EVALUATING THE USE OF BOOT IMAGE ENCRYPTION ON TALOS II ARCHITECTURE

THESIS

Calvin M. Muramoto, Second Lieutenant, USAF

AFIT-ENG-MS-22-M-049

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-22-M-049

EVALUATING THE USE OF BOOT IMAGE ENCRYPTION ON 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 Cyber Operations

Calvin M. Muramoto, B.S.

Second Lieutenant, USAF

March 2022

DISTRIBUTION STATEMENT A
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AFIT-ENG-MS-22-M-049

EVALUATING THE USE OF BOOT IMAGE ENCRYPTION ON TALOS II ARCHITECTURE

THESIS

Calvin M. Muramoto, B.S.
Second Lieutenant, USAF

Committee Membership:

Scott R. Graham, Ph.D.
Chair

Lt Col James W. Dean, Ph.D.
Member

Stephen J. Dunlap, M.S.
Member

AFIT-ENG-MS-22-M-049

Abstract

Sensitive devices operating in unprotected environments are vulnerable to competitors or bad actors conducting hardware attacks like reverse engineering and side channel analysis. This represents a security concern because the root of trust can be invalidated through boot firmware manipulation. For example, boot data is rarely encrypted and typically travels across an accessible bus like the Low Pin Count (LPC) bus, allowing data to be easily intercepted and possibly manipulated during system startup. The flash chip storing the boot data can also be removed from these devices and examined to reveal detailed boot information.

Firmware that developers deem to contain sensitive code or perform innovative operations needs to be protected before being flashed onto the boot flash chip. This paper details an implementation of encrypting a section of the boot image and decrypting it during the Initial Program Load (IPL) of the Talos II. During power-on, the encrypted image travels across the LPC bus into the POWER9 Level3 cache and is decrypted in the processor. This proves that it is possible to prevent adversaries from interfering with the IPL flow or obtaining details on firmware from the flash chip. This method is also expanded upon to create a process of securing the firmware decryption keys. The boot image encryption method is implemented with multiple levels of encryption and an analysis of their efficiency is conducted to determine the performance impact for each algorithm.

AFIT-ENG-MS-22-M-049

This work is dedicated to my brother and my parents for their support, encouragement, and love.

Table of Contents

Page

Abstract . . . iv

Dedication . . . v

List of Figures . . . ix

List of Tables . . . x

List of Acronyms . . . xi

I. Introduction . . . 1

1.1 Background and Motivation . . . 1
1.2 Problem Statement . . . 2
1.3 Research Objectives . . . 3
1.4 Hypothesis . . . 4
1.5 Approach . . . 5
1.6 Contributions . . . 5
1.7 Organization . . . 6

II. Background and Related Work . . . 8

2.1 Overview . . . 8
2.2 Talos II Architecture . . . 8
2.3 PNOR Image . . . 11
2.4 Initial Program Load . . . 12
2.4.1 1st Stage Bootloader . . . 13
2.4.2 Hostboot . . . 15
2.4.3 Skiboot . . . 16
2.5 Secure Boot . . . 17
2.6 Secure Key Storage . . . 19
2.7 Encryption Schemes . . . 19
2.7.1 XOR . . . 19
2.7.2 SPECK . . . 20
2.7.3 Advanced Encryption Standard . . . 20
2.8 Related Work . . . 21
2.8.1 Secure Firmware Updates with AES Encryption . . . 21
2.8.2 U-Boot Image Encryption . . . 22
2.9 Summary . . . 23

Page

III. Implementation Discussion 24

3.1 Implementation 24
3.2 Secure Key Management Process 29
3.3 Application Scenarios 32

IV. Experimental Setup and Methodology 33

4.1 Objective 33
4.2 Assumptions 33
4.3 Control Variables 34
4.4 Independent Variables 35
4.5 Response Variables 35
4.6 Performance Evaluation 36
4.7 Experimental Setup 36
4.7.1 Micro Controller Setup 37
4.8 Summary 39

V. Observations and Analysis 40

5.1 Overview 40
5.2 Secure Key Management Analysis 40
5.3 PNOR Image Size 41
5.4 Performance Experiment Results 43
5.4.1 Performance Impact of Boot Image Encryption 43
5.4.2 Comparison of Encryption Methods 47
5.4.3 Encryption Ratio 49
5.5 Challenges 50
5.6 Summary 52

VI. Conclusion 53

6.1 Overview 53
6.2 Summary 53
6.3 Research Contributions to Hardware Security 54
6.4 Future Work 55
6.4.1 Full boot image encryption 55
6.4.2 Compatibility with IBM Secure Boot 56
6.4.3 Application to Intel or AMD systems 57
6.5 Conclusion 60

Appendix A. XOR100 Decryption Firmware 61

Appendix B. XOR100 Encryption Script 62

Page

Appendix C. SPECK100 Decryption Firmware 63

Appendix D. SPECK100 Encryption Script 64

Appendix E. AES100 Decryption Firmware 65

Appendix F. AES100 Encryption Script 66

Appendix G. Talos Control Firmware 67

Appendix H. UDP Logger Firmware 71

Appendix I. Experiment Script 79

Appendix J. HBBL XOR Decryption Firmware 82

Appendix K. HBBL XOR Script 83

Appendix L. HBBL Speck Decryption Firmware 84

Appendix M. HBBL AES Compilation Fail 85

Bibliography 86

List of Figures

Figure Page

1 Detailed Hardware Layout of the POWER9 Secure Boot Environment [1] 10

2 High-Level View of the PNOR Image [1] 12

3 OpenPOWER Firmware Boot Flow [2] 13

4 Detailed OpenPOWER Boot Flow [1] 14

5 U-Boot Firmware Encryption Setup [3] 22

6 Hostboot Malloc Code 26

7 Hostboot Decrypt Code 27

8 Proof of Concept Decryption 29

9 Stock Talos II Setup 37

10 Microcontroller Setup 38

11 Hostboot Execution Time 45

12 ISTEP 20.1 Execution Time 46

13 Skiboot Decompression Time 47

14 Skiboot Decryption Time 48

15 Skiboot Decryption Time by Encryption Ratio 50

16 Encryption ANOVA Test 51

17 Intel Boot Flow [4] 58

List of Tables

Table Page

1 Decryption Firmware Size by Encryption Type 42

2 Breakdown of Boot Times by Encryption 44

List of Acronyms

AES Advanced Encryption Standard

AIK Attestation Identity Key

ANOVA Analysis of Variance

BIOS Basic Input/Output System

BMC Baseboard Management Controller

CPU Central Processing Unit

ECC Error Correction Code

ECDSA Elliptic Curve Digital Signature Algorithm

FPGA field-programmable gate array

FSI Fast Serial Interface

HBB Hostboot Base

HBBL Hostboot Bootloader

HBI Hostboot Extended Image

HBRT Hostboot Runtime

IOT Internet of Things

IPL Initial Program Load

IPMI Intelligent Platform Management Interface

L3 Level 3

LPC Low Pin Count

MBR Master Boot Record

NSA National Security Agency

OCC On Chip Controller

OEM Original Equipment Manufacturer

OPAL OpenPOWER Abstraction Layer

OS Operating System

OTP One-Time Pad

OTPROM One Time Programmable Read Only Memory

P9 POWER9

PCIE Peripheral Component Interconnect Express

PIB Pervasive Interconnect Bus

PKI Public Key Infrastructure

PMU Phasor Measurement Unit

POST Power-On Self-Test

PPE Programmable PowerPC-lite Engine

PSP Platform Security Processor

RAM Random Access Memory

ROM Read-Only Memory

SBE Self Boot Engine

SEEPROM Serial Electrically Erasable Programmable Read Only Memory

SLW Sleep Winkle

TOC Table of Contents

TPM Trusted Platform Module

UDP User Datagram Protocol

UEFI Unified Extensible Firmware Interface

VFSRP Virtual File System Resource Provider

EVALUATING THE USE OF BOOT IMAGE
ENCRYPTION ON TALOS II ARCHITECTURE

I. Introduction

1.1 Background and Motivation

As the complexity of cyber attacks increases and evolves, the demand for increased security in computer systems also rises. In particular, hardware attacks have become an increasing concern in recent years. Under the assumption that the adversary does not have physical access, device hardware is not normally designed to be resistant to physical attacks. To counter this threat, trusted computing has become a focus for computing system development with security enhancements, such as secure boot. Although the authentication capabilities provided by secure boot can prevent adversaries from loading malicious firmware on a system, no known system also incorporates encryption of the boot firmware as a security measure.

By greatly increasing the cost to reverse engineer boot operations, encrypting boot firmware will help prevent outsiders from discovering vulnerabilities on computer systems. Tamper detection checks are often employed during the boot sequence of a system to catch intruders before the operating system starts. If an outsider is able to observe the boot instructions executed during start up, they may be able to avoid tamper detection to manipulate and gain access to the system. Boot firmware is rarely encrypted under the assumption that the adversary does not have physical access to the hardware. To alleviate this problem, the boot firmware could be, and perhaps should be, encrypted when stored in memory and decrypted in the processor

when the instructions are needed.

Secure boot has become a standard to improve hardware security, and works by ensuring that only signed firmware is run on the system. Although it can prevent unauthorized firmware from executing on a system processor, it cannot protect against hardware physical attacks like chip substitution and bus traffic recording [5]. The focus of this research is to prevent these types of attacks through boot image encryption. There have also been development efforts to encrypt bootloader firmware which can then be decrypted during startup in microcontroller platforms [6]. However, microcontroller bootloaders are very simple and typically employ only two stages to start up. This research focuses on expanding the encrypted microcontroller boot firmware concept to a workstation supporting a powerful processor such as the POWER9 (P9).

Although this research implemented boot image encryption on the Talos II, it could also be applied to any system that utilizes a multi stage bootloader. For example, Intel and AMD both include bootloaders inside the Basic Input/Output System (BIOS) or Unified Extensible Firmware Interface (UEFI) which functions similarly to the firmware in the Talos II. Developing the encryption implementation for x86 architectures would require the ability to alter the firmware and change the IPL. Expanding the implementation of boot firmware encryption from the Talos II to other architecture systems should be straight forward as long as the IPL flow is similar.

1.2 Problem Statement

As described in the previous section, currently there is no system in place that protects the boot firmware stored on computing systems through encryption. Although secure boot helps improve hardware security, it does not prevent adversaries from conducting hardware physical attacks. Secure boot uses shared key encryption

which provides integrity protection but is not designed for tamper protection. Using unencrypted boot firmware leaves the code open to be viewed and tampered with by adversaries. Encrypting the boot firmware will protect proprietary information on commodity hardware through confidentiality, preventing outsiders from accessing the proprietary information and providing an additional layer of deterrence against tampering attempts. Implementing a method of encrypting and decrypting firmware in the processor also requires a method of secure key storage which needs to be resolved with the boot firmware encryption method.

1.3 Research Objectives

As part of this research effort the following objectives are identified below:

- Understand the motivation behind encrypting the boot image and how to accomplish this given the properties of the hardware.
- Understand the structure of the Talos II IPL flow and how data traverses the various bus interfaces during the boot sequence.
- Understand the memory and processing limitations of the Talos II during the early stages of the IPL.
- Evaluate OpenPOWER firmware to determine appropriate locations in the IPL to decrypt firmware.
- Design and implement the boot firmware encryption method with several encryption schemes.
- Configure an experiment with multiple factors to model and collect timing data regarding the firmware decryption code.

- Evaluate the performance of the boot firmware encryption implementation and the implications in securing the PNOR image.
- Understand the functionality and properties of secure boot.
- Design and implement a method of securing the firmware decryption keys.

The questions to be answered by this research in order to meet the aforementioned objectives are as follows:

- Is it feasible to decrypt firmware during the IPL?
- Is it possible to securely store the decryption keys within the P9?
- How much memory space is available during each boot stage to store the decryption firmware?
- What are the constraints and performance limitations of boot firmware encryption?
- Do the benefits of boot firmware encryption outweigh the performance impacts and memory consumption?

1.4 Hypothesis

The hypothesis of this research is that it is feasible to utilize boot firmware encryption on vulnerable computer systems to protect the boot image from adversarial reverse engineering efforts. Boot firmware decryption during the boot sequence will theoretically require the hardware to alter the firmware data immediately before it is executed. It also speculates that the additional layer of security will not cause a significant performance impact to the system during the IPL.

1.5 Approach

This approach consists of implementing boot firmware encryption on the Talos II to protect proprietary information stored on the system. Locations in the IPL have to be identified where the encrypted boot firmware can be decrypted in the processor cache. Once a section of firmware is selected, the decryption code is inserted into the firmware to decrypt the following section of the IPL. Several encryption schemes are selected to test the performance of the boot firmware decryption during the system boot up. Once the boot firmware encryption experiment was executed under various configurations, performance data is collected and analyzed. The boot firmware produces logging data to the console allowing the boot timing to be measured. Other findings and implications revealed during the process are documented. The method of secure key storage can also be solved by utilizing the secure properties of the system architecture.

1.6 Contributions

The contributions of this thesis to the field of hardware security are as follows:

- **Boot Firmware Encryption:** It demonstrates a successful implementation of boot firmware encryption with a variety of encryption methods. It provides a framework for full boot image encryption and possible use on other computer systems like Intel and AMD.

- **Analysis of Decryption Firmware Memory Requirements:** It details the memory required for each decryption algorithm and discusses the most efficient options for a variety of scenarios.

- **Secure Key Storage Management:** It demonstrates a successful implementation of a secure key storage management approach. This implementation

creates the framework to encrypt the entire PNOR image and store the keys securely within the processor. Since only firmware is added to the system, additional hardware like a One-Time Pad storage chip is not required.

- **Performance Analysis:** It demonstrates an approach for measuring the performance of various decryption algorithms during the Talos II boot sequence. The measurements are collected with varying encryption ratios to identify the performance limitations with each encryption algorithm.
- **Qualitative Analysis:** It lists design considerations, challenges, and potential vulnerabilities of the boot firmware encryption implementation described.

1.7 Organization

This thesis is organized as follows. Chapter II provides background information required to understand the complexity of boot firmware encryption and decryption. It introduces the Talos II architecture, firmware, IPL, and encryption algorithms used in this research. Details on secure boot and secure key storage management are also presented. This chapter also presents research related to this work and identifies the areas requiring further research.

Chapter IV lists the assumptions, control factors, and response variables of this research. The process of developing the boot firmware encryption is discussed, including specific developmental challenges encountered in this research. A description of the methodology for conducting the boot firmware encryption performance analysis is presented in this chapter along with an overview of the secure key storage management process in Section 3.2.

Chapter V presents the results and analysis of the experiments described in Chapter IV. An analysis of the secure key storage management implementation is discussed,

followed by presentation of the firmware encryption results. The memory space allocated for each firmware is analyzed along with the performance impact of the boot image encryption, encryption algorithms, and encryption ratio.

Finally, Chapter VI concludes with a summary of the work presented and the contributions to the field. In addition, recommendations for those utilizing similar tools or frameworks are presented. Future work areas for this research which involve improvements to the simulation environment, upgraded components, and other performance data collection tools are also discussed.

II. Background and Related Work

2.1 Overview

This chapter provides the background information required to understand the challenges of boot firmware encryption. It covers the hardware and firmware of the Talos II along with an overview of the boot flow. It builds off previous information to introduce secure boot and the problem of secure key storage. Encryption algorithms that are used in this thesis are also introduced with a description of the use case and strengths of each algorithm. The literature surrounding boot firmware encryption related work is reviewed to supply perspective to the problem this research attempts to solve. The related work also provides possible directions to take for the problem of encrypting low level boot firmware.

2.2 Talos II Architecture

The Talos II workstation was used in this research effort to develop and test the boot encryption implementation. It is designed by Raptor Computing Systems and is the world's first owner controllable workstation-class motherboard that is compatible with open source firmware from OpenPOWER. OpenPOWER is a foundation that strives to promote open sourced high performance processors, firmware, and software [7]. The Talos II supports dual P9 Central Processing Units (CPUs), trusted boot, and is compatible with OpenBMC. The OpenBMC compatibility allows the Talos II to boot from custom firmware which is essential to this research.

The Baseboard Management Controller (BMC) is a specialized service processor that resides on the motherboard and monitors all physical and network data. It is primarily used in server environments to control and monitor multiple systems to ensure normal operation. Even if the server is shut off, the BMC remains on as long

as the system is connected to a power source. The BMC provides the system administrator an avenue for communication with the server, specifically allowing remote power cycling and rebooting. This is possible because the BMC has its own IP address and can be accessed remotely from an external system. Having full control of the BMC firmware enables the capability to modify the instructions executed during the IPL [1] as well as the boot image that the system uses.

IPL refers to the operations that the Talos II executes from power on until the operating system starts up. During this initial start up, several interfaces are used to transfer data between the PNOR flash memory, random access memory, and the P9 module. The Intelligent Platform Management Interface (IPMI) and Fast Serial Interface (FSI) are two important communication interfaces that are used later during the IPL. Since this research focuses on the early portions of the boot flow, IPMI and FSI will not be discussed further in this thesis. LPC and Pervasive Interconnect Bus (PIB) are also two interfaces that are essential to the boot process. The LPC bus connects the P9 to external systems like the BMC. If trusted boot is enabled on the system, the LPC bus is used to communicate with the Trusted Platform Module (TPM) [5]. The LPC bus would connect the TPM and PNOR flash chip in Figure 1. The PIB exists inside of P9 CPUs and provides read and write access to the various components attached such as the On Chip Controller (OCC), Serial Electrically Erasable Programmable Read Only Memory (SEEPROM) and Self Boot Engine (SBE) as shown in Figure 1. Data transfer inside the processor will use the PIB and is considered secure because it is physically contained inside the P9.

