524288 Bytes

The trendline was $0.0003x + 1.4621$. The R^2 value was 1. This indicates that the data was perfectly linear. The standard deviations and ratios for block 4 were split into Figure 13 and Figure 14, because the range of y values for each were no longer close enough to be displayed in a single chart.

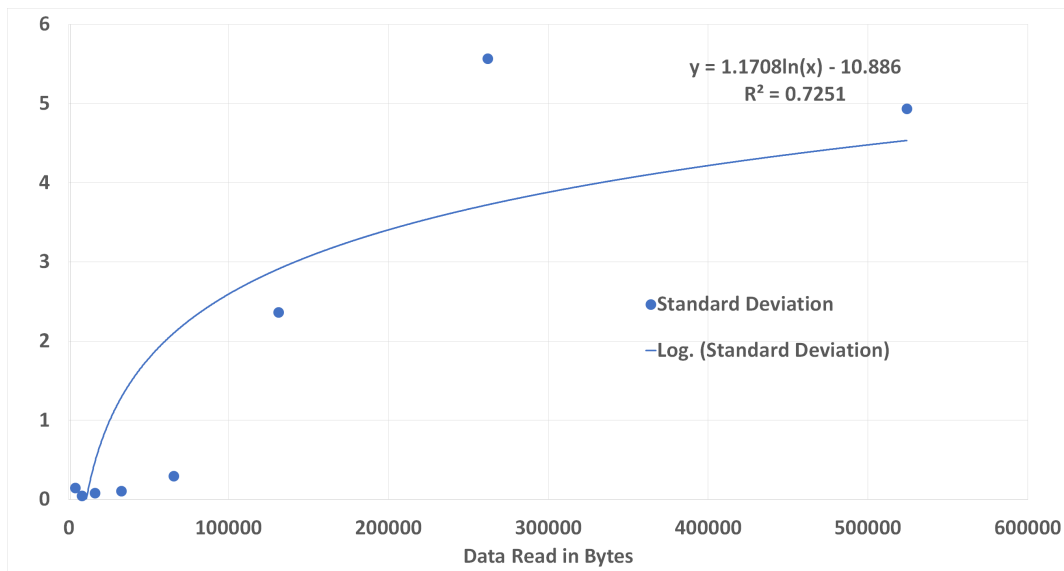


Figure 13. Standard Deviation of Trials Between 4096 to 524288 Bytes

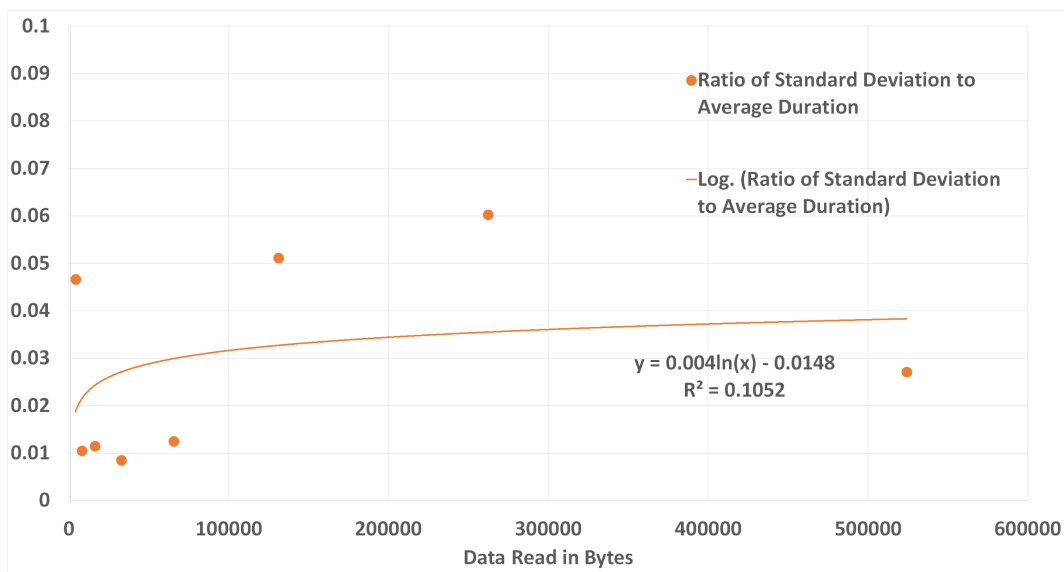


Figure 14. Ratio of Standard Deviation to Average Duration for Trials Between 4096 to 524288 Bytes

The standard deviations and their corresponding ratios to average duration were much less consistent than the previous block. Three data points were over 0.04 for the ratio, which broke with the general trend of the ratio decreasing at larger data read sizes. The other 5 data points were slightly above or well below 0.02, which is where the data stabilized for the previous block.

4.6.2 Scenario 2

The second scenario focused on the isolation of *pdbg* overhead. For these tests, each block has a set data read size. These sizes are 128, 256, 1024, 2048, and 4096. The variable was the number of *pdbg* reads used to read the set amount of data. The results of these blocks are displayed in Tables 4-8.

Table 4. Variable Number of Reads for 128 Bytes

Read Size	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Avg
8	2.322	2.444	2.434	2.379	2.343	2.483	2.421	2.359	2.392	2.397
16	1.280	1.311	1.291	1.286	1.349	1.294	1.311	1.321	1.313	1.306
32	0.819	0.715	0.750	0.886	0.753	0.756	0.730	0.731	0.707	0.761
64	0.475	0.483	0.490	0.458	0.466	0.502	0.504	0.552	0.505	0.493
128	0.352	0.353	0.352	0.355	0.332	0.333	0.374	0.386	0.354	0.354

Table 5. Variable Number of Reads for 256 Bytes

Read Size	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Avg
8	4.731	4.944	4.773	4.838	4.787	4.787	4.740	4.820	4.812	4.804
16	2.612	2.595	2.595	2.638	2.571	2.617	2.657	2.544	2.685	2.613
32	1.487	1.479	1.502	1.500	1.540	1.533	1.470	1.626	1.490	1.514
64	0.958	0.987	0.973	0.936	1.008	0.922	0.986	0.934	0.943	0.961
128	0.704	0.691	0.671	0.756	0.660	0.733	0.731	0.680	0.693	0.702
256	0.642	0.591	0.599	0.575	0.557	0.548	0.568	0.642	0.577	0.589

Table 6. Variable Number of Reads for 1024 Bytes

Read Size	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Avg
8	20.57	20.55	20.47	20.62	20.62	20.62	20.57	20.52	20.71	20.58
16	11.36	11.21	11.25	11.37	11.12	11.26	11.16	11.27	11.28	11.25
32	6.476	6.513	6.644	6.519	6.494	5.911	6.600	6.645	6.547	6.483
64	4.231	4.151	4.227	4.177	4.283	3.145	4.243	4.187	4.189	4.093
128	2.983	3.070	2.978	2.989	3.047	3.380	3.130	3.066	3.057	3.040
256	2.543	2.398	2.558	2.565	2.440	2.451	2.444	2.545	2.408	2.484
512	2.143	2.208	2.142	2.184	2.139	2.258	2.305	2.136	2.209	2.191
1024	1.775	1.784	1.812	1.852	1.878	1.767	1.741	1.779	1.869	1.806