The PIB connects several components, including the SBE and SEEPROM, that are essential to the start of the boot flow [1]. The SBE is an auxiliary microprocessor inside the P9 CPU that is specifically designed to initialize the first core to begin the IPL process. It executes from a Programmable PowerPC-lite Engine (PPE) and its

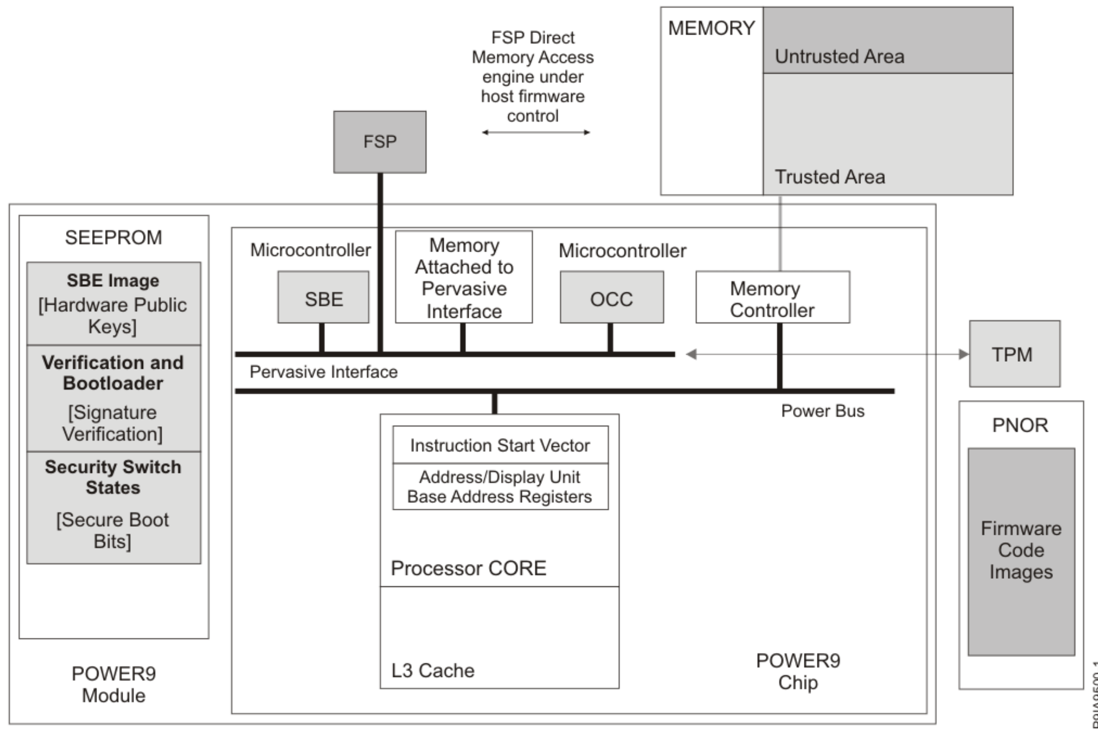

Figure 1. Detailed Hardware Layout of the POWER9 Secure Boot Environment [1]

firmware is stored in SEEPROM. As a backup in the event that the system fails to reach the 2nd stage bootloader, the SEEPROM stores two redundant copies of the initial boot stage. It also stores the root of trust hash when secure boot is enabled.

Two major concerns in this research were memory space limitations within the PNOR image and the Level 3 (L3) cache of the P9. The beginning portion of the IPL executes from the processor cache which is limited to 10 MB. This could cause problems with booting the system if the firmware requirements exceed the memory space available in the L3 cache. The second possible problem concerns the limited amount of space on the PNOR image. Each partition in the PNOR is allocated a specific amount of space so that the entire image fits on the flash chip. Although the PNOR has buffer space built in for each partition, problems could occur if the compiled firmware exceeds the allocated space. This concern was addressed by ensur-

ing that the correct space requirement checks are accomplished during the firmware compile stage. Additional code that causes the firmware to overflow the specified memory space would result in a failed PNOR compilation.

2.3 PNOR Image

The PNOR flash image contains all the instructions needed to boot the Talos II and is split into several sections defined in the PNOR Table of Contents (TOC) as shown in Figure 2. The TOC is one of the most important sections of the boot image because it contains data such as the name of the partition, physical offset, and physical size [8]. It is located at the beginning and end of the PNOR and is queried multiple times during the IPL by the P9.

The PNOR image allows developers to alter almost every part of the boot process with a single boot image file. The PNOR structure in Figure 2 is a general layout of the firmware used in OpenPOWER systems. The PNOR version specific to booting the Talos II is 4 MB and contains 31 sections ranging from 28 KB of memory configuration to 1.8 MB of boot kernel firmware data [8]. Each section in the PNOR contains a firmware module that is used during the IPL. The sections that are most relevant to this research are the Hostboot Extended Image (HBI) and Skiboot partitions.

The PNOR image needs to be compiled with the various firmware libraries used to build the boot image. The PNOR is best compiled through the buildroot-based op-build system, since the op-build system automatically compiles the PNOR image with firmware from the master git repository branch. This can be altered by using the `<pkg>_OVERRIDE_SRCDIR` variable which allows for custom firmware to be compiled into the PNOR image [9]. For this research, the Hostboot library was modified to contain custom firmware so the op-build command was used with `HOSTBOOT_OVERRIDE_SRCDIR`.

PNOR
Table of contents (TOC)
HBB
HB Extended
HBRT
HB data
SBE (update)
SBEc (centaur memory controller)
OCC
OPAL
Petitboot
Guard partition

Figure 2. High-Level View of the PNOR Image [1]

2.4 Initial Program Load

The Initial Program Load (IPL) is a term in OpenPOWER systems that refers to the operations that the system executes from power on until the operating system starts up. The IPL consists of five main sections as shown in Figure 3 with the two SBE stages, Hostboot, Skiboot, and Petitboot. This research focuses on the stages of the IPL up to Skiboot because the earlier sectors are easier to alter and contain important details regarding system start up. The firmware for each of the stages of the IPL are stored in the PNOR, allowing developers to alter the boot firmware by updating the flash image. Figure 4 shows a detailed breakdown of the IPL by ISTEPS and firmware sections. To start the IPL, the BMC sends the system start signal to the SEEPROM through the PIB. This starts up the 1st stage bootloader.

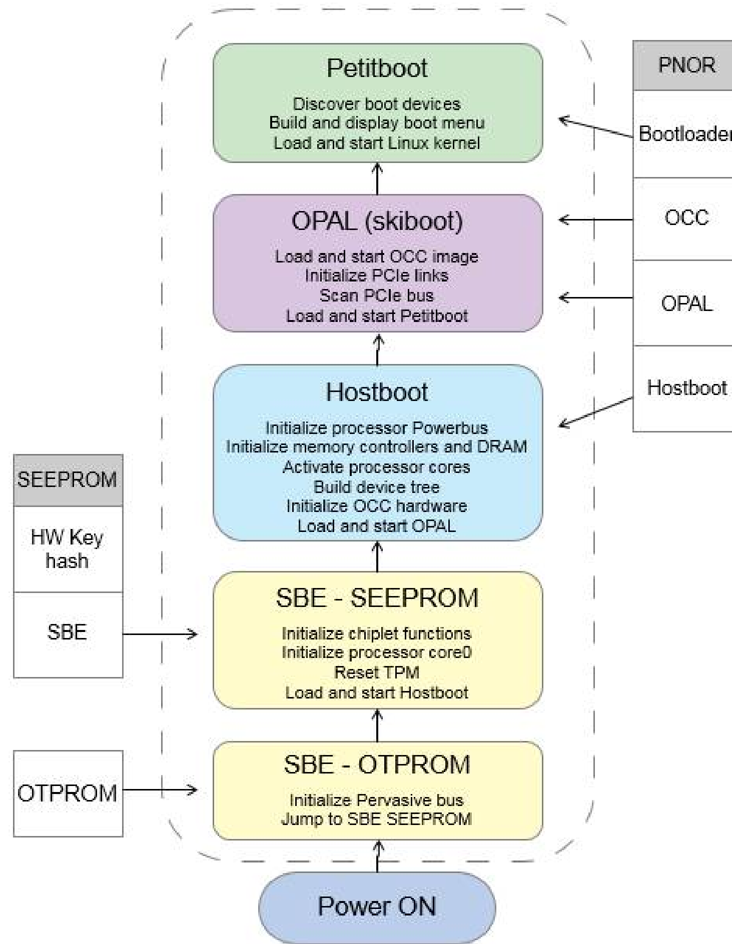

Figure 3. OpenPOWER Firmware Boot Flow [2]

2.4.1 1st Stage Bootloader

The 1st stage bootloader is stored in the SBE of the P9. Working with limited flash memory and a dedicated boot processor, first stage bootloaders face tight memory constraints. Any code added to the firmware must not exceed the flash memory space and can only contain instructions that the SBE is capable of executing. Computationally intensive instructions running out of the SBE can also impact performance of the boot processor and slow down the IPL flow.

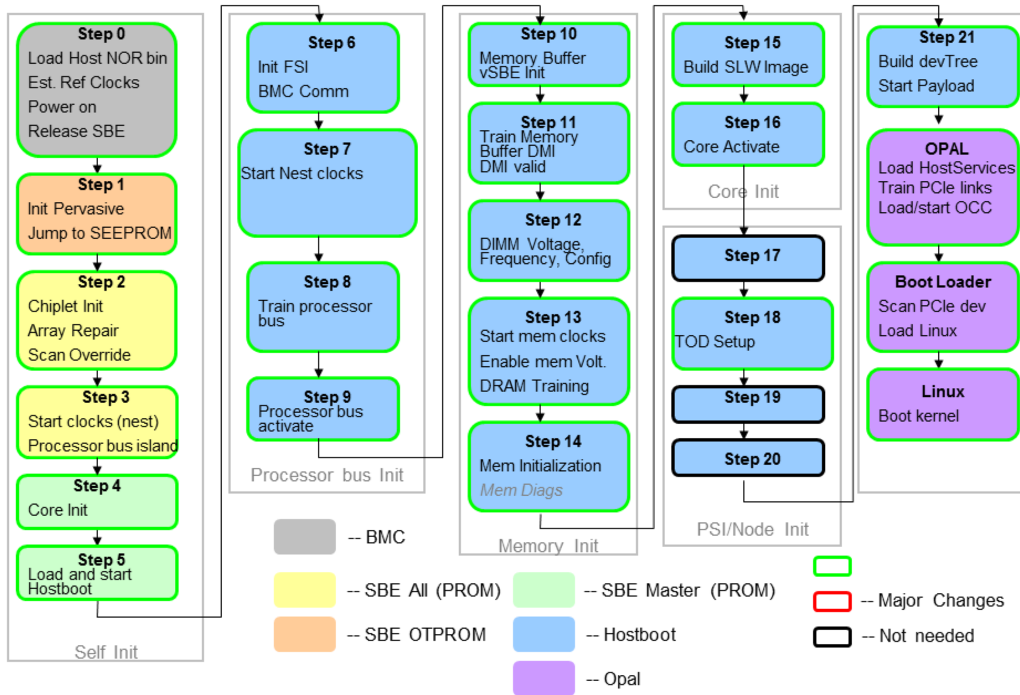

Figure 4. Detailed OpenPOWER Boot Flow [1]

The 1st stage bootloader on the Talos II system comes in two parts. The first section is permanently written on One Time Programmable Read Only Memory (OTPROM) through eFuses in the P9 silicon. The SBE firmware in OTPROM contains the first instructions that are executed on the SBE. The OTPROM section of the SBE initializes the PIB which provides a method of communication within the P9. The PIB is then used to load firmware from SEEPROM into the SBE core [1].

The SBE firmware from SEEPROM is responsible for initializing the first CPU core that the Hostboot Bootloader (HBBL) is run on and initializes the L2 and L3 caches along with access to the PNOR flash memory. If secure boot is enabled, the TPM would be reset during this stage. The SBE running off of the SEEPROM then loads the HBBL into the L3 cache and starts its execution.

2.4.2 Hostboot

Hostboot is an major portion of the IPL because it configures all the interfaces that are needed for Skiboot and the Operating System (OS) kernel. It acts as a cache-contained operating system for self-hosting chip initialization in POWER platforms. Virtual memory and a virtual file system layer are both used for demand-paging to bring code out of the flash chip as necessary [10]. This is because the code and data required for Hostboot does not fit on the 10 MB of L3 cache available. Hostboot is split into three main sections which are HBBL, Hostboot Base (HBB), HBI. They all function differently and have unique purposes to progressing the IPL.

The Hostboot Bootloader (HBBL) is stored in the SEEPROM and contains the first instructions that are run on the CPU core started by the SBE. It is responsible for cryptographically verifying the integrity of Hostboot when secure boot is enabled [10]. The main function of the HBBL is to load the HBB binary from the PNOR into the P9 L3 cache and start execution on the first CPU core. The important characteristic of this firmware is that it is stored within the P9 during the boot sequence, but can be updated through a successful PNOR update. This will be a key feature to securing the firmware decryption keys and will be covered later in Section 3.2.

The Hostboot Base (HBB) partition consists of a base initialization service task list which starts all the services needed in the HBI stage. HBB contains the Hostboot executable core, or kernel, and the services necessary to read and write to the PNOR. It provides message passing, task control, memory management, and interrupt support for the POWER processor. HBB also initializes the dynamic random-access memory, processor bus, and memory buffers, acting as the foundation for the HBI [7].

The Hostboot Extended Image (HBI) stage is executed through an extended initialization service task list in the form of 'ISTEPs' [10]. Each task in the service task list acts to progress the initialization of the chip. The HBI is structured in an

image containing Hostboot sub-component code with the Hostboot extended table of contents. This stage performs the majority of the system initialization to setup the system for the next boot stage. The Powerbus, memory controllers, POWER device tree, and Hostboot Runtime (HBRT) services are all initialized during this stage. The Powerbus is a set of buses that carry data between the processor and external resources like the cache, memory, and I/O. The POWER device tree is a data structure in memory that stores attributes and system configuration information. The HBI also wakes up the remaining processor cores by initializing the Sleep Winkle (SLW) image for each core. The last two ISTEPs in the HBI, ISTEP 20 and 21, are designed load and start Skiboot which is essential for the boot firmware encryption implementation shown in Figure 4. These ISTEPs are covered later in this thesis when discussing the implementation details. Once the final ISTEPs are complete, Hostboot releases control of the CPU to Skiboot.

2.4.3 Skiboot

Skiboot is late stage boot firmware that provides the OpenPOWER Abstraction Layer (OPAL) runtime services used later in the Skiroot. It provides wider platform initialization compared to hostboot and initializes Peripheral Component Interconnect Express (PCIE) controllers, device trees, real time clock, and several sensors [7]. The sensors are integral to the OCC, which is also started in the Skiboot boot stage. The OCC is an embedded subprocessor within the P9 that controls the thermal and power management of the processor. When working with the compiled PNOR image, it is important to note that Skiboot is compressed to fit 16 MB of instructions into 1 MB of space. The Skiboot image is decompressed at the end of the Hostboot stage in ISTEP 20.1. After Skiboot is complete, Skiroot and Petitboot are chain-loaded. Petitboot is the final boot stage and loads the runtime operating system.

2.5 Secure Boot

Secure boot is a feature that is used to verify the authenticity of each stage of the boot process and ensure that the device is only booting with Original Equipment Manufacturer (OEM) software. It usually operates by starting with one time programmable memory which creates the core root of trust. At every stage in the boot process, the bootloaders are signed and verified to ensure that the firmware is authentic with a signature and hash checking mechanism. The core root of trust then authenticates the first stage bootloader which is signed with the manufacturing key. After additional bootloader stages are signed and verified, the OS is signed which finishes the secure boot process. If all the firmware signatures are valid, the device will boot and the firmware gives control of the hardware to the operating system.

IBM's secure boot utility works by having each component in the IPL verify the integrity and authenticity of the following component before allowing it to execute. The integrity verification is completed through a secure hash like SHA512, and the authenticity is verified by a cryptographic signature like Elliptic Curve Digital Signature Algorithm (ECDSA). All the executable components are verified with secure boot with a chain of trust that is rooted in hardware. This means that the root of trust must be stored immutably in system hardware and is assumed to be inherently secure. Each component that follows is loaded from an unprotected location in flash memory so it is not trusted until verified. During the secure boot process, each component in the unprotected location in flash memory is moved to a secure container with the corresponding cryptographic signature. The OpenPOWER firmware must be stored in secure containers within flash memory to enable the secure boot verification. The boot process will stop if any of the components fails the authentication with an incorrect signature. This creates the complete chain of trust in secure boot for all executable boot code where the root of trust is anchored in platform hardware.

A two-level key hierarchy is used to ensure security of the hardware and firmware domain. The root keys are considered the hardware keys, while the signing keys are the firmware keys. The hardware keys sign the firmware keys, and the firmware keys sign each of the firmware components with three keys per set. This separation of duties within the signing process increases the overall security of the secure boot process. Each key pair is made from asymmetric keys which have a public and private side and uses 512-bit ECDSA. The private keys are stored in a hardware security module at the manufacturer while the public keys are compiled with the PNOR so that they are available for the signature check during boot. The hash of the public portion of the three hardware keys are stored in the SEEPROM to anchor the key hierarchy in hardware. During the IPL, each component of the PNOR is verified by checking the hash stored in the container with the value stored in SEEPROM. This is how secure boot verifies that the keys used to sign the firmware are properly authorized.

Enabling secure boot on the Talos II requires the use of a TPM which is a secure crypto-processor that is designed to execute cryptographic operations. The TPM takes a hash at each stage of the process and these hashes are signed to return an Attestation Identity Key (AIK). Attestation in the boot process is a signed report of the hash values retrieved during the various boot stages. The problem with the secure boot process is that the TPM requires the use of the LPC bus to communicate with the P9. An interposer could be used to sniff and modify the LPC signals between the host and TPM. Although secure boot is not used in this research, the structure of firmware verification is used for the secure key storage implementation.

2.6 Secure Key Storage

When considering the problem of using an encrypted PNOR image, the problem of secure key storage also needs to be investigated. The principal challenge to using encrypted boot firmware is that a boot stage needs to exist within the hardware capable of decrypting the subsequent firmware section. If the decryption boot stage is stored on the PNOR flash chip, the plaintext code would need to travel across the LPC bus from the PNOR to the P9, which is vulnerable to interception.

One way to circumvent this problem is through Public Key Infrastructure (PKI). The keys would be stored in separate secure locations and shared over an insecure channel using a combination of symmetric and asymmetric encryption. PKI would allow for data to be transmitted across the LPC bus but also requires control of both endpoints of the PKI implementation. This is difficult with the Talos II meaning that the task remains in the effort to accomplish secure key storage.

2.7 Encryption Schemes

The boot firmware encryption method was initially developed with an XOR cipher due its simplicity as a proof of concept and possible use as a one time pad. After achieving a working proof of concept, SPECK and Advanced Encryption Standard (AES) were chosen as two well known encryption standards to compare the encryption performance during the IPL. SPECK represents the performance of a lightweight cipher [11] while AES represents the performance of a computationally intensive cipher.

2.7.1 XOR

An XOR cipher can be considered a secure encryption algorithm through the use of a one time pad, which cannot be cracked (if used only once) but requires a key that

is as long as the plaintext. In this implementation, the key is truncated to 16 bits, otherwise the decryption key would take up an additional 1 MB of valuable memory. A simple XOR operation was used in this case to explore the capabilities of the P9 during IPL.

2.7.2 SPECK

The second encryption standard selected was SPECK, a lightweight block cipher optimized for software implementation. It was developed by the National Security Agency (NSA) alongside SIMON which is also a lightweight block cipher, but optimized for a hardware implementation. Both SPECK and SIMON are ciphers with a range of options of block and key sizes. They were created for use in Internet of Things (IOT) devices with the US Government to minimize computational performance impact. SPECK was selected because it is one of the best performing lightweight ciphers and does not use lookup tables [11]. This reduces the size of the compiled code as the tables do not have to be stored in the hostboot source code. The lightweight nature of SPECK also allows it to run efficiently, reducing the performance impact the decryption phase has on the boot sequence.

2.7.3 Advanced Encryption Standard

AES was selected as a heavyweight encryption standard because it is a well-known and widely used block cipher. AES utilizes a Rijndael algorithm that was published in 1998 and is maintained by the Department of Commerce, National Institute of Standards and Technology, and Information Technology Laboratory [12]. It is a symmetric key algorithm, using the same key for encrypting and decrypting data. Although AES supports a range of block sizes, this research uses AES-256 as a sufficiently secure configuration. As stated by the NSA, the design and strength of all key lengths of the

AES algorithm are sufficient to protect classified information up to the Secret level while Top Secret information will require use of either the 192 or 256 key lengths [13]. The block size was a concern because the limited space in memory poses a possible problem during run time. AES uses a substitution permutation network which means that it is efficient in hardware and software implementations.

2.8 Related Work

Before developing the boot firmware encryption for this research, background research was conducted to understand previous work in this field. Three examples of successful boot firmware encryption were used as a foundation in this thesis. Although these implementations were designed for secure firmware updates on microcontrollers, the structure of decrypting an encrypted firmware update is similar to boot firmware encryption.

2.8.1 Secure Firmware Updates with AES Encryption

The process of encrypting sections of the bootloader before being loaded onto the system is important to this research. A guide by Silicon Labs helped introduce the idea of encrypting boot firmware and decrypting it in a secure storage space. The goal of this paper was to provide a secure distribution system for firmware updates to microcontrollers. This is needed because application sensitive information could be stored in firmware updates which could be accessed by bad actors to develop zero day vulnerabilities. This implementation from Silicon labs allows the microcontroller system to receive encrypted firmware and decrypt it once it is stored in the internal flash chip [14]. This structure is applicable to boot firmware encryption in the Talos II by applying the encryption and decryption portions to the IPL flow.

Microchip Technology also implemented an encrypted bootloader that functions

similar to the AES encrypted bootloader from Silicon Labs. The purpose of the encrypted bootloader was to safely update ultra-low power microprocessor firmware in the field through encryption [6]. A vulnerable firmware update process for in-field embedded systems would allow attackers to compromise system security and possibly gain complete control of the deployed system.

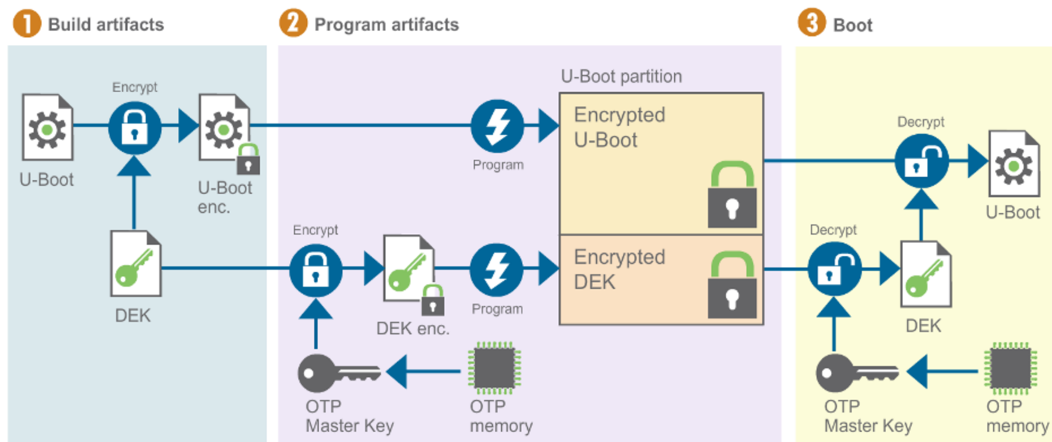

Figure 5. U-Boot Firmware Encryption Setup [3]

2.8.2 U-Boot Image Encryption

Digi developed a method of encrypting signed firmware to obscure image data from unauthorized users. It was applied to U-Boot which is an open source bootloader used in embedded devices and compiles the instructions necessary to boot the embedded system operating system kernel [3] just like the PNOR image in OpenPOWER systems. The process of encrypting firmware and decrypting it during the boot sequence is identical to the process used in this research. Figure 5 shows the layout of running an encrypted U-boot image with secure key storage. The implementation of the encrypted U-boot firmware can be applied to specific stages in the PNOR image. A concern with this implementation is the requirement for an One-

Time Pad (OTP) memory chip within the processor. Since this is not feasible within the scope of this research, only the encryption and decryption format is useful.

2.9 Summary

This chapter covers the background information surrounding boot image encryption like the Talos II architecture, firmware used during boot up, and the stages the firmware is utilized in. Secure boot is also reviewed with the concepts related to secure key storage and encryption algorithms. Finally, the research related to boot firmware encryption is discussed to provide context to the work in this thesis.

III. Implementation Discussion

3.1 Implementation

A key challenge with this research was configuring the decryption firmware to work within the constraints of the hardware during the IPL. In order to demonstrate the feasibility of boot firmware encryption, a simple proof of concept was used before starting the experiment. The methodology for this proof of concept was complex due to the multiple stages required to make firmware changes. Before the PNOR was flashed onto the Talos II, a section of memory needed to be identified for encryption. The decryption function was then programmed into the firmware that loads the section of memory. For example, since the Hostboot stage chain-loads the payload stage, the payload would be encrypted and the decryption function would be located in Hostboot. The PNOR was then compiled and the address space identified earlier was encrypted. The PNOR with the encrypted section and decryption function was flashed onto the Talos II. The implementation proved to be successful through multiple successful boot sequences.

After analyzing the IPL firmware, several points in the code were identified where the decryption code could execute successfully. The first possibility was the HBBL which was located in `bootloader.C`. A function called `handleMMIO` is used to copy the HBB code from the PNOR to a working location. The HBBL code is executed near the start of the boot process, and machine code is used to instruct the LPC to transfer data. At this stage of the boot, the console output (print out and log commands) is not yet accessible, making it difficult to verify the correct address and size of the partition for encryption. Although this was a limitation during the proof of concept development, the `bootloader.C` firmware will be employed for secure key storage.

A non-trivial concern throughout development of the decryption mechanism was the risk of "bricking", or rendering inoperable, the P9 through a broken boot sequence. This could arise while altering the firmware before HBB in the IPL because the SEEPROM uses two sides to save low-level firmware while keeping a backup. When the Talos II boots up with new firmware, the new HBBL is saved to Side 0. If the system successfully runs through the HBBL once, the system saves the new firmware to Side 1. A critical problem may occur if the firmware successfully boots once but fails in subsequent runs, resulting in permanent failure to boot, i.e., the P9 could get "bricked". This risk could be mitigated by altering the firmware that conducts the SEEPROM side updates. By default, the Talos II uses sequential SBE updates which updates the second SEEPROM side upon a successful SEEPROM Side 1 update and boot. The system allows for independent SBE updates which prevent the second side of the SEEPROM from being updated, so the BMC can be manually configured to boot off the Side 1 memory to use the stock HBBL firmware.

A second possible location where decryption code could execute is the HBI. Since HBB handles the modules for starting the HBI, it was possible to narrow down the location to the `vfsrp.C`, which is the Virtual File System Resource Provider (VFSRP). HBI is executed through ISTEPs, and a module list which each have their own services to initialize. HBB reads through the module list and sends messages to the P9 to load each task and initialize it. Each task sends a `MSG_MM_RP_READ`, which executes a `memcpy` to load the relevant data into the local execution space. A concern with using the VFSRP is that a maximum of 68 tasks may be started during the HBI phase. This means that 68 addresses and address spaces must be managed during the encryption phase making this implementation extremely complicated. Each of the 68 modules will need to be individually encrypted and the decryption firmware will need to decrypt each module as it is loaded into the local execution space.

A final possible location where decryption code could execute is the end of the HBI in `call_host_load_payload.C` in ISTEP 20. This code is responsible for calling a function, `load_pnor_section`, which runs one time to load the payload from the PNOR into local memory space. One complication working in this code is the XZ compression. Since the payload section is nearly 16 MB but is only allocated 1 MB on the PNOR image, it must be compressed when stored on the flash chip. The decompression code handles the `memcpy` from PNOR but raised concerns with read and write operations with regard to the LPC address space.

```
// Display a banner on the status of PNOR compression
CONSOLE::displayf(NULL, "PNOR is XZ compressed");
CONSOLE::flush();

struct xz_buf b;
struct xz_dec *s;
enum xz_ret ret;

xz_crc32_init();
s = xz_dec_init(XZ_SINGLE, 0);
if(s == NULL)
{
    TRACFCOMP(ISTEPS_TRACE::g_trac_isteps_trace,ERR_MRK
            "load_pnor_section: XZ Embedded Initialization failed");
    return err;
}

CONSOLE::displayf(NULL, "Starting malloc");
CONSOLE::flush();

uint8_t* temp_buf = (uint8_t*) malloc(originalPayloadSize);
```

Figure 6. Hostboot Malloc Code

One approach is to allocate a space to store the encrypted payload, as shown in Figure 6, and copy the data from the PNOR to the local memory space. The encrypted payload was loaded in 4 KB chunks, as shown in Figure 7. After the encrypted data was stored in the `temp_buf`, the data was encrypted in place in Figure 7. The

encrypted data was then passed into the XZ decompression function. The `displayf` function from the console was used to display important data to the BMC console client and helped determine if the code failed at a specific section.

```
const uint32_t BLOCK_SIZE = 4096;
for ( uint32_t i = 0; i < originalPayloadSize; i += BLOCK_SIZE )
{
    memcpy( reinterpret_cast<void*>(
                reinterpret_cast<uint64_t>(temp_buf) + i ),
            reinterpret_cast<void*>( pnorSectionInfo.vaddr + i ),
            std::min( originalPayloadSize - i, BLOCK_SIZE ) );
}

CONSOLE::displayf(NULL, "Done copying to local buffer");
CONSOLE::displayf(NULL, "Starting XOR");
CONSOLE::flush();

for ( uint64_t i = 0; i < originalPayloadSize; i++ )
{
    temp_buf[i] ^= 0x55;
}
```

Figure 7. Hostboot Decrypt Code

That approach results in a problem with `call_host_load_payload.C` because the payload in the Talos II PNOR is XZ compressed. The `memcpy` responsible for loading the compressed payload exists in the XZ decompression code, adding a layer of complication. In this implementation, space is allocated in memory to manually store the PNOR image, decrypt the compressed image, and pass the decrypted data into the decompression function as shown in Appendix A. The corresponding XOR firmware encryption code, shown in Appendix B, is programmed in Python for simplicity.

In this approach, the XOR decryption function is used directly on the data pointed to by the `pnorSectionInfo.vaddr`. The virtual address specified for each PNOR section is memory mapped to an address space on the LPC bus. This is a problem with attempting to XOR the data in place because this would attempt to write over

the PNOR section over the LPC bus. An alternate implementation, with a higher cost to memory, allocates a 1 MB block that would copy memory from PNOR and XOR the data in place. This memory space is then passed into the decompression function resulting in a successful boot.

After successfully running all the steps, a standard boot operation was conducted as a test. This was done to verify that the XOR function did not interfere with the OPAL firmware and the following boot process. Figure 8 shows the OpenBMC console client logs where the print statements in Figures 6 and 7 are shown. The figure also shows ISTEP 21.3 and the following log shows that Skiboot was able to start, implying that the XOR code ran successfully. The XOR of the header can be verified with the data printed out in the console client. The header for Skiboot begins with the hex values of `fd 37` which matches the XORed header values. After running the boot sequence several times and cycling the power, the Talos II was able to boot Linux without any problem.

With a working XOR proof of concept completed, SPECK was chosen as the lightweight block cipher to implement next. The decryption function, shown in Appendix C, was programmed in `C++`. The Speck and Simon implementation guide provided working sample code with test vectors. The encryption function in Appendix D was programmed in Python due to the simpler address management and file management code. After confirming that the SPECK cipher successfully works with the boot firmware encryption model, the final algorithm to implement was AES. The decryption function was programmed in `C++` because the cipher must work with the rest of the firmware which is in `C++`. As shown in Appendix E, the substitution boxes along with the supporting functions must be stored in the ISTEP 20 firmware along with the decryption function. Appendix F shows the AES encryption function written in Python for simplicity.

```
69  33.84818|ISTEP 20. 1 - host_load_payload
70  33.87024|call_host_load_payload.C: Starting load_pnor_section()
71  33.87144|call_host_load_payload.C: PNOR is XZ compressed
72  33.87391|call_host_load_payload.C: Starting malloc
73  34.53205|call_host_load_payload.C: Done copying to local buffer
74  34.53735|Header 0 = fd
75  34.53027|Header 1 = 37
76  34.53836|Header 2 = 7a
77  34.53878|Header 3 = 58
78  34.53926|Header 4 = 5a
79  34.53970|Header 5 = 00
80  34.54022|Header 6 = 00
81  34.54065|Header 7 = 01
82  34.54111|Header 8 = 69
83  34.54157|Header 9 = 22
84  34.54202|Header 10 = de
85  34.54250|Header 11 = 36
86  34.54295|Header 12 = 02
87  34.54348|Header 13 = 00
88  34.54393|Header 14 = 21
89  34.54442|Header 15 = 01
90  34.54484|call_host_load_payload.C: Done XORing
91  34.54563|Running Decompression
92  34.66993|Freeing buffer
93  34.67050|call_host_load_payload.C: Ending load_pnor_section()
94  34.67252|ISTEP 20. 2 - host_load_hdat
95  35.55996|ISTEP 21. 1 - host_runtime_setup
96  41.29561|htmgt|OCCs are now running in ACTIVE state
97  46.02538|ISTEP 21. 2 - host_verify_hdat
98  46.05073|ISTEP 21. 3 - host_start_payload
99 [   46.323404859,5] OPAL skiboot-9858186 starting...
```

Figure 8. Proof of Concept Decryption

3.2 Secure Key Management Process

The data contained in the P9 module is considered secure under an assumption from Section 3.3; that adversaries do not have the capability to sniff the interfaces within the processor. This implies that data and keys are secure if they are stored inside the P9 module, provided there are no methods to retrieve them from external interfaces. With this understanding, there are three methods that can help solve the problem of secure key management that will be discussed.

The first method involves adding an OTP storage chip in the POWER9 module which is how manufacturers typically manage keys. The OTP storage chip would be connected to the PIB and transmit the decryption key to the P9 L3 cache when the

firmware decryption code executes. Although this implementation would allow for secure key management for boot firmware encryption, it also requires hardware to be physically added to the P9. Since this research focuses on firmware-only solutions, this method is not feasible.