Table 7. Variable Number of Reads for 2048 Bytes

Read Size	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Avg
8	41.36	41.29	40.99	41.04	41.22	41.16	41.38	41.25	41.10	41.20
16	22.54	22.56	22.40	22.49	22.57	22.32	22.64	22.70	22.45	22.52
32	13.13	13.08	12.99	13.13	13.12	13.10	13.05	13.10	13.07	13.09
64	8.439	8.437	8.392	8.397	8.444	8.476	8.554	8.404	8.395	8.438
128	5.784	6.219	6.030	6.072	6.066	6.101	6.125	6.145	6.090	6.070
256	5.048	4.723	4.941	5.010	4.930	5.050	4.956	5.017	4.936	4.957
512	4.406	4.425	4.435	4.429	4.427	4.370	4.408	4.479	4.388	4.419
1024	3.581	3.621	3.528	3.588	3.583	3.651	3.720	3.554	3.621	3.605

Table 8. Variable Number of Reads for 4096 Bytes

Read Size	Trial 1	Trial 2	Trial 3	Trial 4	Trial 5	Trial 6	Trial 7	Trial 8	Trial 9	Avg
8	82.51	82.70	82.30	82.49	82.45	82.26	82.72	82.44	82.26	82.46
16	44.94	44.91	45.00	44.96	44.89	45.03	45.13	45.04	44.27	44.91
32	26.40	26.08	26.10	26.23	26.32	26.17	28.79	26.09	26.14	26.48
64	16.96	16.88	16.84	16.81	16.72	16.80	16.92	16.82	17.10	16.87
128	11.85	12.13	12.35	12.21	12.15	12.10	12.20	12.28	12.21	12.16
256	9.855	9.886	9.923	10.02	10.04	10.01	9.996	10.00	9.862	9.955
512	8.779	8.816	8.329	8.719	8.951	8.866	8.838	8.848	8.889	8.782
1024	7.156	7.098	7.168	7.323	7.050	7.210	7.226	7.333	7.137	7.189

The results from the first two blocks with the data read sizes of 128 bytes and 256 bytes are graphed in Figure 15 and the third through fifth blocks with the data read sizes of 1024, 2048 and 4096 bytes are graphed in Figure 16. This was done for purpose of like scale between those groupings of blocks.

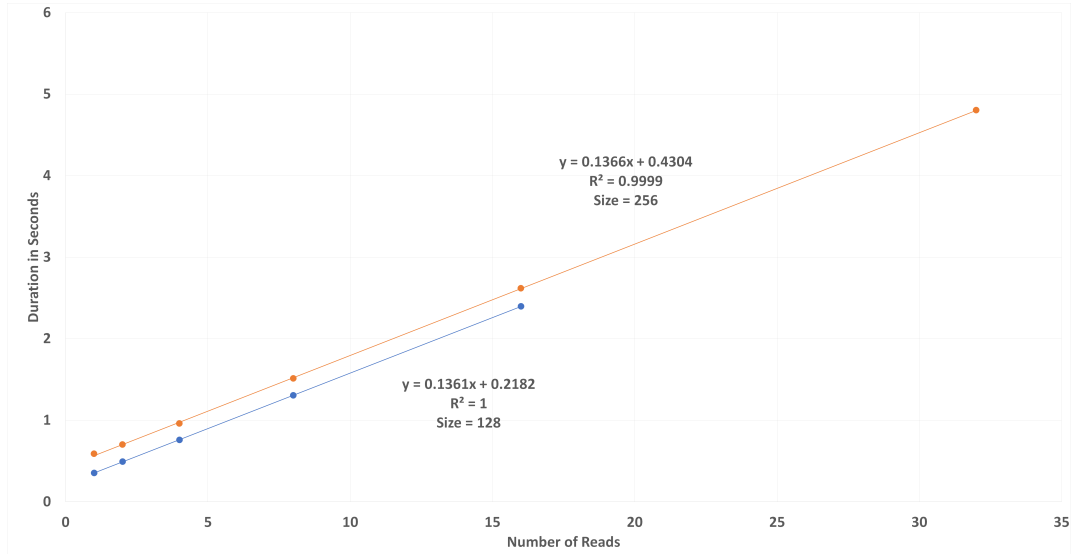


Figure 15. Variable Reads Results 128 and 256 Bytes

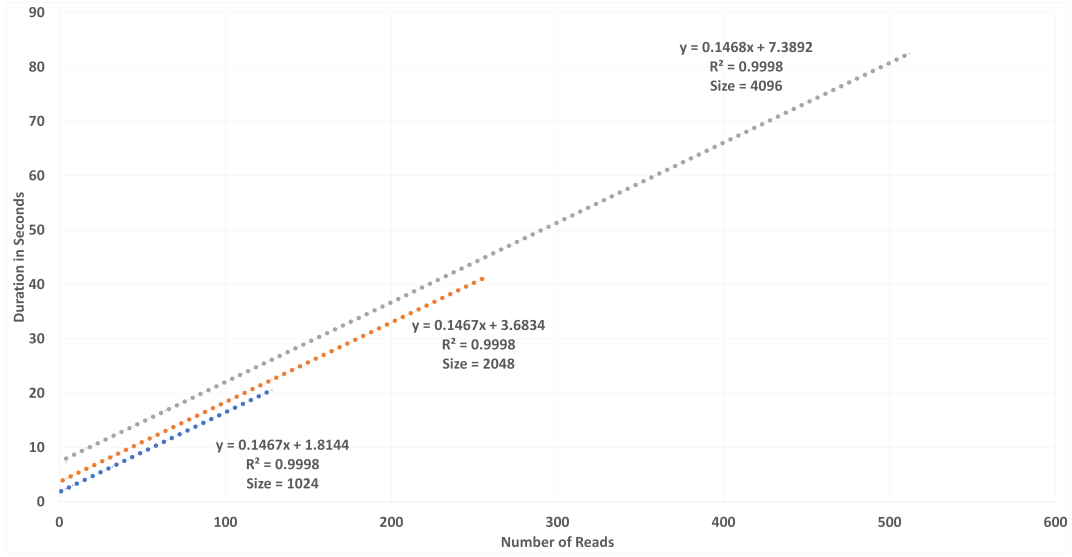


Figure 16. Variable Reads Results 1024 - 4096 Bytes

The data for all blocks were linear. The slope for 256 byte and 512 byte data sets were very similar, and the slopes in each other case was almost identical at around 0.1467. The R^2 values were all at least 0.9998, indicating a near perfect adherence of the data to these trendlines. Figures 17 - 21 show the standard deviations and associated ratios for each block.