An alternative to adding a storage chip to the P9 is to utilize existing SEEPROM memory space on the SBE. The SEEPROM memory stores the SBE firmware along with verification keys for secure boot. During secure boot, the verification keys are compared to the hash of the boot image. The key management system can be used to securely transport keys to the image decryption firmware when needed. A significant constraint with this option is that the SEEPROM memory space is extremely limited in size. In fact, the verification keys must be stored as hashes because they occupy too much space as raw keys. This limitation with the SEEPROM memory prevents this option.

Another method for secure key management can be implemented in firmware by utilizing the structure of the initial program load with boot firmware encryption. The Hostboot bootloader stage is integral to this method because it is stored and runs in the SBE. This means that it is isolated to the P9 module and is secured from adversaries. The following stage is the HBB image which travels from the PNOR, across the LPC bus, and into the L3 cache of the P9. The HBB section can be secured by encrypting the base section and implementing a decryption function in the bootloader.

This option also enables the user to update the decryption firmware through the PNOR because the boot image contains a HBBL section, meaning that the bootloader firmware can be altered and recompiled into a new PNOR image. Upon the first successful Talos II boot with an altered HBBL image, the SBE will detect a firmware difference and proceed to update the SEEPROM Side 0 memory at the start of the

IPL. The system will then reboot from Side 0. Then, if the system successfully reaches ISTEP 10, the SEEPROM Side 1 will also be updated. This is important because it allows the user to safely update the SBE and HBBL firmware.

After the final bootloader firmware is updated on both sides of the SEEPROM, several steps need to be executed to ensure that the bootloader stays protected. The first step is to ensure the SEEPROM side updates are disabled. This is required because the SBE automatically updates the HBBL firmware if it detects a firmware change. The first step will allow us to encrypt the HBBL or remove the HBBL section in the PNOR but still use the unencrypted HBBL stored in the SBE, protecting the key stored in the firmware. Preventing automatic SEEPROM updates will lock the system so that the SBE firmware can't be updated.

To implement this method, a PNOR containing the decryption code in HBBL with an unencrypted HBB needs to be flashed onto the system. This is required because on the initial boot, the SBE will not have the decryption code in the HBBL to decrypt HBB. After reaching ISTEP 10.5, the system will flash the new HBBL to the SBE. This will give the system the capability to boot with an encrypted HBB section in subsequent system startups.

To remove the secure key storage and management, the reverse of the aforementioned method needs to be executed. A PNOR image with HBBL firmware without the decryption code and encrypted HBB firmware needs to be compiled and flashed. After the SEEPROM side update, the SBE will have the HBBL firmware without the decryption code, meaning it can now boot an unencrypted HBB. The 2nd SEEPROM side will be updated on ISTEP 10.5 again allowing the system to boot with an unencrypted HBB. A PNOR without decryption firmware and an unencrypted HBB will then be flashed onto the system, allowing for normal PNOR updates.

3.3 Application Scenarios

Boot firmware encryption can be applied to any embedded application with proprietary information that needs to be protected. For example, self driving cars may be exposed to corporate espionage and reverse engineering attempts. Since it is not feasible to limit the sale of self driving cars, anyone can get access to the hardware and attempt to leak information from the firmware. Manufacturers may want to deter attackers from revealing sensitive firmware from the hardware. Leaking the firmware may allow competitors to get access to proprietary self driving algorithms and possibly replicate it. Encrypting the boot firmware of the embedded systems would prevent this and make it extremely difficult for any 3rd party to uncover sensitive data in firmware.

Another application scenario for boot firmware encryption is in public infrastructure. This research can be applied to unprotected end node environments in systems like the power and water grid [15]. In the power grid, the Phasor Measurement Unit (PMU) records the phasor data at various points in the grid and the data gets aggregated at phasor data collectors. These data collectors may be vulnerable to hardware attacks, allowing attackers to gain access to the infrastructure system [16]. Manipulating the sensor systems could cause a power grid failure and damage power plant systems. The water grid also has similar vulnerabilities with pressure and flow sensors being placed in vulnerable end node environments. The data collectors for these sensors could be protected with boot firmware encryption, preventing possible hardware attacks.

IV. Experimental Setup and Methodology

4.1 Objective

The goal of this experiment is to measure the performance impact of an implementation of boot image encryption. The experiment is designed to provide sufficient data for a discussion on the options of boot firmware encryption and the possible performance trade-off for security. The experimental setup is specified along with a breakdown of the development process along with the secure key management implementation. The computer setup for developing the boot firmware encryption and collecting data is also specified for future research and development.

4.2 Assumptions

The assumptions made in this experiment are related to the experiment design decisions, constraints and known factors.

- Level 3 Cache Security: Adversaries do not have the capability to directly read the L3 cache to observe data stored inside during operation. Since the encrypted firmware is decrypted in the processor, the plaintext firmware would be observable in the cache. It would be extremely difficult to pause the boot process at the right time and remove volatile memory from the P9 module, but it may be possible for a skilled hardware attacker.

- SEEPROM Security: Adversaries do not have the capability to remove the SEEPROM from the P9 and extract data from memory. The secure key management implementation relies on the assumption that the memory interfaces within the P9 module are relatively secure. The ability to read data out of

SEEPROM would allow attackers to retrieve the firmware decryption keys and invalidate the security of the PNOR firmware encryption.

- BMC Firmware Access: The BMC cannot access data within the L3 cache or SEEPROM during the IPL. The ability to forcibly read data from the L3 cache or SEEPROM would allow outsiders to view the decrypted firmware, invalidating this research effort. The SEEPROM can be secured through the use of Secure Boot because the SBE memory is locked from the BMC when the secure mode jumper is enabled [17].
- Effect of PNOR Updates on Boot Timing: Each trial does not have an effect on the subsequent boot up. Artifacts left from previous PNOR images may cause marginal effects on the system performance during IPL. Randomizing the selection of PNOR images should mitigate any residual effects caused by previous boot images.

4.3 Control Variables

The control variables are those held constant over the execution of the various scenarios and experiments of this research. In this experiment, the Talos II system was kept constant as the only system the data collection was run on. The firmware was also constant throughout the duration of the data collection to minimize possible sources of error. The firmware loaded onto the microcontroller allows it to act as a BMC and perform the repeated trials for data collection in this experiment. Although the microcontroller has custom functionality added to the firmware, it still performs similar to a stock Talos II system.

4.4 Independent Variables

The independent variables are aimed at evaluating and optimizing the boot image encryption performance. For this experiment there are several factors to consider when capturing boot image encryption performance. This effort focused on two factors which directly answer the research questions pertaining to this effort. The main research objective was to understand the performance impacts of various types of encryption for boot firmware encryption. The first factor was the type of encryption, with 4 representative levels, including: no encryption, XOR cipher, SPECK, and AES. The second factor is the amount of the section that is encrypted which will be referred to as the encryption ratio. Although partially encrypting the Skiboot firmware makes little security sense, the experiment used firmware versions capable of encrypting 25%, 50% and 100% of the Skiboot firmware in order to measure the per byte encrypted overhead.

A full factorial design was utilized when narrowing down the experimental factors. Since this experiment has two factors with each factor having three to four levels, an analysis would require at least 16 trials. In this experiment, we decided to run 100 trials per factor level resulting in 1000 total runs. This helps us determine the main effects and interactions on the boot timing metrics. A full factor analysis will reveal the optimal encryption type and encryption ratio for boot performance and security.

4.5 Response Variables

The response variables for this research are correlated to the performance of the boot image encryption. The data collected for this experiment is primarily time based to measure the effects the decryption code has on boot time. Several locations throughout the IPL were selected to measure the total time to run Hostboot, time to load the payload, time to decrypt Skiboot, and time to decompress Skiboot. These

metrics were collected through timestamps in the console client logs.

The second response variable was memory consumption on the flash chip for the decryption firmware. The boot firmware encryption requires the decryption code to be stored on the PNOR which has limited space. The amount of memory left available in the HBI was measured and collected at the end of each PNOR image compilation. This will allow for an evaluation of the decryption firmware memory requirements.

4.6 Performance Evaluation

The performance of the boot firmware encryption is evaluated by measuring the boot time required by different encryption algorithms and encryption ratios. Characterizing the various boot firmware encryption factors can help paint a clear picture of its impact on the Talos II. Different portions of the boot process are measured to decompose the performance impacts of the decryption firmware. For example, during ISTEP 20, the payload section is copied into the local buffer, decrypted, and decompressed. Each of these steps are measured to assess how the different encryption algorithms perform.

4.7 Experimental Setup

Consistent test runs are required to gather reliable boot timing data because there may be artifacts from previous boot images that could affect the boot time of future trials. To solve this problem, the PNOR images were randomized to minimize the influence of noise or other anomalies. During a new PNOR image boot sequence, an error correction check is executed and updates the boot image. This is done during the first few boots of a new image update so each PNOR image has to boot multiple times until it stabilizes. The finalized images were downloaded after they were able to boot without requiring a reboot during the IPL. This allows different PNOR images to

be flashed for each trial without needing to reboot the system during data collection.

Although this experiment can be run on a stock Talos II system as shown in Figure 9, running hundreds of boot ups for data collection on the stock BMC can be a problem. Since the ASPEED BMC cannot be modified at the hardware level in this research, it is extremely difficult to add functionality that is required for running hundreds of data collection trials. Instead, a Talos II setup with a microcontroller acting as the BMC was used which allows for more control over the BMC functionality.

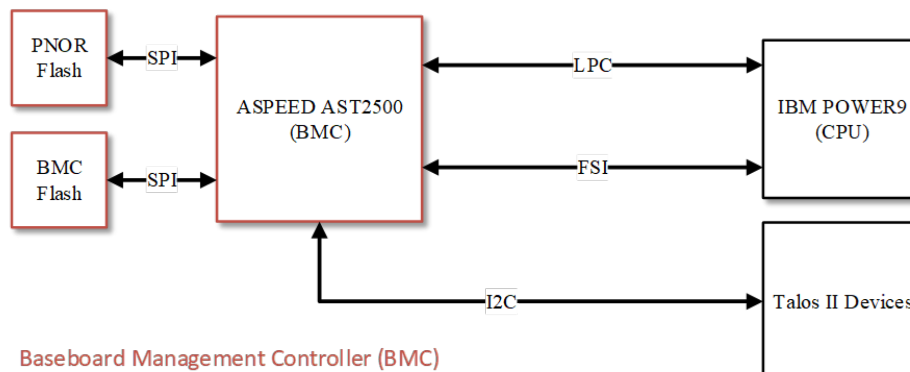

Figure 9. Stock Talos II Setup

4.7.1 Micro Controller Setup

To run the data collection for boot encryption efficiently, a microcontroller setup was combined with the Talos II as shown in Figure 10. Instead of booting from the Talos-supplied ASPEED 2500 BMC, an ARM Cortex microcontroller was employed to accomplish the task. A field-programmable gate array (FPGA) was also used as the LPC bridge between the microcontroller and P9. This setup enabled a change to how the PNOR is transferred onto the Talos II.

Three python scripts were used to run the experiment with the micro controller setup. The first is the `talos_ctrl` code in Appendix G which handles the BMC com-

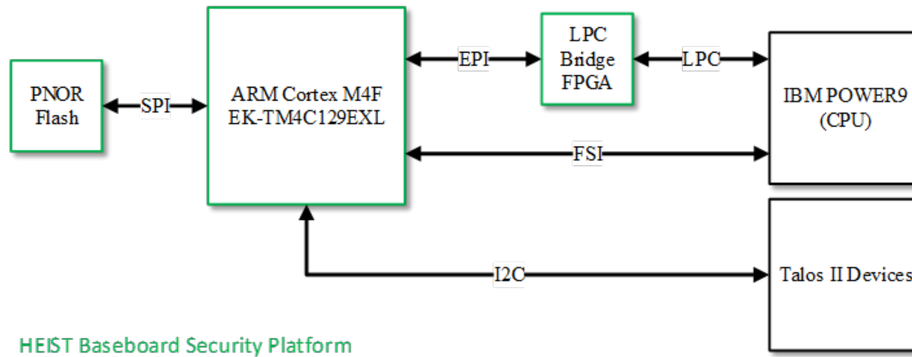

Figure 10. Microcontroller Setup

mands in the microcontroller. This was important for scripting the data collection trials because it allowed for the automation of PNOR flashing and the Talos II reboots required after each data collection run. The `udp_logger` shown in Appendix H was essential in two ways for this research. The firmware contains the functionality to collect detailed IPL data and save the files to the system controlling the microcontroller. The firmware also allowed the Talos II to boot off of PNOR images supplied over Ethernet with a User Datagram Protocol (UDP) server which was essential for automating the data collection. The `experiment_script` in Appendix I combined the functions in the `talos_ctrl` and `udp_logger`. It processed the data file of randomized PNOR images and collected the boot time metrics for each trial.

The data collection system was set up so the microcontroller acting as the BMC would wait for the PNOR image to be transmitted over Ethernet during boot. A laptop with all the PNOR images would select one image to send through UDP. The capability to boot without flashing the PNOR chip saved around 5 minutes per boot. Although the PNOR images were supplied to the Talos II through Ethernet, this does not affect the performance of boot time decryption which is entirely contained within and dependent on the P9.

The microcontroller setup also has a UDP logger that is set to collect data from the boot process. All the boot and error logs are transferred to a laptop for straightforward data collection on repeated runs. The logger records the LPC data transfers, PNOR access messages, warning logs, and P9 messages. The data of interest is contained in the P9 message log file.

4.8 Summary

This chapter described the objectives, assumptions, and variables of the experiment covered in Chapter V. The process of developing boot firmware encryption and secure key management is also explained to provide a better understanding of this research. The application scenarios detailed the different ways boot firmware encryption could be employed and how its implementation would solve the possible vulnerabilities in the scenarios. The performance evaluation, secure key management, and experimental setup are further detailed in Chapter V.

V. Observations and Analysis

5.1 Overview

This chapter presents the observations, results, and analysis from the experiment described in Chapter IV. An evaluation of the secure key management implementation is presented along with its constraints and limitations. The memory requirements for each encryption algorithm are discussed to show the possible hardware limitations with this research. The performance experiment consists of an evaluation of the boot firmware encryption, comparison of encryption methods, and the encryption ratio performance.

5.2 Secure Key Management Analysis

The method of securing the firmware decryption keys was described in Chapter IV and implemented to assess its feasibility. The main concern when altering firmware that is stored in the SBE is the amount of memory space available in SEEPROM. The HBB decryption was first tested with XOR by loading the XOR decryption function from Appendix J into the `bootloader.C` firmware. The compiled PNOR image was then flashed onto the system without encrypting the HBB section. This is required so that the SEEPROM updates will be complete after ISTEP 10 which requires the HBB image to be unencrypted. After the system successfully updates the SEEPROM side 2, a PNOR with an encrypted HBB from Appendix K can be flashed onto the Talos II. The HBBL firmware is located at `0x3971000` and the specific `bootloader.C` firmware is located at `0x3972200`. The stock HBBL firmware requires 10197 Bytes. With the XOR code loaded into the HBBL firmware, the memory requirement of `bootloader.C` increased by 107 Bytes to 10304 Bytes. The system was also able to boot with SPECK encryption in the firmware shown in Appendix L on the HBB

section. Incorporating this algorithm only required 917 more Bytes than the stock HBBL firmware to implement successfully and required 11114 Bytes.

An AES HBBL implementation was attempted but failed due to the memory requirement to store the decryption firmware. Appendix M shows the compilation failure from op-build stating that the HBBL raw size without padding and Error Correction Code (ECC) was 21040 Bytes when the limit was 20480 Bytes. This resulted in HBBL not being able to fit the hardware key hash at the end of the section without overwriting real data. Since there was not enough room within the HBBL section, the PNOR image did not compile.

The security of the firmware decryption keys relies on the security of the SBE and SEEPROM. If we assume that the interfaces within the P9 are secure, this method of securing the firmware decryption keys should also be secure. Since the keys in the later IPL stages will be stored in the source code which is encrypted, it is safe to assume that the keys for the later firmware stages are secure. Future work will consider the assumption that the interfaces within the P9 are secure.

5.3 PNOR Image Size

Understanding the memory space required for the firmware decryption keys and the firmware decryption function is essential when applying this research to different systems. Since the PNOR image only has a limited amount of space allocated for each firmware section, it is important to avoid overflowing the space available. Another consideration is the amount of space available in the execution space memory. For example, the SEEPROM that stores the SBE firmware is only 64 KB, so memory space is a concern when adding code to the firmware. Compiling the PNOR image through the op-build system makes the process of altering firmware safer because it verifies the firmware will not overflow the allocated space for each section.

Although the op-build compile system prints out the memory utilization for each section, the metrics are not a true representation of the real memory space used. This is because some of the sections are substantially padded before applying the error correction code so the compilation output is not accurate for memory analysis. To get around this problem, HxD which is a hex editor, was used to manually calculate the amount of memory consumed for each encryption algorithm.

Table 1. Decryption Firmware Size by Encryption Type

Encryption Type	Start Address	End Address	Image Size (Bytes)	Total Size (Bytes)
Normal	0x7EA400	0x7EB6F0	4848	9216
XOR	0x7EA400	0x7EB820	5152	9216
SPECK	0x7EA400	0x7EB830	6144	9216
AES	0x7EA400	0x7EC410	8208	13824

Each PNOR image was loaded into the hex editor and the start of ISTEP 20 was located at `0x7EA400` as shown in 1. This was done by searching for the text used in the print statements for this ISTEP. This is possible because the PNOR image stores the plaintext of the print statements at the end of the chunk of ISTEP 20 hex instructions. For the stock Hostboot PNOR image, the ISTEP 20 section ended at `0x7EC800` meaning that the normal Hostboot ISTEP 20 section takes 4848 Bytes. There is padding at the end of each section, giving ISTEP 20 9216 Bytes of space in the PNOR image.

The XOR decryption firmware ends at the hex address `0x7EB820` meaning that it requires 5152 Bytes which is 304 Bytes more than the stock PNOR image. The SPECK firmware takes 6144 Bytes which is 1296 more Bytes than the stock firmware. Both the XOR and SPECK algorithm and keys do not exceed the default padding allocated so they only require 9216 Bytes of space in total on the PNOR image.

With the ECC added to the AES decryption firmware, the ISTEP 20 firmware section overflows the 9216 Bytes allocated, requiring more memory space. After compiling the PNOR image with AES, the final ISTEP section takes 8208 Bytes. With the ECC and padding, the section takes a total of 13824 Bytes on the PNOR image. This means that the code requires 3360 Bytes more than the stock image and the padding causes the section to require 4608 Bytes more than the stock firmware. Since the HBI firmware has extra padding at the end of the allocated space, the additional firmware does not cause problems in the final image compilation process. Since there is extra room in the PNOR image with the padding, the additional memory requirement is not a limitation. Using AES on a system would only become a limitation if the flash chip could not support the additional 4608 Bytes required.

5.4 Performance Experiment Results

After running 1000 boot data collection trials, a python script was used to process the experimental data in Appendix J. The timestamps corresponding to the boot stages that are important to this research were extracted through regular expressions and inserted into a data frame. The data was exported into a .CSV file and processed in RStudio. 2 shows the total data collected for each encryption type and encryption ratio. Since the data shows the boot time metrics, a lower time is better. The Load Payload data represents the amount of time the system requires to execute the `host_load_payload.C` firmware section.

5.4.1 Performance Impact of Boot Image Encryption

The first research objective was to observe any performance differences with implementing the boot image encryption. Figure 11 displays the time for the Hostboot phase of the IPL to complete. This is measured from ISTEP 3 until the start of

Table 2. **Breakdown of Boot Times by Encryption**

Encryption Type	Encryption Ratio	Load Payload	Decryption	Decompress	Hostboot Time
NORMAL	-	0.27955	-	0.26748	24.777
XOR	100%	0.52089	0.01030	0.13118	25.096
	50%	0.51726	0.00574	0.13094	25.081
	25%	0.51645	0.00559	0.13009	25.049
SPECK	100%	0.59249	0.08755	0.13105	25.077
	50%	0.55308	0.04457	0.12630	29.959
	25%	0.53777	0.02281	0.13098	25.008
AES	100%	1.0439	0.53969	0.12878	25.557
	50%	0.78171	0.27139	0.13234	25.298
	25%	0.64457	0.13670	0.12878	25.119

Skiboot which helps put the performance impact of boot image encryption in perspective. The median Hostboot time for AES was 25.51 sec while the time for a normal boot was 24.69 sec. This means that it took Hostboot 0.86 seconds longer to run by protecting the entire Skiboot firmware section with AES-256. The SPECK implementation had a median Hostboot execution time of 24.99 sec while XOR had a median of 25.06 sec. It is interesting that SPECK performed slightly better than XOR by 0.07 seconds which was not a statistically significant difference when conducting a two sample t-test (p=0.632). This may be caused by the implementation of the XOR function as each Byte in Skiboot is iterated through and XORed while SPECK loads and decrypts Skiboot in 8 Byte chunks.

The difference in performance overhead can be observed better in Figure 12 where the time for the `load_payload` function in ISTEP 20 is shown for each encryption method. The differences in the Hostboot execution times are caused by the execution of the load Skiboot step which contains the decryption function. The median time for AES took 1.0443 sec, SPECK took 0.5923 sec, XOR took 0.5193 sec, and unencrypted firmware took 0.2796. The variation in Skiboot loading times are much smaller than

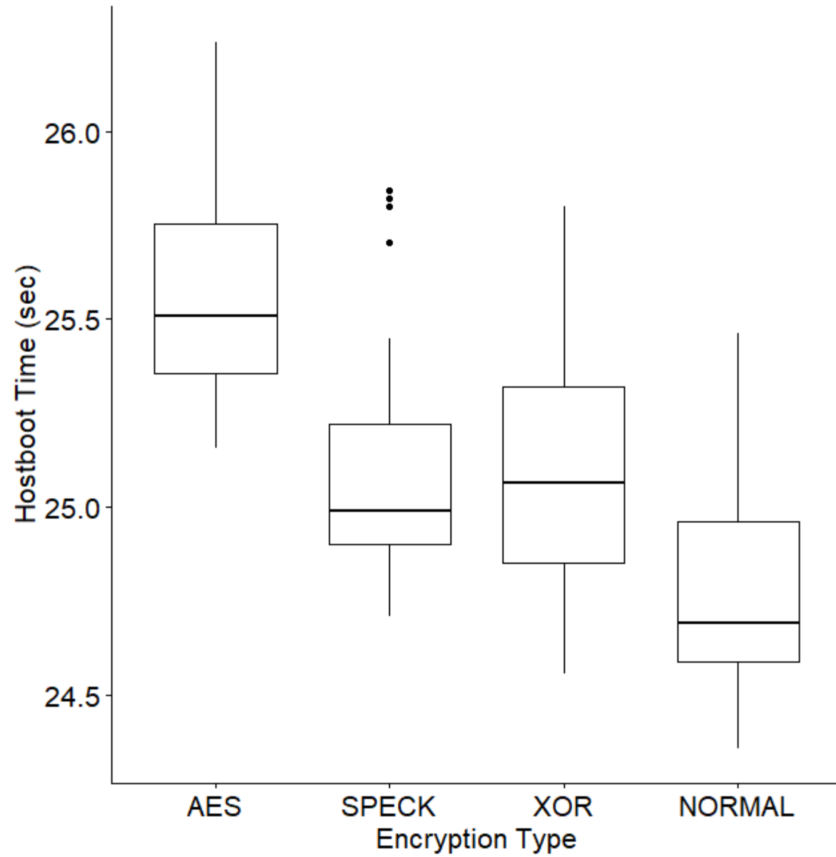

Figure 11. Hostboot Execution Time

the hostboot execution times in Figure 11 which may be due to the earlier ISTEPs in Hostboot introducing variability.

A one way Analysis of Variance (ANOVA) was used to analyze the boot timing data to see if there is a significant difference between the different encryption methods. The first ANOVA test was run on the Hostboot execution time revealing that the boot time had a significant difference between the different factors. The second ANOVA was run on execution time to load Skiboot revealing that was also a significant difference between the different encryption options. These tests help answer the research question showing that there is a performance difference between each

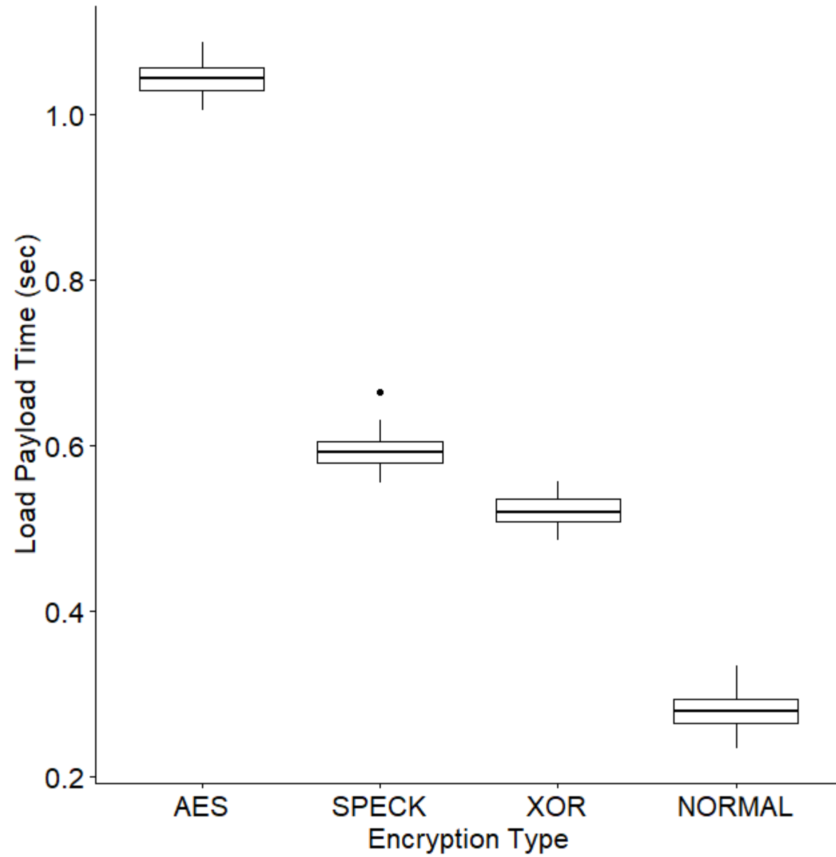

Figure 12. ISTEP 20.1 Execution Time

encryption method and the unencrypted Skiboot implementation.

Figure 13 shows the time taken for the decompression of Skiboot with boot image encryption. In the PNOR without Skiboot encryption, the decompression code must wait for the compressed Skiboot section to get transferred into the working memory which takes 0.2654 seconds on average. In the decryption code for the encrypted Skiboot firmware, memory is allocated for the decryption to execute before the decompression occurs. The average decompression time for encrypted firmware takes 0.1281 seconds which is 0.1373 seconds faster than the stock boot image. This helps offset the decryption time by handling the PNOR image data transfer in the

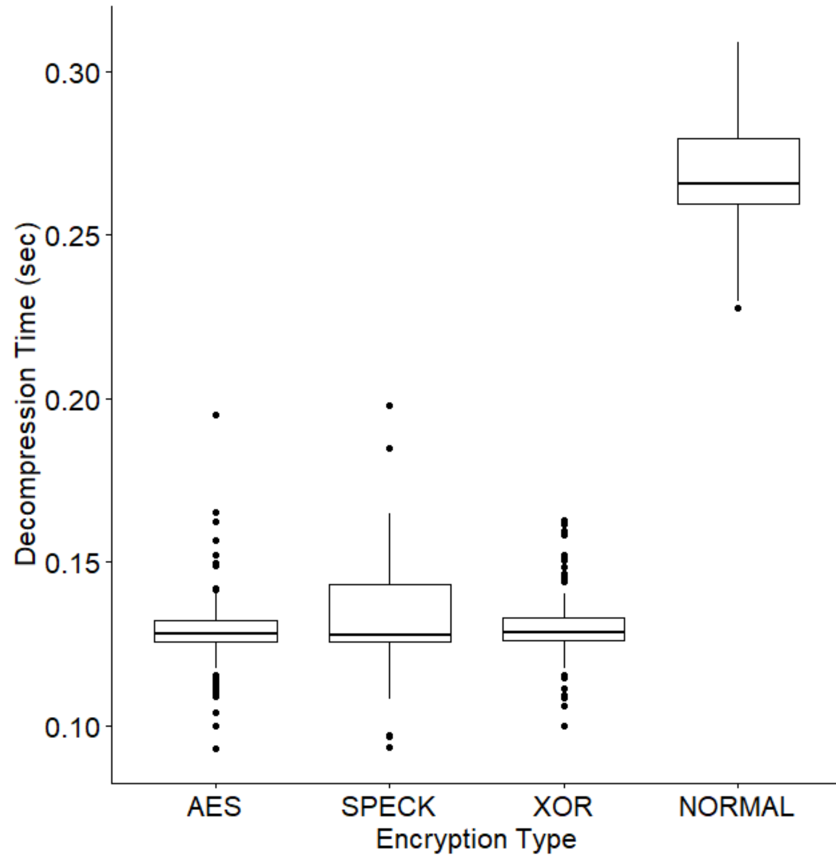

Figure 13. Skiboot Decompression Time

decryption stage instead of the decompression stage.

5.4.2 Comparison of Encryption Methods

The three types of ciphers were implemented in this research to demonstrate the range of encryption that the P9 would be able to handle during the IPL. The Skiboot decryption times for each encryption type is shown in Figure 14. The XOR cipher takes an average of 0.0103 seconds which shows the efficiency of XOR operations. This implementation would be efficient in a system that required minimal boot firmware execution delay while supporting MBs of free memory to store the one time pad

key. SPECK also ran efficiently with decryption taking 0.0875 seconds. Although the SPECK cipher is not as secure as AES, it provides a layer of protection while decrypting Skiboot around 6 times as quickly. AES took the longest to decrypt but provides the most secure encryption implementation with an average decryption time of 0.5397 seconds. An ANOVA test showed that the implementation of each algorithm resulted in a statistically significant boot time increase.

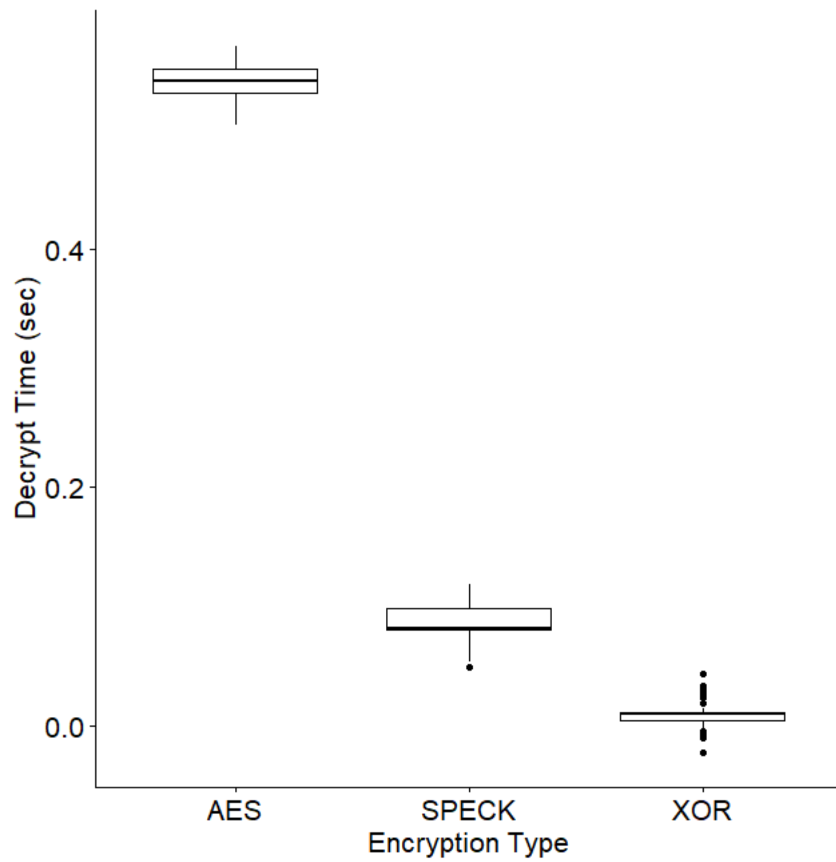

Figure 14. Skiboot Decryption Time

5.4.3 Encryption Ratio

Encryption ratio is shorthand for describing how much of Skiboot is subject to encryption. Figure 15 displays the Skiboot decryption time for each encryption type and ratio. The XOR cipher performance for each encryption ratio shows that the amount encrypted does not affect the overall performance by a significant amount. The increase from 25% to 100% only increased the Skiboot decryption time by 4.71 ms. The SPECK and AES implementations both show a linear increase in decryption time as the encryption ratio increases. AES takes longer to decrypt per byte compared to SPECK which is observable in Figure 15. AES takes 5.43 ms for each percentage of Skiboot that is encrypted while SPECK takes 0.89 ms for each additional percent encrypted. Skiboot is 1 MB, one percent is 10,486 bytes, so AES takes 518 ns per encrypted byte of Skiboot, SPECK takes 89 ns per encrypted byte, and XOR takes 4.71 ns per encrypted byte.

Based on these results, an estimation of the performance for a hypothetical implementation of a complete encryption of the PNOR image could be done. Encrypting the entire 64 MB image with XOR would increase the boot time by 0.3161 seconds, SPECK would add 5.973 seconds, and AES would add 34.76 seconds.

A two way ANOVA was used to test if there is a significant difference between each encryption method and encryption ratio. The results in Figure 16 revealed that there was a significant difference between the boot timing between each encryption implementation and the ratio of Skiboot that was encrypted. This means that it may be worthwhile to investigate the possibility of only encrypting the section of firmware that contains sensitive information. For example, information or keys that need to be secured within the IPL may not require the entire firmware section to be encrypted. Isolating and encrypting a smaller percentage of the firmware would be meaningful in an implementation scenario where a slight delay in boot time is consequential.