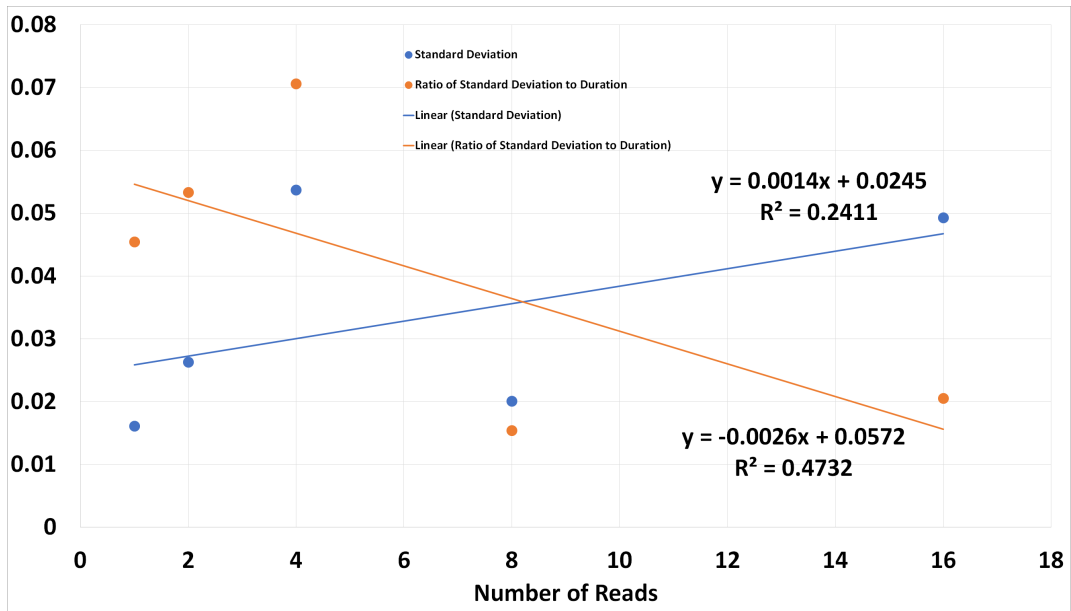


Figure 17. Standard Deviation and Ratio for Variable Reads at Data Read Size 128

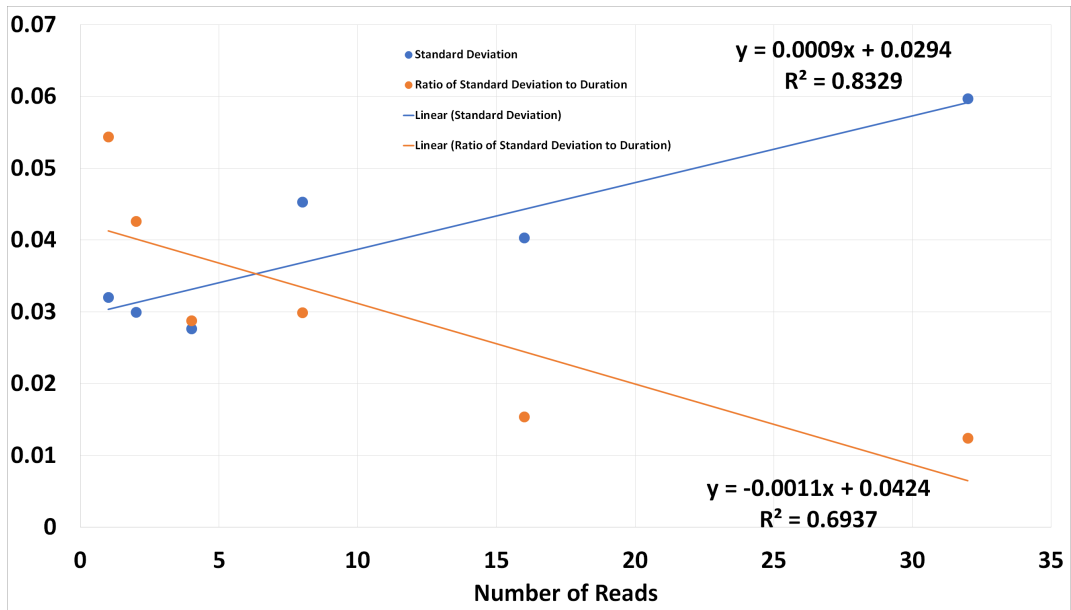


Figure 18. Standard Deviation and Ratio for Variable Reads at Data Read Size 256

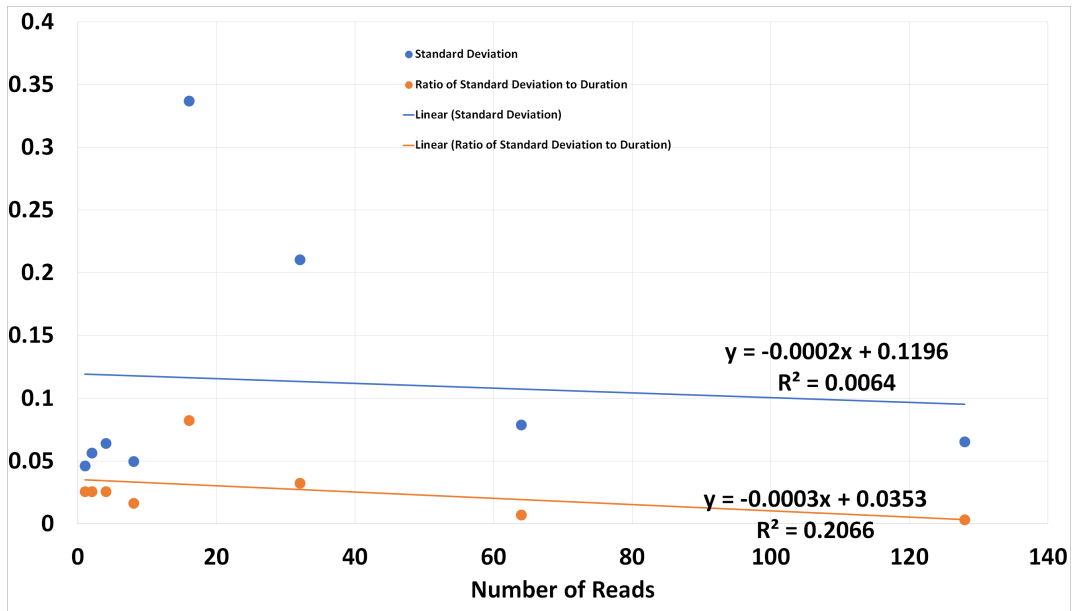


Figure 19. Standard Deviation and Ratio for Variable Reads at Data Read Size 1024

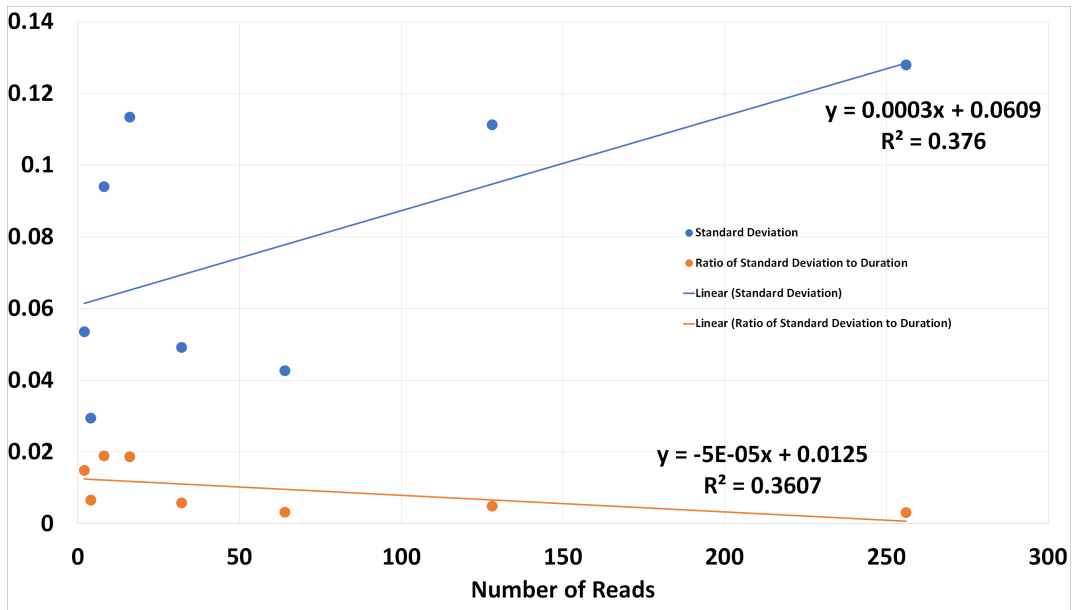


Figure 20. Standard Deviation and Ratio for Variable Reads at Data Read Size 2048