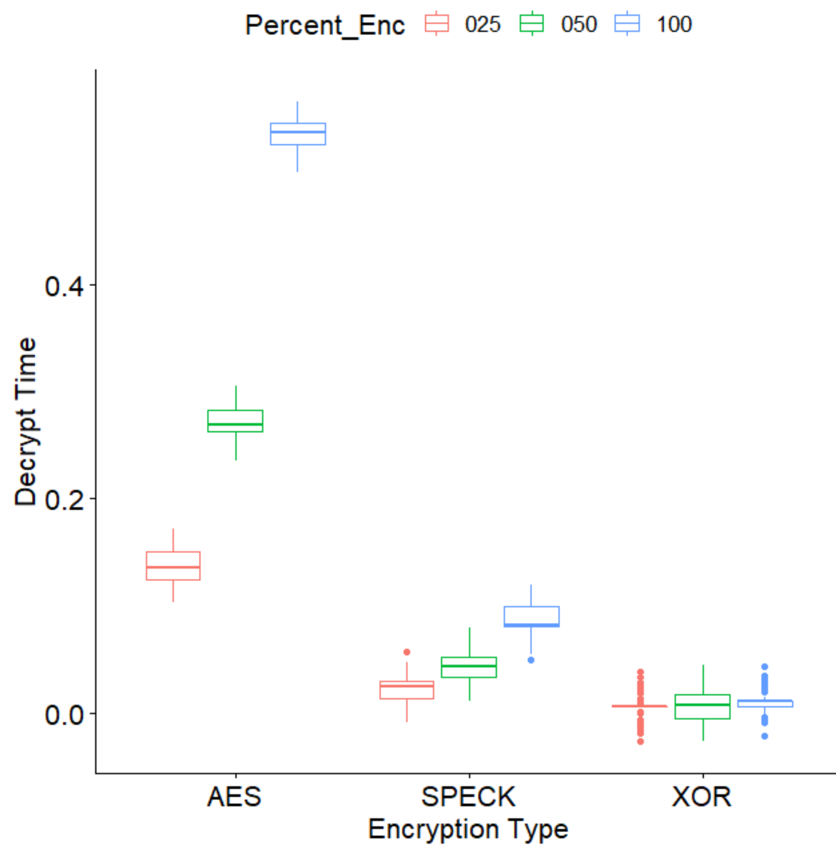

Figure 15. Skiboot Decryption Time by Encryption Ratio

5.5 Challenges

There were several challenges that were encountered when progressing through this research. The first was the lack of detailed documentation regarding the interaction between the hardware and firmware, and the documentation within the code to explain the functionality of each section of code. Since this research requires a thorough understanding of the IPL, the lack of completed documentation created obstacles for analyzing the firmware. For example, the `readme.txts` for most of the firmware sections were empty, limiting the ease of access for firmware development.

```
             Df Sum Sq Mean Sq F value Pr(>F)
Percent_Enc   2  535.6  267.78   208.0 <2e-16 ***
Enc_Type      2  448.8  224.39   174.3 <2e-16 ***
Residuals   894 1150.9    1.29
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Figure 16. Encryption ANOVA Test

The second challenge in this research was the limitations with debugging during the IPL. Since this research focused on the development of the boot firmware encryption, the process of debugging the decryption firmware was important. There was no easily accessible method of predicting that the decryption function would work correctly before running the IPL and observing a successful boot sequence. The console client helps by allowing data during the later stages of the IPL to be printed out to the console logs. This was used to help verify that the XZ decompression header matched the expected values during the decryption process.

The last major challenge was the risk of bricking the P9. Since the Talos II uses two sides for the SEEPROM, the risk of bricking the processor is low but still a possibility. At the beginning of the IPL, the SEEPROM Side 0 is updated with the new SBE firmware. If the system is able to reach ISTEP 10, the SBE firmware change will be saved into the SEEPROM Side 1. This means that the Talos II verifies that the new firmware can boot successfully at least once before permanently updating the SBE. There is still the possibility of the firmware functioning correctly for a single boot sequence and failing upon consecutive boot ups. This would cause both sides of the SEEPROM to contain faulty firmware which will prevent the system from booting past the HBBL. This challenge can be mitigated by altering the SBE update firmware. By default, the Talos II boots in Sequential mode which is how the SEEPROM gets updated as explained previously. This can be changed to Independent mode which

only updates the SEEPROM Side 0 and does not alter the second side.

5.6 Summary

This chapter discussed the findings and results of the scenarios presented in Chapter IV. The evaluation of the secure key management implementation was discussed along with a memory analysis of the various encryption methods. The performance experiment results include the performance impact of boot image encryption, the encryption algorithm comparison, and the analysis of the encryption ratio.

VI. Conclusion

6.1 Overview

This chapter summarizes the work performed for this research including the design and development of boot firmware encryption and secure key management. It reiterates contributions of the work and summarizes the observations and analysis of the experiment. Areas of future work for this research effort are also covered with a plan described for each of the areas.

6.2 Summary

The primary goal of this research was to develop a method of protecting the boot flash image, securing any proprietary information loaded onto commodity hardware. In this research, the commodity hardware was the Talos II which supports open source firmware. This allows developers to customize the firmware and alter boot functionality, allowing for the possibility of protecting proprietary information through firmware. The work in this paper has shown that the Talos II is capable of decrypting firmware during the early stages of the IPL by carefully designing around the limitations of the hardware.

This was accomplished by identifying locations in firmware that connected the various sections of the IPL. The first section selected was the HBI which loads Skiboot. The decryption firmware was loaded into the HBI and Skiboot was encrypted prior to being compiled and flashed onto the Talos II. The experiment tested the performance of three different encryption schemes and compared them to the standard PNOR image.

The results from the experiment demonstrate that even in the case of a cryptographically strong algorithm, such as AES, encryption on a portion of the boot image

is cost efficient. SPECK would work effectively in an implementation where minimizing boot time or memory usage is critical. Since a system only needs to boot once in most implementations, a 0.86 second delay to incorporate AES-256 on Skiboot to the boot firmware is negligible in most circumstances. The evaluation of encryption ratios also shows that there is a constant cost for encrypting each additional byte. Since there is a significant performance difference between the encryption ratios, reducing the amount of firmware encrypted may be required in systems with larger boot firmware images.

This boot firmware encryption implementation is secured through the encryption of the firmware decryption keys. The root of trust for the secure key management exists in the SBE where it is assumed that the plaintext firmware stored in the SEEPROM is secure. The keys to decrypt the following firmware stage are hardcoded into the source code but will be protected since the entire firmware section will be encrypted through the boot firmware encryption implementation. This should be changed in a final implementation to provide an additional layer of security of the decryption keys.

6.3 Research Contributions to Hardware Security

Boot Firmware Encryption: The successful implementation of the PNOR encryption methodology in this research demonstrated that it is possible to add a layer of security over the IPL without changing the hardware. The encrypted boot instructions are decrypted in the P9 and stored in the L3 cache, making it extremely difficult for an adversary to obtain the plaintext firmware. The ability to alter boot data during the IPL showed that the capabilities of the P9 are only limited by the memory space available during the early stages of the boot process.

Memory Analysis of Decryption Firmware: This analysis enumerated the amount

of memory required for each decryption algorithm and discusses the most efficient options for a variety of scenarios.

Secure Key Management: This effort successfully demonstrated a successful implementation of a secure key management approach. This implementation creates the framework to encrypt the entire PNOR image and store the keys securely within the processor. Since only firmware is added to the system, additional hardware like a One-Time Pad storage chip is not required.

Performance Analysis: This research demonstrated an approach for measuring the performance of various decryption algorithms during the Talos II boot sequence. The measurements are collected with varying encryption ratios to identify the performance limitations with each encryption algorithm.

Qualitative Analysis: This analysis provided design considerations, challenges, and potential vulnerabilities of the boot firmware encryption implementation described.

6.4 Future Work

There are several areas where this research can be matured, expanded and further developed. Listed below are topics of interest that would expand the scope of this research:

6.4.1 Full boot image encryption

The first area of future work is the completion of boot image encryption on all firmware sections in the PNOR image. This research demonstrates a successful implementation of boot image encryption on HBB and Skiboot firmware sections. For the secure key management method to be considered secure, the entire PNOR image should be encrypted. This would require further investigation on the encryption of

HBI, Hostboot Data, OCC, and Petitboot firmware. The code responsible for transporting the specific PNOR sections will need to be located and the decryption code will be added to the existing firmware. For the encryption of the HBI, the HBB section contains the firmware that will require the decryption function. Since the HBI uses a VFSRP, only the modules that need to be initialized are loaded into memory. This could cause possible problems with encryption and decryption for the modules because each module in the HBI will need to be encrypted individually because the HBB firmware will decrypt each module individually.

There are possible limitations when investigating full PNOR encryption. The first possible problem is limited computational power available at the early boot stages which would cause a significant performance impact during boot up. Another possible limitation is the use of Random Access Memory (RAM) in the later boot stages. Since data traversing outside the P9 is considered vulnerable to adversaries, firmware running out of RAM will be stored as plaintext and will not be secure.

6.4.2 Compatibility with IBM Secure Boot

The next area of research is to ensure compatibility of the boot image encryption implementation with IBM's secure boot utility. The boot firmware encryption research implementation is not designed to be used with IBM's secure boot. When developing and testing the custom firmware, the secure boot jumper was not set, so the secure boot utility was not used. The secure key management structure along with firmware decryption stages is able to prevent unauthorized firmware from booting on the Talos II. This works because overwriting portions of firmware in the PNOR will also overwrite the encryption, causing the system to fail to boot when the decryption code runs. Although the firmware decryption key management can prevent unauthorized firmware from booting on the system, the secure boot utility can lock

the SEEPROM, preventing BMC from accessing the plaintext HBBL firmware [18]. Since the secure boot utility runs additional instructions during the IPL, there should not be significant conflicts with the firmware encryption and secure boot systems.

6.4.3 Application to Intel or AMD systems

Finally, the last area of future work is to apply the boot firmware encryption method to Intel and AMD systems. Since both system manufacturers use BIOS or UEFI with bootloaders, the structure of the IPL should be similar. With access to detailed documentation and the source code of the boot firmware, implementing the decryption code should be straightforward. Aside from IPL structure, the ability to alter the firmware in the flash chip is essential to getting the firmware encryption to work. Having access to the firmware source code is required for developing this implementation on new architectures.

On Intel and AMD computing systems, a BootROM, PreLoader, and Bootloader are traditionally used to boot up the OS as shown in Figure 17. The bootloader on x86 platforms works similarly to the SBE firmware where it is decomposed into at least two stages [19]. The first stage consists of machine code binary on the Master Boot Record (MBR) that locates the second stage boot loader and loads it into memory. The second stage boot loader handles the Power-On Self-Test (POST) screen for the user, showing the OS or kernels available for boot [19]. It is also responsible for locating and loading the OS kernel into the memory. On AMD processor systems, there is a secure processor called the Platform Security Processor (PSP) that executes immutable on-chip boot ROM, locates the Embedded Firmware Table and PSP Directory Table in the Serial Peripheral Interface ROM at system power on [20]. It then verifies and executes the PSP off-chip bootloader. Although x86 architecture systems perform a similar boot sequence to OpenPOWER systems, the usage of BIOS and UEFI could

be a possible obstacle with the compatibility of boot firmware encryption.

Figure 17. Intel Boot Flow [4]

6.4.3.1 Legacy BIOS Boot Flow

BIOS is low level software that provides critical configuration and booting services from the moment of powering on a computer until the kernel's `main()` routine begins. When the computer system boots, the processor searches for and executes the BIOS at the end of system memory [19]. The BIOS is written into read-only permanent memory because it controls the first step of the boot process. It also provides the lowest level interface to peripheral devices. The BIOS then loads the bootloader at `0x7C00` as well as the partition table stored in the MBR and transfers control of the boot process to the bootloader [21].

Applying boot firmware encryption to the legacy BIOS initialization will be difficult due to the early stages of the BIOS boot flow. The processor is pre-programmed to always search for the BIOS at the end of system memory in Read-Only Memory (ROM). Once the processor executes the instructions to jump to the startup program, the BIOS starts performing the POST. Implementing the boot firmware encryption requires the decryption instructions to be stored within the processor module so the data sent to the processor can be decrypted. If the pre-programmed instructions to locate the BIOS cannot be altered, the first section of the BIOS will need to be stored in plaintext which complicates the problem of securing the firmware decryption keys.

If researchers are able to flash new instructions to the CPU, the BIOS can be encrypted beforehand and decrypted in the processor. The BIOS could then be altered to include a decryption function for the specific bootloader used in the following portion of the IPL. Another foreseeable problem is the use of a boot drive and RAM for the BIOS. Since the assumption that data traversing on buses external to the processor is not secure, the BIOS data stored in the RAM and boot drive must be encrypted for the chain of firmware encryption to perform correctly.

6.4.3.2 UEFI Boot Flow

BIOS has existed for over 40 years and has many limitations with modern computer systems. For example, BIOS requires the processor to run in 16-bit mode, and only has 1 MB of space to execute in [22]. This causes hardware initialization to run slowly, especially when multiple hardware devices need to be initialized. This led to the development of UEFI to keep up with advancements in computer systems. UEFI has a few advantages over BIOS as it supports boot drives larger than 2 TB with more than 4 partitions on a drive. It also enables faster booting and more efficient power management. Legacy BIOS based systems and UEFI based systems both come with the BIOS ROM which contains the instructions to perform the initial power-on sequence and jump to the bootloader [22]. UEFI is stored on non-volatile memory and runs instead of legacy BIOS. It also eliminates the need for a second stage bootloader by using a mountable file system. Although there are differences between BIOS and UEFI with the bootloader process, there is not a significant difference during the beginning stages of the IPL.

6.5 Conclusion

This research demonstrated that encrypting boot firmware and decrypting it during the IPL is feasible and adds a layer of protection over the firmware stored in the boot flash chip. It also demonstrated a method of securing the keys used in the firmware decryption stage and outlines an implementation that would work alongside secure boot. A successful implementation of this research would enable the system to boot in a vulnerable environment without leaking firmware details. The evaluation of the performance with the decryption code also showed that the additional layer of security did cause a significant performance impact to ISTEP 20, but the effect was marginal when comparing the impact to the entire IPL. Boot firmware encryption can be expanded to other architectures which will allow more proprietary hardware to secure the sensitive information.

Appendix A. XOR100 Decryption Firmware

This is to explain about the code...

```
const uint32_t BLOCK_SIZE = 4096;
for (uint32_t i=0;i<originalPayloadSize;i+=BLOCK_SIZE) {
    memcpy( reinterpret_cast<void*>(
            reinterpret_cast<uint64_t>(temp_buf) + i ),
        reinterpret_cast<void*>( pnorSectionInfo.vaddr + i ),
        std::min( originalPayloadSize - i, BLOCK_SIZE ) );
}

for ( uint64_t i = 0; i < originalPayloadSize; i++ ) {
    temp_buf[i] ^= 0x55;
}
```

Appendix B. XOR100 Encryption Script

```
1  infile = 'A:/Calvin Muramoto/Documents/Talos PNOR Images/
      PNOR_XOR_100/preXOR100_payload_talos.pnor'
2  outfile = 'A:/Calvin Muramoto/Documents/Talos PNOR Images/
      PNOR_XOR_100/XOR100_payload_talos.pnor'
3
4  with open(infile, 'rb') as input:
5      image = input.read()
6
7  print(f'Original file size: {len(image)}')
8
9  part_start = 0x21a2000
10 part_end = part_start + 0xFF000
11
12 pre = image[: part_start]
13 print(f'Pre-Payload: {len(pre)}')
14 payload = image[part_start: part_end]
15 print(f'Payload size: {len(payload)}')
16 post = image[part_end: ]
17 print(f'Post-size: {len(post)}')
18
19 L = list(payload)
20 print(range(len(L)))
21 for i in range(len(L)):
22     L[i] = L[i] ^ 0x55
23
24 final_image = pre + bytes(L) + post
25
26 print(f'End Addr: {part_end:08x}')
27 print(f'New file size: {len(image)}')
28
29 with open(outfile, 'wb') as write_file:
30     write_file.write(final_image)
```

Appendix C. SPECK100 Decryption Firmware

The SPECK decryption firmware uses code from the SIMON and SPECK Implementation Guide. The supporting decryption functions can be found in this guide [11].

```
1  uint8_t* temp_buf = (uint8_t*) malloc(originalPayloadSize);
2  const uint32_t BLOCK_SIZE = 4096;
3  for (uint32_t i=0;i<originalPayloadSize;i+=BLOCK_SIZE) {
4      memcpy(reinterpret_cast<void*>(
5              reinterpret_cast<uint64_t>(temp_buf) + i),
6          reinterpret_cast<void*>(pnorSectionInfo.vaddr + i),
7          std::min(originalPayloadSize - i, BLOCK_SIZE ));
8  }
9
10 for ( uint64_t i = 0; i < originalPayloadSize / 8; i++ ) {
11     /***************************************************
12     *    Prep Cipher text
13     ***************************************************/
14     int ct_numbytes = 8;
15     uint8_t ct_bytes[ct_numbytes];
16     for(uint32_t j = 0; j < 8; j++) {
17         ct_bytes[j]=temp_buf[(8 * i) + j];
18     }
19     // Convert byte array to 32bit words
20     int ct_length = ct_numbytes / 8;
21     uint32_t ct_words[ct_length];
22     BytesToWords32(ct_bytes, ct_words, ct_numbytes);
23
24     /***************************************************
25     *    Prep Key
26     ***************************************************/
27     // 12 Byte = 96 Bit key
28     uint8_t key[] = {0x00,0x01,0x02,0x03,0x08,0x09,0x0a,0x0b,0
          x10,0x11,0x12,0x13};
29     int key_numbytes = sizeof(key);
30     uint32_t key_words[key_numbytes];
31     BytesToWords32(key, key_words, key_numbytes);
32
33     /***************************************************
34     *    Key Schedule
35     ***************************************************/
36     uint32_t rk[26];
37     Speck6496KeySchedule(key_words, rk);
38
39     /***************************************************
40     *    Decrypt
41     ***************************************************/
42     uint32_t pt_words[ct_length];
43     Speck6496Decrypt(pt_words, ct_words, rk);
44
45     uint8_t pt_bytes[ct_numbytes];
46     Words32ToBytes(pt_words, pt_bytes, 2);
47     for(uint32_t j = 0; j < 8; j++) {
48         temp_buf[(8 * i) + j]=pt_bytes[j];
49     }
50 }
```

Appendix D. SPECK100 Encryption Script

This script shows the process of encrypting a section of the PNOR with SPECK. The supporting encryption functions are not displayed but can be found on github: https://github.com/inmcm/Simon_Speck_Ciphers.