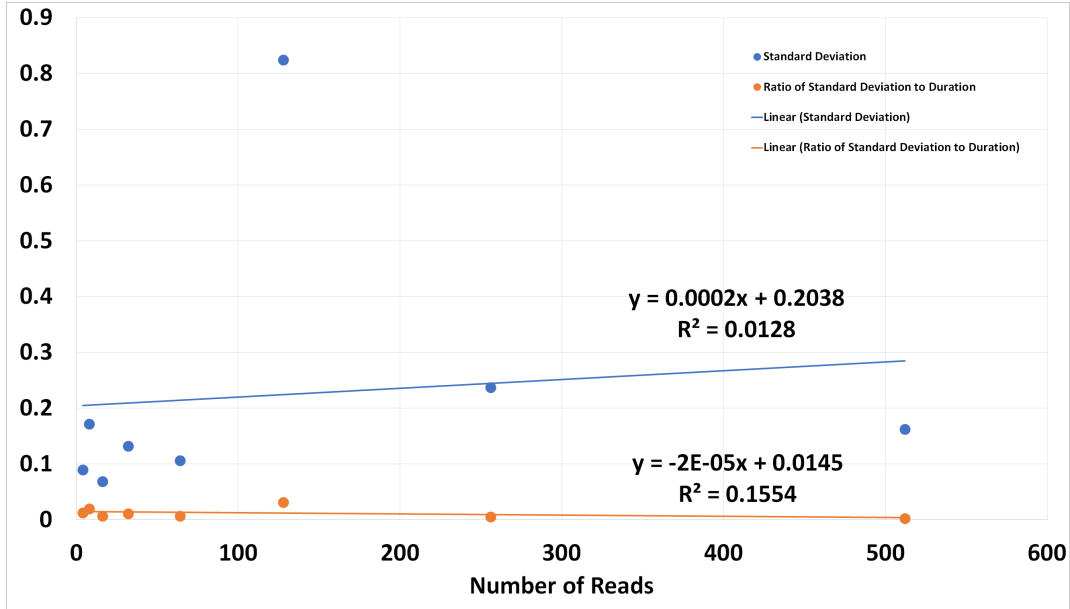


Figure 21. Standard Deviation and Ratio for Variable Reads at Data Read Size 4096

The data was very inconsistent. The standard deviations seemed to have very little to do with the number of reads. The ratio of the standard deviations to the average read time was always under 0.05, meaning that data points falling within one standard deviation of the mean deviated from the mean by $\leq .5\%$ of the mean.

4.7 Summary

4.7.1 Scenario 1

The run time seems to be best described by two discontinuous linear trendlines. Around the 800 Byte mark, there is a severe change in X value from .0018 to .0006. From the second to the third, and third to the fourth blocks respectively the X value smoothly transitions to 0.0004 and 0.0003. As the data read size increases, the cost of each additional byte read consistently decreases until the last block. Seeing as how the last block has an R^2 value of 0.9999, it is safe to assume that the final X value of

0.0003 holds for all values larger than 4096. This gives a data read rate of $3.\bar{3}$ KB/s for read sizes greater than 4096 bytes. The ratio of standard deviation to average duration of reads at a given data size fairly consistently logarithmically reduced as read sizes increased. There were multiple outliers for data read sizes above 4096 bytes, particularly reads at 262144 bytes, which had a ratio of standard deviation to average duration of 0.06. Overall, the largest ratios were for the smallest reads, and those ratios were all less than 0.14.

4.7.2 Scenario 2

Scenario 2 shows that the overhead of each *pdbg* call increases until around 1024 bytes and then stabilizes around 0.1467 seconds. The standard deviations of the data do not seem to be linked to the number of reads, but the individual trials do not deviate very much from the averages. As the number of reads increased, the ratio of standard deviation to average duration generally fell. Additionally, blocks with larger overall data read sizes had smaller ratios. This was in keeping with findings from Scenario 1. The ratios for the block with the overall data read size of 4096 were all below 0.03.

V. Observations and Analysis

5.1 Overview

This chapter presents observations and analysis of the experiments described in Chapter 5. The objective of the experiment was to derive an equation to describe the time to complete the execution of *pdbg* in a program. The formula was expected to take the linear form of Equation 1.

$$z = ax + by + c \tag{1}$$

In Equation 1, x is the amount of data read in bytes, a is the cost in seconds per byte read, y is the number of calls to *pdbg* to read the total amount of data, b is the overhead in seconds per call to *pdbg*, and z is the total cost in seconds of the read in question.

5.2 Analysis

5.2.1 Single Read

The most notable artifact of the data was the inflection point of 800 bytes reducing the slope of the data from 0.0018 to 0.0006 for data reads between 800 and 1000 bytes. The reduction of the slope from 0.0006 to 0.0004 for data reads between 1152 and 3584 bytes, and the subsequent reduction of the slope from 0.0004 to 0.0003 for data reads between 4096 and 524288 bytes were much smoother transitions.

The constants for the linear regression equations provide insight into what the expected overhead size would be. Data read sizes between 8 and 800 bytes were described by Equation 2.

$$y = 0.0018x + 0.1404 \tag{2}$$

Data read sizes between 800 and 1016 bytes were described by Equation 3

$$.y = 0.0006x + 1.0997 \quad (3)$$

Data read sizes between 1152 and 3584 bytes were described by Equation 4

$$.y = 0.0004x + 1.2688 \quad (4)$$

Data read sizes over 4096 bytes were described by Equation 5

$$.y = 0.0003x + 1.4621 \quad (5)$$

For each equation, the constant increases as the slope decreases. This is perfectly logical, as for a portion of the graph where the slope decreases, the y intercept will increase. When the average time for 8 byte data reads is considered, 161.9 milliseconds, this only slightly exceeds what the first regression equation predicts, 154.8 milliseconds. That being said, it can be concluded that the overhead of a single write is no greater than 0.16 seconds.

A constant *pdbg* overhead incurred every time *pdbg* is invoked should not have an impact on the slope of the cost per byte read. This implies that there is some hidden cost heretofore unaccounted for. This cost cannot be a cost associated with each 8 byte read, as that would impose a constant cost as the total data read size increases, and would not therefore affect the overall slope. This hidden cost decreases over time, to the point that it eventually becomes unnoticeable or disappears entirely. This will be commented upon further.

5.2.2 Variable reads

The primary finding from the variable read scenario was that the overhead of individual *pdbg* calls increased until about 1024 bytes and then leveled off. This is likely tied to the same phenomena observed in scenario 1 of the cost of data reads leveling off after 800 bytes. It is not clear if there is a single phenomena that accounts for this discrepancy or instead multiple factors. It would seem that there is one phenomena that drives up the cost of individual *pdbg* reads at the lower range of reads up to 800 bytes, and another that factor that adds costs at a higher level that is eventually drown out when data reads become much larger. The reasoning here is that the overhead of *pdbg* reads stabilizes to an essentially stable value by 1024 bytes of data read, but the data read cost for individual reads continue to decline up to 4096 bytes. The overhead for reads over 800 bytes seems to jump by 10 milliseconds. The reason for this is not clear. However, this cannot account for why the cost per byte of a 1024 byte read is more expensive than a 4096 byte read. This hidden cost is not a constant, but instead seems to be incurred cumulatively as the amount of data read increases. To further analyze this hidden cost, the blocks of data in chapter 4 can be broken down into smaller blocks for analysis.

From 8 to 800 bytes, the cost per byte actually slightly increases with value. For data sizes 8 to 200 and 200 to 400, the cost is 1.8 milliseconds per byte. The cost for reads between 400 and 600 and 600 and 800 is 1.9 milliseconds per byte. For data between 800 and 1016, this trend reverses. For data between 800 and 912, the cost is 0.7 milliseconds. For data between 912 and 1016, the cost is 0.6 milliseconds. For read sizes of 1152 to 2048, the cost is 0.4 milliseconds, which is also the case for read sizes between 2048 bytes and 3584 bytes. Any subdivision of read sizes greater than or equal to 4096 bytes cost 0.3 milliseconds per byte. Therefore, it seems the cost per byte slightly increases from 8 bytes to 800 bytes, then sharply decreases to

7 milliseconds per byte after 800 bytes, and smoothly decreases from there to a final value of 3 milliseconds per byte.

5.2.3 Proposed Equation

For these reasons, a single equation to predict the cost in time for *pdbg* reads cannot be proposed. However, a piecewise linear equation can be proposed. Equation 6 defines the marginal cost in time to complete for the previously discussed ranges of data read.

$$(6) \quad \left\{ \begin{array}{ll} .0018x + .1361 & 8 \leq x \leq 400 \\ .0019x + .1367 & 400 \leq x \leq 800 \\ .0007x + .1467 & 800 \leq x \leq 1024 \\ .0006x + .1467 & 1024 \leq x \leq 2048 \\ .0004x + .1467 & 2048 \leq x \leq 4096 \\ .0003x + .1467 & 4096 \leq x \end{array} \right.$$

Each of these equations account for the cost of reading data in the given range. This means that if one was to read 8192 bytes, the predicted cost would be described by Equation 7.

$$0.0018(400) + 0.0019(400) + 0.0007(224) + 0.0006(1024) + 0.0004(2048) + 0.0003(4096) + 0.1467 \quad (7)$$

The result of Equation 7 is 4.4459 seconds. This is slightly higher than the actual average for 8192 bytes of 4.303, but this works as an approximation. This calculation

is due to data at different ranges running at different costs. Only overhead for the largest category is taken into account for a single call to *pdbg*. With further research, an explanation for this discrepancy could be found that could produce a unified equation.

5.3 Application Scenario - Feasibility of Detecting Malware Processes Using *pdbg*

This section evaluates the feasibility of deploying the program defined in the test case in Chapter III in light of the experimental results from Chapter IV. The test case was built without cost in time in mind. As the various fields are roughly 2000 bytes apart in the *task_struct* struct, a single *pdbg* read of 2000 bytes was used to read each *task_struct*. Using Equation 6, the predicted cost of executing the program from Chapter III, found in A, is described by Equation 8.

$$(0.0018(400) + 0.0019(400) + .0007(224) + 0.0006(1024) + 0.0004(1952) + 0.1467)x \quad (8)$$

For Equation 8, x is the number of *task_structs* read. Using the number 206 for x, where 206 was the average number of processes running on the POWER9 when no user defined processes were running, the cost in time would be roughly 10 minutes 54 seconds. The program defined in Appendix A was executed repeated until an instance where 206 processes were running on the POWER9, and the total time to execute was recorded. It took 10 minutes 47 seconds to complete.

In the experiment from Chapter IV, the closest data read size to 2000 bytes was 2048 bytes, with the second closest being 1920 bytes. The standard deviation for data reads of size 2048 was .0518 seconds. The standard deviation for data reads of size 1920 bytes was .0515 seconds. For data reads of size 2048 bytes, if every *pdbg*

execution was less than one standard deviation below the mean for 206 executions, the most the cumulative run time of said 206 *pdbg* executions could deviate below the mean would be 10.6708 seconds. For data reads of size 1920, the same metric would be 10.609 seconds. The 10 minutes 47 seconds to complete taken by the program was only 7 seconds less than the predicted time. This shows that, at least on average, each *pdbg* execution was within one standard deviation of the measured mean for data reads of that size. This indicates that this equation should hold regardless of normal variance.

In a real world scenario, the number of processes running could be an order of magnitude greater than this, making the deployment of this solution very difficult. If malware had created a hidden process that ran constantly in the background, this solution would still find it. Therefore, even with this cost in time, it would still be feasible for this purpose.

The program could be optimized to execute *pdbg* 5 times to read 8 bytes a piece for each of the `task_struct` fields required by the solution. Equation 9 predicts the time to execute this modified version of the program.

$$(0.0018(8) + 0.1361) * 5)x \tag{9}$$

For Equation 9, x is the number of running processes, 5 is the number of *pdbg* executions per processes, and the rest the time to read 8 bytes. Assuming 206 processes, this would give the time 2 minutes 35 seconds.

This is an improvement, but in terms of the overall cost it is less than an order of magnitude. It would certainly be preferable for the solution to complete in 2 hours rather than 10, but in terms of function it would still only be useful for detecting very long running processes.

Using the method of targeted reads, described by Equation 9, as the denominator,

Equation 10, which calculates the probability of detecting a malware process, can be proposed.

$$\frac{y}{((0.0018(8) + 0.1361) * 5)x} \quad (10)$$

For Equation 10, y is the runtime of a hidden malware process, and x is the number of running processes. The quotient is the probability of the program finding the hidden process. If y is greater than the value for Equation 9, Equation 10 projects the probability of finding the malware as 100%. The rationale for this equation is that if malware is spawned at the time of the initiation of the solution, the chances of finding the malware is the runtime of the malware divided by the runtime of the solution. In short, a program can use *pdbg* to find hidden malware processes, but the chances of it doing so are dependent on the number of running processes and runtime of the hidden malware process.

5.4 Memory Read Using a Microcontroller

Recent research has produced a program running on a different architecture that mimics the functionality of *pdbg*. This program reads memory over an FSI bus from the OCC, except the machine giving commands to the OCC was a microcontroller rather than a BMC. This program was able to achieve a constant data read speed of roughly 0.94 milliseconds per byte with no overhead. While slower than *pdbg* for large data sizes, the lack of overhead makes this approach very attractive for targeted reads as explored in Equation 9. The modified program whose runtime is predicted by Equation 9 could be further modified to run on the microcontroller by using this new method as an alternative to *pdbg*. If it were to do so, assuming a data read rate of 0.94 milliseconds/byte, the cost of this solution would be described by the Equation 11.

$$0.00094(5))x \quad (11)$$

If 206 processes were running in the target memory, where x in Equation 11 is the number of running processes, the run time of the program would be 957 milliseconds, as opposed to 2 minutes 35 seconds using *pdbg*. This makes the solution far more feasible. The probability of finding any given hidden process would be given by the following Equation 12

$$\frac{y}{(0.00094(5))x} \quad (12)$$

Equation 12 is simply an adaptation of Equation 10 to the new architecture. If a hidden process spawned by malware runs for 3 seconds and 2000 processes are running on the machine, this puts the likelihood of finding the malware at 32%. While this hidden process is likely to slip through, longer processes have a much higher probability of being detected.

5.5 Summary

This chapter analyzed the experimental results from Chapter IV, and used them to generate a piecewise equation, Equation 6, to predict the run time of *pdbg* executions. Using Equation 6, the runtime of the program used for the test case in Chapter III, see Appendix A, was predicted by Equation 8. This predicted time was compared to the actual run time of the program, and the difference between the actual result and the predicted result was within the margins that would be expected from examining the standard deviations of experimental results from Chapter IV. This is a strong indicator that Equation 6 is an accurate predictor. A change to said program was also proposed, and Equation 9 was created to predict the time to execute for the modified program. Equation 10 predicts the probability that a hidden malware process will be found by the modified program.

This equation is the ultimate answer to the hypothesis, “*pdbg* can be used to traverse data structures in memory, which will provide information that can expose

when a complete view of memory is being obfuscated from user space on a potentially compromised POWER9 machine”. *pdbg* can be used to traverse data structures in memory. The program utilizing *pdbg* can be used to find information that exposes when a processes in memory is being obfuscated from the view of the user space of a compromised Power9. There is not a guarantee that it will. The probability that a program using *pdbg* to achieve this goal is described by Equation 10. With this caveat, the hypothesis is confirmed.