```
1  import binascii
2
3  infile = 'A:/Calvin Muramoto/Documents/Talos PNOR Images/
       PNOR_SPECK_100/preSPECK100_payload_talos.pnor'
4  outfile = 'A:/Calvin Muramoto/Documents/Talos PNOR Images/
       PNOR_SPECK_100/SPECK100_payload_talos.pnor'
5
6  with open(infile, 'rb') as input:
7      image = input.read()
8
9  print(f'Original file size: {len(image)}')
10
11 part_start = 0x21a2000
12 part_end = part_start + 0xFF000
13
14 pre = image[: part_start]
15 print(f'Pre-Payload: {len(pre)}')
16 payload = image[part_start: part_end]
17 print(f'Payload size: {len(payload)}')
18 post = image[part_end: ]
19 print(f'Post-size: {len(post)}')
20
21 tempkey = 0x131211100b0a090803020100
22 my_speck = SPECK(64, 96, tempkey)
23
24 L = list(payload)
25 for i in range(0, len(L), 8):
26     tempPayload = L[i : i + 8]
27     block = int.from_bytes(tempPayload, 'little', signed =
           False)
28     encrypted = my_speck.encrypt(block)
29     L[i : i + 8] = encrypted.to_bytes(8, 'little')
30
31 final_image = pre + bytes(L) + post
32
33 print(f'End Addr: {part_end:08x}')
34 print(f'New file size: {len(final_image)}')
35
36 with open(outfile, 'wb') as write_file:
37     write_file.write(final_image)
```

Appendix E. AES100 Decryption Firmware

This appendix contains the firmware decryption code in ISTEP 20. The supporting AES decryption code and sboxes are from https://github.com/kokke/tiny-AES-c.

```
1  uint8_t* temp_buf = (uint8_t*) malloc(originalPayloadSize);
2
3  const uint32_t BLOCK_SIZE = 4096;
4  for (uint32_t i=0;i<originalPayloadSize;i+=BLOCK_SIZE) {
5      memcpy( reinterpret_cast<void*>(
6              reinterpret_cast<uint64_t>(temp_buf) + i ),
7          reinterpret_cast<void*>( pnorSectionInfo.vaddr + i ),
8          std::min( originalPayloadSize - i, BLOCK_SIZE ) );
9  }
10
11 /*************************************************
12 *    Prep Key and Initialization Vector
13 *************************************************/
14 uint8_t key[] = { 0x60, 0x3d, 0xeb, 0x10, 0x15, 0xca, 0x71, 0
      xbe, 0x2b, 0x73, 0xae, 0xf0, 0x85, 0x7d, 0x77, 0x81, 0x1f,
      0x35, 0x2c, 0x07, 0x3b, 0x61, 0x08, 0xd7, 0x2d, 0x98, 0x10,
       0xa3, 0x09, 0x14, 0xdf, 0xf4 };
15 uint8_t iv[]  = { 0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0
      x07, 0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f };
16 struct AES_ctx ctx;
17 AES_init_ctx_iv(&ctx, key, iv);
18
19 for (uint64_t i = 0; i < originalPayloadSize / 64; i++)
20 {
21     /*************************************************
22      *    Prep Cipher text
23      *************************************************/
24     int ct_numbytes = 64;
25     uint8_t ct_bytes[ct_numbytes];
26     for (uint32_t j = 0; j < 64; j++)
27         ct_bytes[j] = temp_buf[(64 * i) + j];
28
29     /*************************************************
30      *    Decrypt
31      *************************************************/
32     AES_CBC_decrypt_buffer(&ctx, ct_bytes, 64);
33
34     for (uint32_t j = 0; j < 64; j++)
35         temp_buf[(64 * i) + j] = ct_bytes[j];
36 }
```

Appendix F. AES100 Encryption Script

This script shows the process of encrypting a section of the PNOR with AES using functions from Crypto.Cipher. The supporting documentation is located on pycryptodome: https://pycryptodome.readthedocs.io/en/latest/src/cipher/aes.html

```
1  import binascii
2  import json
3  from base64 import b64encode
4  from Crypto.Cipher import AES
5  from Crypto.Util.Padding import pad
6  from Crypto.Util.Padding import unpad
7
8  infile = 'A:/Calvin Muramoto/Documents/Talos PNOR Images/
      PNOR_AES_100/preAES100_payload_talos.pnor'
9  outfile = 'A:/Calvin Muramoto/Documents/Talos PNOR Images/
      PNOR_AES_100/AES100_payload_talos.pnor'
10
11 with open(infile, 'rb') as input:
12     image = input.read()
13
14 print(f'Original file size: {len(image)}')
15
16 part_start = 0x21a2000
17 part_end = part_start + 0xFF000
18
19 pre = image[: part_start]
20 print(f'Pre-Payload: {len(pre)}')
21 payload = image[part_start: part_end]
22 print(f'Payload size: {len(payload)}')
23 post = image[part_end: ]
24 print(f'Post-size: {len(post)}')
25
26 temp_key = b'\x60\x3d\xeb\x10\x15\xca\x71\xbe\x2b\x73\xae\xf0\
      x85\x7d\x77\x81\x1f\x35\x2c\x07\x3b\x61\x08\xd7\x2d\x98\x10
      \xa3\x09\x14\xdf\xf4'
27 temp_iv = b'\x00\x01\x02\x03\x04\x05\x06\x07\x08\x09\x0a\x0b\
      x0c\x0d\x0e\x0f'
28 cipher = AES.new(temp_key, AES.MODE_CBC, iv = temp_iv)
29
30 L = list(payload)
31 for i in range(0, len(L), 64):
32     tempPayload = L[i : i + 64]
33     encrypted = cipher.encrypt(unpad(pad(bytes(tempPayload),
           AES.block_size), AES.block_size))
34     L[i : i + 64] = encrypted
35
36 final_image = pre + bytes(L) + post
37
38 print(f'End Addr: {part_end:08x}')
39 print(f'New file size: {len(final_image)}')
40
41 with open(outfile, 'wb') as write_file:
42     write_file.write(final_image)
```

Appendix G. Talos Control Firmware

```
1  import time
2  import json
3  import argparse
4
5  import os
6  import contextlib
7
8  # Dependencies:
9  #        pip3 install dlipower
10 #        pip3 install paramiko
11 #
12 import dlipower
13 import paramiko
14
15
16
17 def get_devs(config_filename):
18
19         with open(config_filename, 'r') as infile:
20                 devs = json.loads(infile.read())
21
22         return devs
23
24
25
26 ### Create a PowerSwitch object using the provided device
       credentials
27 ### devs should be retrieved using get_devs(config_filename)
28 ###
29 ### Usage:
30 ###         config_filename = 'local/talos_config.json'
31 ###         devs = get_devs(config_filename)
32 ###         switch = get_switch(devs)
33 ###
34 ### NOTE: I was going to make this a context manager, but
       PowerSwitch doesn't
35 ###        have any close or cleanup function.
36 ###
37 def get_switch(devs):
38         ip = devs['power']['ip']
39         user = devs['power']['user']
40         pw = devs['power']['pw']
41
42         return dlipower.PowerSwitch(hostname=ip, userid=user,
              password=pw)
43
44 def send_bmc_cmds(devs, commands):
45         ip = devs['bmc']['ip']
46         user = devs['bmc']['user']
47         pw = devs['bmc']['pw']
48
49         start = time.time()
50         timeout = 180
51         print('Attempting to connect to the BMC')
52         while time.time() - start < timeout:
```

```
53                  try:
54                          # This suppresses error messages from
                               Paramiko ssh
55                          # Source: https://stackoverflow.com/a
                               /46129367
56                          with open(os.devnull, 'w') as f,
                               contextlib.redirect_stderr(f):
57                                  # Source: https://www.
                                       thepythoncode.com/article/
                                       executing-bash-commands-
                                       remotely-in-python
58                                  #
59                                  ssh = paramiko.SSHClient()
60                                  ssh.
                                       set_missing_host_key_policy
                                       (paramiko.AutoAddPolicy())
61
62                                  ssh.connect(ip, username=user,
                                        password=pw, allow_agent =
                                        False, timeout = 3)
63
64                                  for command in commands:
65                                          print(command)
66                                          stdin, stdout, stderr
                                               = ssh.exec_command(
                                               command)
67                                          print(stdout.read().
                                               decode())
68
69                                          err = stderr.read().
                                               decode()
70                                          if err:
71                                                  print(err)
72
73                                  # TODO: check stdout.read()
                                       result to see if it
                                       succeeded
74                          print(stdout.read().decode())
75                          print(f'BMC took {time.time() - start}
                                seconds to boot')
76                          return True
77                  except Exception as e:
78                          pass #print(e)
79
80          print('Failed to connect to the BMC')
81          return False
82
83  # TODO: Fix gpio-fsi driver bind
84  #
85  #
86  def enable_bmc(devs):
87          print('Enabling BMC')
88
89          # List of commands that will be sent through ssh
90          #
91          commands = [
92                  'devmem 0x1e6e20ac 32 0x000000FF',
93                  #'echo gpio-fsi > /sys/bus/platform/drivers/
```

```
                    fsi-master-acf/bind',
94                  'devmem 0x1e7801e4 32 0x00050000',
95                  'devmem 0x1e780004 32 0x19000040',
96                  'devmem 0x1e6e2090 32 0x07FF0000'
97          ]
98
99          send_bmc_cmds(devs, commands)
100
101 def disable_bmc(devs):
102         print('Disabling BMC')
103
104         # List of commands that will be sent through ssh
105         #
106         commands = [
107                 'devmem 0x1e6e20ac 32 0x00000000',
108                 'echo gpio-fsi > /sys/bus/platform/drivers/fsi
                    -master-acf/unbind',
109                 'devmem 0x1e7801e4 32 0x00000000',
110                 'devmem 0x1e780004 32 0x19000000',
111                 'devmem 0x1e6e2090 32 0x023F0000'
112         ]
113
114         send_bmc_cmds(devs, commands)
115
116 def power_off(switch):
117         print('Powering off Talos II')
118         # This seems like a better method, but the index it
              returns is 1 higher
119         # than it should be. Further testing is needed to see
              if we can use this
120         #       out = switch.determine_outlet('Talos_II')
121         #       switch[out-1].off()
122         for outlet in switch:
123                 if 'Talos' in outlet.name:
124                         return outlet.off()
125
126 def power_on(switch):
127         print('Powering on Talos II')
128         # This is slow, but it works
129         for outlet in switch:
130                 if 'Talos' in outlet.name:
131                         return outlet.on()
132
133 def power_reset(switch):
134         power_off(switch)
135         # TODO: Find a better method
136         print('Power off, waiting 10 seconds')
137         time.sleep(10.0)
138         power_on(switch)
139
140 if __name__ == "__main__":
141
142         DEFAULT_CONFIG = 'local/talos_config.json'
143
144         examples = '''
145         Example commands:
146
147         '''
```

```
148
149         parser = argparse.ArgumentParser(description='
               heist_ctrl.py', epilog=examples, formatter_class=
               argparse.RawTextHelpFormatter)
150
151         parser.add_argument('-c', '--config', dest='config',
               metavar='FILENAME', default=DEFAULT_CONFIG, type=
               str, help=f'JSON config file with device
               information (default={DEFAULT_CONFIG})')
152
153         parser.add_argument('-r', '--reset', dest='reset',
               default=False, action='store_true', help='Reset the
                power of the Talos II (power off then power on)')
154
155         parser.add_argument('-p', '--poweroff', dest='poweroff
               ', default=False, action='store_true', help='Power
               on the Talos II if not already on')
156         parser.add_argument('-P', '--poweron', dest='poweron',
                default=False, action='store_true', help='Power on
                the Talos II if not already on')
157
158         parser.add_argument('-b', '--bmc_disable', dest='
               bmc_disable', default=False, action='store_true',
               help='Disable the BMC interfaces via SSH')
159         parser.add_argument('-B', '--bmc_enable', dest='
               bmc_enable', default=False, action='store_true',
               help='Enable the BMC interfaces via SSH')
160
161         args = parser.parse_args()
162
163         config_filename = args.config
164         devs = get_devs(config_filename)
165         switch = get_switch(devs)
166
167         if args.reset:
168                 power_reset(switch=switch)
169
170         if args.poweroff:
171                 power_off(switch=switch)
172
173         if args.poweron:
174                 power_on(switch=switch)
175
176         if args.bmc_disable:
177                 disable_bmc(devs)
178
179         if args.bmc_enable:
180                 enable_bmc(devs)
```

Appendix H. UDP Logger Firmware

```
1  #####
2  ### This module provides logging for UDP messages received
     from the uC
3  ###
4  ### Multiple UDP ports are open and each port gets logged to a
      different file
5  ### TODO: List of ports and purposes
6  #####
7  import threading
8  import argparse
9  import socket
10 import queue
11 import time
12 import os
13
14 # ref:
15 #       https://stackoverflow.com/questions/36760127/how-to-
     use-the-new-support-for-ansi-escape-sequences-in-the-
     windows-10-console
16
17 from sys import platform
18
19 if platform == 'win32':
20         import ctypes
21         kernel32 = ctypes.windll.kernel32
22         kernel32.SetConsoleMode(kernel32.GetStdHandle(-11), 7)
23
24 start_time = time.time()
25
26 ### Print debug messages from PNOR server
27 ### Warning: this causes delays that may prevent
28 ### the system from booting. Only use for testing
29 ###
30 DEBUG = False
31
32 LOG_DIR = 'local/'
33
34 ### ports 1000 - 10002 == info, warn, error
35 ### port  1016 == P9 messages (0x3f8)
36 ###
37 ports = [1000, 1001, 1002, 1003, 1016]
38 filenames = ['log_info.txt', 'log_warn.txt', 'log_error.txt',
     'log_lpc.txt', 'log_p9_msgs.txt']
39
40 aggregate_filename = 'log_all.txt'
41 aggregate_lock = threading.Lock()
42
43 pnor_log_file = 'pnor_accesses.txt'
44 pnor_port = 999
45 BLOCK_SIZE = 4096
46 pnor_size = 67108864
47
48 pnor_log_q = queue.Queue()
49
50 # Check the beginning of each log message and remove these
     prefixes from log file.
```

```
51  #
52  prefixes = ['INFO: ', 'WARN: ', 'ERROR: ']
53
54  _log_threads = []
55  _pnor_thread = None
56  _thread_running = False
57
58  _boot_finished = False
59  _boot_failed = False
60
61  ### Signal the logging thread to stop
62  ###
63  def stop_udp_logger():
64          global _thread_running
65
66          _thread_running = False
67
68  ### Spawn a thread and launch the logger
69  ###
70  def start_udp_logger(remove_old=False, timestr = ''):
71          global _thread_running
72          global _log_threads
73
74          _thread_running = True
75
76          for i in range(0, len(ports)):
77                  t = threading.Thread(target=
                        _logger_thread_entry, args=(i, remove_old,
                        timestr))
78                  t.start()
79                  _log_threads.append(t)
80
81  ### Spawn a thread to serve up a PNOR image
82  ###
83  def start_pnor_server(pnor_filename, remove_old, timestr = '')
        :
84          global _thread_running
85          global _pnor_thread
86
87          _thread_running = True
88
89          _pnor_thread = threading.Thread(target=
                _pnor_thread_entry, args=(pnor_filename, remove_old
                , timestr))
90          _pnor_thread.start()
91
92  ### Internal function
93  ### Entry point for logger thread launched by start_udp_logger
94  ###
95  def _logger_thread_entry(port_index, remove_old=False, timestr
        = ''):
96          global _thread_running, _boot_finished, _boot_failed
97
98          ## Create empty local directory if it doesn't exist
99          ##
100         if not os.path.exists(LOG_DIR):
101                 os.makedirs(LOG_DIR)
102
```

```
103
104         ## Prepend the timestamp
105         ##
106         filename = timestr + filenames[port_index]
107
108         ## Pre join the log dir and filenames
109         ##
110
111         file_path = os.path.join(LOG_DIR, filename)
112
113         if remove_old and os.path.isfile(file_path):
114                 print('Deleting old files')
115                 os.remove(file_path)
116
117         aggregate_path = os.path.join(LOG_DIR, timestr +
            aggregate_filename)
118
119         with aggregate_lock as lock:
120                 if remove_old and os.path.isfile(
                    aggregate_path):
121                         print('Deleting old files')
122                         os.remove(aggregate_path)
123
124         sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM
            )
125         sock.setblocking(True)
126         ## We need a timeout so we can kill the thread with
            _thread_running=False
127         sock.settimeout(1.0)
128
129         # Bind to the port
130         sock.bind(('', ports[port_index]))
131
132         print(f'Log server {port_index} thread started')
133
134         buffer = ''
135
136         ## Run until we are flagged to stop
137         ##
138         while _thread_running:
139
140                 try:
141                         data, sender_addr = sock.recvfrom
                            (4096)
142
143                         if len(data) > 0:
144                                 data = data.decode("utf-8", "
                                    ignore")
145
146                                 if len(data) > 500 and not
                                    ports[port_index] == 0x3f8:
147                                         print(f'Received {len(
                                            data)} bytes on
                                            port {ports[
                                            port_index]}')
148                                 else:
149                                         print(data, end='')
150
```