Additionally, the use of other architecture to access memory could greatly increase the probability of identifying hidden malware processes. This future work will be further commented upon in Chapter VI.

VI. Conclusion

6.1 Overview

This chapter summarizes key aspects of the background, the test case, and experiment completed in this research. It outlines the contributions made, and suggests areas for future work.

6.2 Summary

This research considered two looming threats to modern anti-malware solutions. First, fileless malware is able to load scripts directly into memory and execute them utilizing trusted programs on the target machine. This circumvents malware file signature scanning as it does not write to disk, and by utilizing a trusted program it masks its activity as legitimate. This threat has been met by some anti-malware solutions incorporating monitoring of processes running in memory into their approach. Second, rootkits possess the ability to replace system binaries or change the system call table. This can allow them to prevent programs in user space, such as anti-malware software, from accessing kernel functions to access currently running processes. For this reason, if it is assumed that a machine could be compromised with a rootkit, its ability to access the processes running in memory cannot be trusted. With this problem identified, the objective of this research was to utilize the *pdbg* program to identify when information is being hidden by malware from the user space of a compromised machine, and evaluate the feasibility of deploying *pdbg* for this task.

The test case in Chapter III proposed a program for identifying when aspects of memory are being obfuscated from the user space of a POWER9 machine. If a trusted processor is able to gain access to the memory used by a potentially compromised machine, the process tree detailing the running processes can be exfiltrated to the

trusted machine using *pdbg*. From here, a reconstructed process tree can be compared to the output of a utility ran from user space on the potentially compromised machine. If processes are found by direct memory access from the secure machine that are not found by listing processes on the potentially compromised machine, this is an indication that a rootkit is hiding said processes from the user space of the potentially compromised machine. To demonstrate this, the first step was to execute the command *ps* on the POWER9 processor to list the processes that can be seen from its user space. This output was then transferred to the BMC, the processor assumed secure. This output was read into a Python program, which can be found in Appendix A. This program then used *pdbg* to traverse the process tree in the memory used by the POWER9. When it completed its traversal, the processes retrieved by means of *pdbg* were compared to the processes output by the user space program on the POWER9. When processes not identified by the POWER9 were identified by the program they were flagged as hidden processes. This demonstrates the ability of indirect access to memory through the use of *pdbg* to provide an accurate view of running processes in the memory of a compromised machine when said processes are being hidden from programs on said compromised machine.

The experiment performed in Chapter IV provided insight into the latency of *pdbg*. By running numerous trials reading different sizes of data, it was able to estimate the cost in seconds/byte at different sizes of data read. By running multiple trials that used a variable amount of calls to *pdbg* to read a set amount of data, it was able to isolate the overhead cost of a call to *pdbg*.

In Chapter V, the results from Chapter IV were used to create Equation 6, which predicts the latency of executing *pdbg* for a given data read size. A modification to the program from Chapter III was proposed, and Equation 6 was used to predict its runtime for a given number of processes running on the target POWER9. Addition-

ally, Equation 10 was generated which calculates the probability that the program will find a hidden malware process for a given number of processes running on the target POWER9 and time left before the termination of the malware process at the time that program begins. This equation answered in the affirmative the hypothesis, “*pdbg* can be used to traverse data structures in memory, which will provide information that can expose when a complete view of memory is being obfuscated from user space on a potentially compromised POWER9 machine”. *pdbg* can be used to traverse data structures in memory, and a program utilizing *pdbg* can be used to provide information that exposes when a processes in memory is being obfuscated from the view of the user space of a compromised Power9. The probability that the program used in the test case will succeed in this task is described by Equation 10. With this caveat, the hypothesis is confirmed.

6.3 Contributions

The first contribution of this research was the demonstration of remote traversal of the process tree in the memory used by a POWER9 processor with the use of *pdbg*. This functionality was shown to have the capacity to detect processes hidden from the user space of a compromised POWER9 machine. Secondly, this research provided experimental metrics to predict the cost of the use of *pdbg* with varying data read sizes. These metrics were used to assess the feasibility of utilizing *pdbg* to monitor for diverse sets of malware threats.

6.4 Future Work

The first area of future work is the exploration of new architecture with which remotely traverse memory. A mechanism which could consistently read data at a faster rate would dramatically increase the odds of detecting short running fileless

malware, as well as decrease consumption of system resources. Secondly, additional data structures in memory, such as the *dentry* tree, could be explored. The *dentry* tree is a tree structure that contains each file system entry entered into memory. This tree could be traversed in a similar method to the process tree in order to hunt for files that have been hidden from the user space of a target machine. While this is a much larger structure than the process tree, it is also less volatile, and therefore a greater runtime would be less prohibitive. Additionally, a more efficient architecture would also reduce runtime. Very specific to the Talos II architecture, more research could be done to determine the source of the discontinuous cost of data read after 800 bytes. Research in this area could both help to better define expectations for *pdbg* performance, and to potentially make improvements to *pdbg* itself.

6.5 Conclusion

This research demonstrated analyzing data structures in the memory of a potentially compromised machine from a secure external system as a means to detect when memory is being obfuscated from the view of the user space of the targeted machine. It analyzed the cost of the use of *pdbg* for a range of data read sizes. The hypothesis of this research, “*pdbg* can be used to traverse data structures in memory, which will provide information that can expose when a complete view of memory is being obfuscated from user space on a potentially compromised POWER9 machine”, was confirmed by the test case. Informed by the experimental results of Chapter IV, Equation 10 predicts the probability that an optimized version of the program used in the test case will detect a hidden malware process. Future work that expands this methodology to new architectures in which the traversal time of the process tree would be significantly reduced as an alternative to the utilization of *pdbg* in the Talos II architecture will greatly increase probability of detecting hidden malware.

Appendix A. Process Tree Traversal Script

```
1
2 #!/usr/bin/python
3 from __future__ import print_function
4 import subprocess
5 import array
6 import sys
7 import string
8 import re
9 import time
10 from datetime import datetime
11
12 now=datetime.now()
13 byteCounter = 0
14 print(now.strftime("%H:%M:%S"))
15 class Pcb:
16     name = ""
17     timeRunning = 0
18     pid = -1
19     level = 0
20     readTime = 0
21     flag = 0
22     def __init__(self):
23         self.children = []
24
25 class procEntry:
26     def __init__(self):
27         self.pid = -2
28         self.mo = "null"
29         self.day = -1
30         self.hr = -1
31         self.min = -1
32         self.sec = -1
33         self.yr = 0
34         self.name = "hello"
35
36 def translate(hexNum):
37     return int(hexNum, 16)
38
39
40 proclistCount = 0
41 proclist = []
42 pcbHiddenList = []
43 pcbFlaggedList = []
44 with open('psmalicious.txt', 'r') as f:
45     for line in f:
46         line2 = re.sub(' +', ' ', line)
47         chunks = line2.split(' ')
48         print("hello")
49         print(chunks)
50         if chunks[1] != "PID":
51             print("")
52             entry = procEntry()
53             offset = 0
54             if len(chunks[0])>4:
55                 offset = 1
```

```

56         entry.pid = int(chunks[1-offset])
57         entry.mo = chunks[3-offset]
58         entry.day = chunks[4-offset]
59         entry.hr = int(chunks[5-offset][0:2])
60         entry.min = int(chunks[5-offset][3:5])
61         entry.sec = int(chunks[5-offset][6:8])
62         entry.yr = int(chunks[6-offset])
63         entry.name = chunks[7-offset]
64         print(entry.pid, entry.mo, entry.day, entry.hr,
65               entry.min, entry.sec, entry.yr, entry.name)
66         proclstCount = proclstCount + 1
67         proclst.append(entry)
68
69     checkCount = 0
70     def checkPcb(thisPcb):
71         global checkCount
72         checkCount = checkCount + 1
73         found = 0
74         for x in proclst:
75             if thisPcb.pid == x.pid:
76                 print(thisPcb.name + " " + " is " + str(x.pid))
77                 return 1
78         if found == 0:
79             print(thisPcb.name + " not found ")
80             pcbFlaggedList.append(thisPcb)
81             return 0
82         else:
83             return 1
84
85
86
87
88     levelsCount = []
89     init_task = "0x1070a00"
90     count = 0
91     depth = 1
92
93     def traverse(currentPcb, pdbgInput, currAddr, parentAddr,
94                 parentlevel):
95         print("Curr addr ", currAddr)
96         global count
97         global depth
98         global byteCounter
99         count = count + 1
100        then = time.time()
101        hexdump = subprocess.check_output(pdbgInput, shell=True)
102        byteCounter += 4000
103        now = time.time()
104        currentPcb.readTime = now - then
105        print("time read " + str(currentPcb.readTime))
106        name = hexdump[3776:3806].decode("hex")
107        name2 = name.encode("ascii", "ignore")
108        currentPcb.name = name2.decode()
109        currentPcb.level = parentlevel + 1
110        if currentPcb.level > depth:
111            depth = currentPcb.level
112        print("name is ", currentPcb.name)

```

```

112     thePid = hexdump[2966] + hexdump[2967]
113     for i in range(1,4):
114         thePid = thePid + hexdump[2966 - 2*i] + hexdump[2967 -
115             2*i]
116     hexPid = translate(thePid)
117     print("pid ", hexPid)
118     currentPcb.pid = hexPid
119
120     hextime = hexdump[3550] + hexdump[3551]
121     for i in range(1,8):
122         hextime = hextime + hexdump[3550 - 2*i] + hexdump[3551
123             - 2 * i]
124         print(hextime)
125     hextime2 = int(hextime, 16)
126     print("correct hextime " + str(hextime2))
127     print(hextime.decode("hex"))
128     currentPcb.timeRunning = hextime2
129
130     Next = hexdump[3008:3080]
131     print(Next)
132     siblingLast = hexdump[3051]
133     siblingAddr = siblingLast + hexdump[3052]
134     for i in range(1,7):
135         siblingAddr = siblingAddr + hexdump[3052 - 2*i] +
136             hexdump[3052 - 2*i + 1]
137
138     print("siblingAddr " + siblingAddr)
139
140     childLast = hexdump[3019]
141     childAddr = childLast + hexdump[3020]
142     for i in range(1,7):
143         childAddr = childAddr + hexdump[3020 - 2*i] + hexdump
144             [3020 - 2*i + 1]
145
146     print("ChildAddr " + childAddr)
147     checkEndOffset = "5E0"
148     fiveFOffset = "5F0"
149
150     childCheckhex = int(checkEndOffset, 16)
151     siblingCheckhex = int(checkEndOffset, 16)
152     fhexOffset = int(fiveFOffset, 16)
153
154     chexAddr = int(childAddr, 16)
155     shexAddr = int(siblingAddr, 16)
156
157     newChildCheck = chexAddr - childCheckhex
158     newSiblingCheck = shexAddr - siblingCheckhex
159
160     newccAddr = "0x%x" % newChildCheck
161     newcsAddr = "0x%x" % newSiblingCheck
162     newcAddr = "0x%x" % newChild
163     newsAddr = "0x%x" % newSibling
164
165     print("New caddr is now " + newcAddr + " sib " + newsAddr)

```

```

166     cinput = "pdbg -p0 getmem " + newcAddr + " 4000 | hexdump
        -ve '1/1 \".2x\"',"
167     print(cinput)
168     sinput = "pdbg -p0 getmem " + newsAddr + " 4000 | hexdump
        -ve '1/1 \".2x\"',"
169     print(sinput)
170     print("CCinput is " + newccAddr + " curraddr " + currAddr)
171     newPcb.flag = checkPcb(newPcb)
172     print(newPcb.name + " flag is " + str(newPcb.flag))
173     if newccAddr != currAddr:
174         print(newccAddr + " != " + currAddr)
175         done = False
176         newPcb = Pcb()
177         siblingInput, siblingAddr = traverse(newPcb, cinput,
            newcAddr, currAddr, currentPcb.level)
178         currentPcb.children.append(newPcb)
179         while done == False:
180             if siblingInput != "done":
181                 childsSiblingPcb = Pcb()
182                 print("current sibling infor is Addr of " +
                    siblingAddr + " input of " + siblingInput)
183                 siblingInput, siblingAddr = traverse(
                    childsSiblingPcb, siblingInput, siblingAddr,
                    currAddr, currentPcb.level)
184                 checkReturn = checkPcb(childsSiblingPcb)
185                 childsSiblingPcb.flag = checkReturn
186                 currentPcb.children.append(childsSiblingPcb)
187             else:
188                 done = True
189         else:
190             print("noChildren")
191
192     print("newcsAddr is " + newcsAddr + " parentAddr is " +
        parentAddr)
193     if newcsAddr != init_task and newcsAddr != parentAddr and
        newsAddr != currAddr:
194         print("new siblingaddr is " + newsAddr + " parnt is " +
            parentAddr)
195         print("Returning Sibling Info of "+currentPcb.name + "
            " + newsAddr + " " + sinput)
196         return(sinput, newsAddr)
197
198     else:
199         return("done", "done")
200
201 def truncator(name):
202     count = 0
203     for c in name:
204         if c not in string.ascii_letters and c not in string.
            digits and c != '/':
205             return name[:count]
206         else:
207             count += 1
208     return name
209 treeCount = 0
210 def treePrinter(currPcb, length):
211     count = 0
212     global treeCount

```

```

213     if currPcb.pid == 0:
214         print(str(currPcb.pid) + ":" + currPcb.name)
215     else:
216         space = ""
217         print(space.ljust(length, ), end='')
218         print(str(currPcb.pid) + ":" + currPcb.name)
219         treeCount = treeCount + 1
220         length = length + len(str(currPcb.pid)) + 1 + len(
            currPcb.name)
221     while count < len(currPcb.children):
222         treePrinter(currPcb.children[count],length)
223         count += 1
224
225 def checkTime(originalPcb, currPcb):
226     if(currPcb.timeRunning > originalPcb.timeRunning and
        originalPcb.flag == 0):
227         print(currPcb.name + " " + str(currPcb.pid) + " " +
            str(currPcb.timeRunning) + " started after
            original Pcb " + originalPcb.name + " " + str(
            originalPcb.pid) + " " + str(originalPcb.
            timeRunning) + " but is also not on the list")
228         if(currPcb.timeRunning > originalPcb.timeRunning
            and originalPcb.flag == 1):
229             print(currPcb.name + " " + str(currPcb.pid) + " " +
                str(currPcb.timeRunning) + " is before original
                Pcb " + originalPcb.name + " " + str(
                originalPcb.pid) + " " + str(originalPcb.
                timeRunning))
230         return 0
231     else:
232         count = 0
233         while count < len(currPcb.children):
234             retVal = checkTime(originalPcb,
                currPcb.children[count])
235             if retVal == 1:
236                 count += 1
237             else:
238                 return 0
239         return 1
240 sum = 0
241 count =1
242
243 def timeAvg(currPcb):
244     global sum
245     global count
246     sum += currPcb.time
247     count += 1
248     i = 0
249     while i < len(currPcb.children):
250         timeAvg(currPcb.children[i])
251
252 count -=1
253 print("Sum " + str(sum) + " count " + str(count))
254 newAddr = "0x1070a00"
255 input = "pdbg -p0 getmem " + newAddr + " 4000 | hexdump -ve
        '1/1 \".2x\"'"
256 pcbCount = 1
257 p0 = Pcb()

```

```

258 then = time.time()
259 traverse(p0, input, newAddr, init_task, 0)
260 now = time.time()
261 traverseTimer = now - then
262
263 print("count is " + str(count))
264 treePrinter(p0,0)
265
266 print('Flagged Procs')
267 for procEntry in pcbFlaggedList:
268     print(procEntry.pid)
269     if checkTime(procEntry, p0) == 0:
270         pcbHiddenList.append(procEntry)
271
272 print('Hidden Procs')
273 for procEntry in pcbHiddenList:
274     print(procEntry.pid)
275
276 print("proclistCount ", str(proclistCount))
277 print("tree count was ", str(treeCount))
278 print("check count was ", str(checkCount))
279 print("Bytes read " + str(byteCounter))
280 print("Time to read " + str(traverseTimer))

```

Bibliography

1. D. Patten, “The Evolution to Fileless Malware,” p. 13, 2017. [Online]. Available: <https://infosecwriters.com/Papers/DPatten.Fileless.pdf> [Accessed: May 12 2020]
2. ARS Staff, “Intel, Microsoft to use GPU to scan memory for malware Ars Technica.” [Online]. Available: <https://arstechnica.com/gadgets/2018/04/intel-microsoft-to-use-gpu-to-scan-memory-for-malware/> [Accessed: March 5 2020]
3. D. Wampler and J. Graham, “A Method For Detecting Linux Kernel Module Rootkits,” *IFIP International Federation for Information Processing*, vol. 242, pp. 107–116, 2007.
4. C. Oprea, M. Chechicheş, and A. Năndrean, “Locality-sensitive hashing optimizations for fast malware clustering,” in *2014 IEEE 10th International Conference on Intelligent Computer Communication and Processing (ICCP)*. IEEE, 2014, pp. 97–104.
5. R. S. S. Obermeier and A. Hristova, “Securing industrial automation and control systems using application whitelisting,” *Proceedings of the 2014 IEEE Emerging Technology and Factory Automation (ETFA)*, pp. 1–4, 2014.
6. A. Bazhaniuk, “BMC, IPMI, and the Data Center Underbelly - Eclipsium,” 2018. [Online]. Available: <https://eclipsium.com/2018/08/09/bmc-ipmi-and-the-data-center-underbelly/> [Accessed: May 23 2020]
7. L. Khemani and S. Sudarshan, “Switching of host network traffic through base-board management controller (bmc),” 2018.
8. R. Wilkins and B. Richardson, “Uefi secure boot in modern computer security solutions,” in *UEFI Forum*, 2013.
9. S. Zaharoff, “OpenPower Foundation,” 2015. [Online]. Available: <https://searchdatacenter.techtarget.com/definition/OpenPower-Foundation> [Accessed: November 28 2020]
10. T. Rosedahl, “OpenPOWER Summit 2019: New Innovations in Power and Performance Management,” 2019. [Online]. Available: <https://www.youtube.com/watch?v=pqhI1rVwayQ> [Accessed: Feb 20 2020]

11. OpenPOWER Foundation, “OpenFSI Specification:Field Replaceable Unit Service Interface,” 2016. [Online]. Available: <http://cdn.openpowerfoundation.org/wp-content/uploads/resources/OpenFSI-spec-100/OpenFSI-spec-20161212.pdf> [Accessed: Feb 13 2020]
12. M. Broyles, “OCC Firmware Interface Specification for Open Power,” 2016. [Online]. Available: https://raw.githubusercontent.com/open-power/docs/master/occ/OCC_P9_FW_Interfaces.pdf [Accessed: Feb 26 2020]
13. S. Kumar *et al.*, “An emerging threat fileless malware: a survey and research challenges,” *Cybersecurity*, vol. 3, no. 1, pp. 1–12, 2020.
14. A. Bunten, “Unix and linux based rootkits techniques and countermeasures,” in *16th Annual First Conference on Computer Security Incident Handling, Budapest*, 2004.
15. J. Levine, J. Grizzard, and H. Owen, “A methodology to detect and characterize kernel level rootkit exploits involving redirection of the system call table,” in *Second IEEE International Information Assurance Workshop, 2004. Proceedings.* IEEE, 2004, pp. 107–125.
16. R. Riley, X. Jiang, and D. Xu, “Multi-aspect profiling of kernel rootkit behavior,” in *EuroSys ’09: Proceedings of the 4th ACM European conference on Computer systems*, 2009, pp. 47–60.
17. C. Siebenmann, “The technical effects of being an out of tree Linux kernel module,” 2013. [Online]. Available: <https://utcc.utoronto.ca/~cks/space/blog/linux/TechnicalNonGPLKernelModules> [Accessed: May 22 2020]
18. R. O’Neill, *Learning Linux binary analysis: uncover the secrets of Linux binary analysis with this handy guide.* Packt Publ., 2016.

REPORT DOCUMENTATION PAGE					Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY) 25-03-2021		2. REPORT TYPE Master's Thesis			3. DATES COVERED (From — To) Sept 2019 — Mar 2021	
4. TITLE AND SUBTITLE Remote Monitoring of Memory Data Structures for Malware Detection in a Talos II Architecture				5a. CONTRACT NUMBER		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Willburn, Robert A., 2d Lt, USAF				5d. PROJECT NUMBER 21G171		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology Graduate School of Engineering an Management (AFIT/EN) 2950 Hobson Way WPAFB OH 45433-7765					8. PERFORMING ORGANIZATION REPORT NUMBER AFIT-ENG-MS-21-M-094	
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Research Laboratory 2241 Avionics Circle WPAFB OH 45433-7765 Attn: Pranav Patel COMM 937-656-9045 Email: pranav.patel.2@us.af.mil					10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/Ryda	
					11. SPONSOR/MONITOR'S REPORT NUMBER(S)	
12. DISTRIBUTION / AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A: APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.						
13. SUPPLEMENTARY NOTES						
14. ABSTRACT New forms of malware, namely fileless malware and rootkits, pose a threat to traditional anti-malware. In particular, Rootkits have the capacity to obscure the present state of memory from the user space of a target machine. If this happens, anti-malware running in the user space of an affected machine cannot be trusted to operate properly. To combat this threat, this research proposes the remote monitoring of memory from a second, secure processor running OpenBMC, serving as a baseboard management controller for a POWER9 processor, which is assumed vulnerable to exploitation. The baseboard management controller includes an application called <i>pdbg</i> , used for debugging POWER9 processors. This application allows for both reading and writing to registers and system memory of the POWER9 processor from the baseboard management controller directly via the field replaceable unit support interface bus. This research developed a program to run on the baseboard management controller which utilizes <i>pdbg</i> to traverse the process tree active in the memory of the POWER9 processor. By traversing this data structure, it can view the entire process tree remotely, verifying whether information in memory is being hidden from user space on the POWER9.						
15. SUBJECT TERMS Anti-Malware, Secure Processing, Remote Memory Monitoring						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT UU	18. NUMBER OF PAGES 86	19a. NAME OF RESPONSIBLE PERSON Dr. Scott Graham, AFIT/ENG	
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U			19b. TELEPHONE NUMBER (include area code) (937) 255-6565 x4581; scott.graham@afit.edu	