```
151                                     with open(file_path, 'a') as
                                           log:
152                                                log.write(data)
153
154                                     ## All threads share this file
                                           , so we need a lock
155                                     ##
156                                     with aggregate_lock as lock,
                                           open(aggregate_path, 'a')
                                           as log:
157                                                        log.write(data
                                                              )
158
159                                     buffer += data
160
161                                     if 'Enjoy!' in buffer:
162                                             _boot_finished = True
163                                             buffer = ''
164
165                                     if 'Fault callback issued' in
                                           buffer:
166                                             _boot_failed = True
167                                             buffer = ''
168
169                                     if len(buffer) > 100:
170                                             buffer = buffer[-100:]
171
172                     except Exception as e:
173                             pass
174                             #print(e)
175
176             print('Closing UDP Socket')
177             sock.close()
178
179     ### Internal function
180     ### Entry point for pnor server thread launched by
           start_pnor_server
181     ###
182     def _pnor_thread_entry(pnor_filename, remove_old, timestr = ''
           ):
183             global _thread_running
184
185             assert os.path.isfile(pnor_filename), 'ERROR: Given
                   PNOR file does not exist'
186
187             ## Create empty local directory if it doesn't exist
188             ##
189             if not os.path.exists(LOG_DIR):
190                     os.makedirs(LOG_DIR)
191
192
193             ## Prepend the timestamp
194             ## Pre join the log dir and filename
195             ##
196             file_path = os.path.join(LOG_DIR, timestr +
                   pnor_log_file)
197
198             if remove_old:
```

```
199                 print('Deleting old files')
200 
201                 if os.path.isfile(file_path):
202                         os.remove(file_path)
203 
204         ## Create the UDP server sockets
205         ##
206         sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM
            )
207         sock.setblocking(False)
208 
209         # Bind to the port
210         sock.bind(('', pnor_port))
211 
212 
213         # Storing the PNOR image in lists of BLOCK_SIZE bytes
            will allow
214         # us to avoid string slicing which is super slow
215         pnor_image = []
216 
217         with open(pnor_filename, 'rb') as p:
218                 for i in range(0, pnor_size, BLOCK_SIZE):
219                         pnor_image.append(p.read(BLOCK_SIZE))
220 
221         length = len(b''.join(pnor_image))
222 
223         assert length == pnor_size, f'PYTHON ERROR: PNOR file
            - expected {pnor_size} bytes, got {length} bytes'
224 
225         print('PNOR server thread started')
226 
227         total_length = 0
228         last_time = start_time
229 
230         ## Run until we are flagged to stop
231         ##
232         while _thread_running:
233 
234 
235                 ## Try to receive from the interface
236                 ##
237                 try:
238                         data, client_addr = sock.recvfrom
                            (5000)
239 
240                         #print(f'Received {len(data)} bytes: {
                            data}')
241 
242                         ## Use the flash read command syntax
                            CFR,[ADDR],[FLENGTH]
243                         if data.startswith(b'CFR,'):
244 
245                                 parts = data.strip().split(b',
                                    ')
246 
247                                 # Part 0 is CFR,
248                                 addr = int(parts[1], 0)
249                                 length = int(parts[2], 0)
```

```
250
251                     if addr % BLOCK_SIZE > 0 or
                           not length == BLOCK_SIZE:
252                             print('ERROR: Address
                                   or length not block
                                    aligned')
253
254                     fw = pnor_image[int(addr /
                           BLOCK_SIZE)]
255
256                     sock.sendto(fw, (client_addr
                           [0], 999))
257
258                     now = time.time()
259                     log_msg = f'{now - last_time
                           :.03f}: {parts[0]},{addr:08
                           x},{length}\n'
260                     pnor_log_q.put(log_msg)
261                     last_time = now
262
263                     if DEBUG:
264                             if addr % 0x10000 ==
                                   0:
265                                     print(log_msg,
                                            end='')
266
267
268             ## CFW,[ADDR],[FLENGTH],[DATA]
269             elif data.startswith(b'CFW,'):
270
271                     # Split the command into 4
                           parts - CFW,[ADDR],[FLENGTH
                           ],[DATA]
272                     parts = data.split(b',', 4)
273
274
275                     addr = int(parts[1], 0)
276                     length = int(parts[2], 0)
277
278                     if addr % BLOCK_SIZE > 0 or
                           not length == BLOCK_SIZE:
279                             print('ERROR: Address
                                   or length not block
                                    aligned')
280
281                     fw_data = parts[3]
282
283                     while _thread_running and len(
                           fw_data) < length:
284                             try:
285                                     data,
                                           client_addr
                                            = sock.
                                           recvfrom(
                                           length -
                                           len(fw_data
                                           ))
286                                     #print(len(
```

```
                                                        data))
287                                                 fw_data +=
                                                        data
288                                         except:
289                                                 pass
290
291                                 sock.sendto(b'OK\n', (
                                        client_addr[0], 999))
292
293                                 total_length += len(fw_data)
294
295                                 pnor_image[int(addr /
                                        BLOCK_SIZE)] = fw_data
296
297                                 # Do this last so the uC isn't
                                        waiting on us
298                                 #
299                                 now = time.time()
300                                 log_msg = f'{now - last_time
                                        :.03f}: {parts[0]},{addr:08
                                        x},{length}\n'
301                                 pnor_log_q.put(log_msg)
302                                 last_time = now
303
304                                 if DEBUG:
305                                         if addr % 0x10000 ==
                                                0:
306                                                 print(log_msg,
                                                        end='')
307
308                         else:
309                                 print(f'Received invalid
                                        command: {data}')
310
311                 except Exception as e:
312                         #print(e)
313                         pass
314
315         print('Closing UDP socket')
316         sock.close()
317
318         ## When we're finished, write the modified pnor out to
                file
319         ##
320         pnor_image = b''.join(pnor_image)
321         assert len(pnor_image) == pnor_size, f'PYTHON ERROR:
                PNOR file after modification - expected {pnor_size}
                bytes, got {len(pnor_image)}'
322
323         print(f'Final length of written data: {total_length}')
324         print('Saving modified PNOR image')
325         with open(pnor_filename, 'wb') as p:
326                 p.write(pnor_image)
327
328         ## And save off the access log
329         ##
330         print(f'Writing log to: {file_path}')
331         with open(file_path, 'a') as log:
```

```
332                 while not pnor_log_q.empty():
333                         log.write(pnor_log_q.get())
334
335 def get_local_ip(server_ip):
336         s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
337         s.connect((server_ip, 0))
338         return s.getsockname()[0]
339
340 if __name__ == "__main__":
341
342         print(socket.gethostname())
343         parser = argparse.ArgumentParser(description='
            udp_logger.py')
344         parser.add_argument('-r', '--remove_old', dest='
            remove_old', default=False, action='store_true',
            help='Delete default log files if they exist (e.g.,
             local/log_info.txt)')
345         parser.add_argument('-t', '--timestamp', dest='
            timestamp', default=False, action='store_true',
            help='Add timestamp to log filenames')
346         parser.add_argument('-p', '--pnor', dest='pnor_image',
             metavar='PNOR_FILE', type=str, help='Start a
            firmware server with the given PNOR file')
347         args = parser.parse_args()
348
349         # https://stackoverflow.com/questions/1112343/how-do-i
            -capture-sigint-in-python
350
351         import signal
352
353         def signal_handler(sig, frame):
354                 print('You pressed Ctrl+C!')
355                 stop_udp_logger()
356
357         timestr = ''
358
359         if args.timestamp:
360                 timestr = time.strftime('%Y%m%d-%H%M%S_')
361                 print(f'Timestamp: {timestr[:-1]}')
362
363         start_udp_logger(args.remove_old, timestr)
364
365         if not args.pnor_image is None:
366                 start_pnor_server(args.pnor_image, args.
                   remove_old, timestr)
367
368         signal.signal(signal.SIGINT, signal_handler)
369         print('Press Ctrl+C')
370
371         while _thread_running:
372                 time.sleep(1.0)
```

Appendix I. Experiment Script

```
1  import talos_ctrl
2  import udp_logger
3  import time
4  import socket
5  from src.heist_device import HeistDevice
6
7  experiments = open('local/random_experiments.txt', 'r').
      readlines()
8
9  #for trial in experiments:
10 while len(experiments) > 0:
11         trial = experiments[0]
12
13         prefix, pnor_filename = trial.strip().split(',')
14
15         print(f'Starting: {prefix}, {pnor_filename}')
16
17         #Step 1: Power cycle the motherboard and Disable BMC
18         print('Power cycling and disabling BMC')
19         devs = talos_ctrl.get_devs('talos_config_example.json'
              )
20         switch = talos_ctrl.get_switch(devs)
21         talos_ctrl.power_reset(switch)
22         talos_ctrl.disable_bmc(devs)
23
24         #Step 2: Start UDP logger and PNOR server
25         timestr = time.strftime('_%Y%m%d-%H%M%S_')
26         udp_logger._boot_finished = False
27         udp_logger._boot_failed = False
28         #start_udp_server with log file prefix
29         print('Starting UDP Logger')
30         udp_logger.start_udp_logger(False, prefix + timestr)
31         #start_pnor_server with desired PNOR file
32         print(f'Starting PNOR Server with local/FINAL_IMAGES/{
              pnor_filename}')
33         udp_logger.start_pnor_server('local/FINAL_IMAGES/' +
              pnor_filename, False, prefix + timestr)
34
35         #Step 3
36         interface = socket.socket(socket.AF_INET, socket.
              SOCK_STREAM)
37         interface.settimeout(1.0)
38         interface.connect(('192.168.237.101', 23))
39         interface.setblocking(False)
40         hdev = HeistDevice(interface)
41
42         #sys_reset
43         print('System Resetting and Reconnecting')
44         hdev.system_reset()
45         interface.close()
46         time.sleep(5.0)
47         interface = socket.socket(socket.AF_INET, socket.
              SOCK_STREAM)
48         interface.settimeout(1.0)
49         interface.connect(('192.168.237.101', 23))
```

```
50          interface.setblocking(False)
51          hdev = HeistDevice(interface)
52          print('Reconnect Success')
53
54          #TO-DO: Check if uC started up correctly
55
56          #Send options to uC
57          udp_addr = udp_logger.get_local_ip('192.168.237.101')
58          resp, success = hdev.cmd_logging_set(False, True,
               udp_addr)
59          if not success:
60                  print('uC Failed to Start!')
61                  break
62
63          #
64          resp, success = hdev.cmd_pnor_set_options(udp_addr)
65          if not success:
66                  print('uC Failed to Start!')
67                  break
68
69          #chassis_on
70          print('Chassis On')
71          hdev.chassis_on()
72          time.sleep(2.0)
73
74          #init
75          print('CMD Init All')
76          hdev.cmd_init_all()
77
78          #boot
79          print('Boot Host')
80          hdev.boot_host()
81
82          start_time = time.time()
83          # if 'Fault callback issued' then run failed
84          #wait for udp_logger to finish
85          print('Waiting for UDP Logger to finish')
86          while (not udp_logger._boot_finished) and (not
               udp_logger._boot_failed):
87                  time.sleep(1.0)
88                  #Check if time greater than 5 mins
89                  if time.time() - start_time > 300:
90                          print('UDP Logger Took longer than 5
                               minutes... BREAK!')
91                          break
92                          #Log the trial and restart run
93
94          interface.close()
95
96          time.sleep(2.0)
97
98          print('Stopping UDP logger')
99          udp_logger.stop_udp_logger()
100
101         udp_logger._pnor_thread.join()
102
103         for t in udp_logger._log_threads:
104                 t.join()
```

```
105
106         print('UDP logger threads all joined')
107
108         #copy/move log to results folder
109         if udp_logger._boot_finished:
110                 experiments.remove(trial)
111
112                 with open('local/random_experiments.txt', 'w')
                     as outfile:
113                         outfile.write(''.join(experiments))
114
115                 with open('local/completed.txt', 'a') as
                    outfile:
116                         outfile.write(trial)
117
118         else:
119                 with open('local/failed.txt', 'a') as outfile:
120                         outfile.write(trial)
```

Appendix J. HBBL XOR Decryption Firmware

```
1 uint8_t* buffer = (uint8_t*) ((l_hbbEcc) ?
2     (HBB_ECC_WORKING_ADDR | IGNORE_HRMOR_MASK) :
3     (HBB_WORKING_ADDR | IGNORE_HRMOR_MASK));
4
5 for (uint32_t i = 0; i < workingLength; i++) {
6     buffer[i] ^= 0x55;
7 }
```

Appendix K. HBBL XOR Script

```
1  infile = 'A:/Calvin Muramoto/Documents/Talos PNOR Images/
      PNOR_XOR_HBBL/preXOR_hbbl_stock_talos.pnor'
2  outfile = 'A:/Calvin Muramoto/Documents/Talos PNOR Images/
      PNOR_XOR_HBBL/XOR_hbbl_stock_talos.pnor'
3
4  with open(infile, 'rb') as input:
5      image = input.read()
6
7  print(f'Original file size: {len(image)}')
8
9  part_start = 0x205000
10 part_end = part_start + 0x100000
11
12 pre = image[: part_start]
13 print(f'Pre-Payload: {len(pre)}')
14 payload = image[part_start: part_end]
15 print(f'Payload size: {len(payload)}')
16 post = image[part_end: ]
17 print(f'Post-size: {len(post)}')
18
19 L = list(payload)
20 print(range(len(L)))
21 for i in range(len(L)):
22     L[i] = L[i] ^ 0x55
23
24 final_image = pre + bytes(L) + post
25
26 print(f'End Addr: {part_end:08x}')
27 print(f'New file size: {len(image)}')
28
29 with open(outfile, 'wb') as write_file:
30     write_file.write(final_image)
```

Appendix L. HBBL Speck Decryption Firmware

```
1  uint8_t* buffer = (uint8_t*) ((l_hbbEcc) ?
2      (HBB_ECC_WORKING_ADDR | IGNORE_HRMOR_MASK) :
3      (HBB_WORKING_ADDR | IGNORE_HRMOR_MASK));
4
5  for ( uint64_t i = 0; i < workingLength / 8; i++ ) {
6      /*********************************************************
7      *    Prep Cipher text
8      *********************************************************/
9      int ct_numbytes = 8;
10     uint8_t ct_bytes[ct_numbytes];
11     for(uint32_t j = 0; j < 8; j++) ct_bytes[j]=buffer[(8 * i)
           + j];
12
13     // Convert byte array to 32bit words
14     int ct_length = ct_numbytes / 8;
15     uint32_t ct_words[ct_length];
16     BytesToWords32(ct_bytes, ct_words, ct_numbytes);
17
18     /*********************************************************
19     *    Prep Key
20     *********************************************************/
21     // 12 Byte = 96 Bit key
22     uint8_t key[] = {0x00,0x01,0x02,0x03,0x08,0x09,0x0a,0x0b,0
          x10,0x11,0x12,0x13};
23     int key_numbytes = sizeof(key);
24
25     uint32_t key_words[key_numbytes];
26     BytesToWords32(key, key_words, key_numbytes);
27
28     /*********************************************************
29     *    Key Schedule
30     *********************************************************/
31     uint32_t rk[26];
32     Speck6496KeySchedule(key_words, rk);
33
34     /*********************************************************
35     *    Decrypt
36     *********************************************************/
37     uint32_t pt_words[ct_length];
38     Speck6496Decrypt(pt_words, ct_words, rk);
39
40     uint8_t pt_bytes[ct_numbytes];
41     Words32ToBytes(pt_words, pt_bytes, 2);
42     for(uint32_t j = 0; j < 8; j++) buffer[(8 * i) + j]=
          pt_bytes[j];
43 }
```

Appendix M. HBBL AES Compilation Fail

```
1 //============= Generate PNOR Image Settings ===========//
2 PNOR Layout = /home/murtly50/talos-op-build/output/build/
     openpower-pnor-06cd438e98b89d1b28b3cbfcdf6a880e9c02621f/
     p9Layouts/defaultPnorLayout_64.xml
3 Emit ECC-less versions of output files, when possible = No
4 Test Mode = No
5 Secureboot = Disabled
6 Sign Mode = NA
7 Key Transition Mode = NA
8 Lab security override (valid for SBE partition only) = Yes
9 //=====================================================//
10 Loading bin files...
11 TRACE: manipulateImages
12 HBBL raw size (/home/murtly50/talos-op-build/output/host/
     powerpc64le-buildroot-linux-gnu/sysroot/
     openpower_pnor_scratch//HBBL.staged) (no padding/ecc) =
     21040/20480
13 HBBL cannot fit HW Keys' Hash (64 bytes) at the end without
     overwriting real data at /home/murtly50/talos-op-build/
     output/host/powerpc64le-buildroot-linux-gnu/sysroot/
     hostboot_build_images//genPnorImages.pl line 632.
```

Bibliography

1. D. Heller and N. Sastry, "OpenPower secure and trusted boot, Part 2: Protecting system firmware with OpenPOWER secure boot," 2019. [Online]. Available: https://developer.ibm.com/articles/protect-system-firmware-openpower/

2. A. Geissler, "Hostboot POWER Systems Initialization Firmware," 2015. [Online]. Available: https://github.com/open-power/docs/blob/master/hostboot/HostBoot_PG.md

3. "Image Encryption," 2019. [Online]. Available: https://www.digi.com/resources/documentation/digidocs/embedded/dey/2.6/cc8x/yocto-trustfence_t_image-encryption.html

4. "AN 709: HPS SoC Boot Guide - Cyclone V SoC Development Kit." [Online]. Available: https://www.intel.com/content/www/us/en/docs/programmable/683265/current/boot-flow.html

5. "Secure Initial Program Load (IPL) process," 2020. [Online]. Available: https://www.ibm.com/docs/en/power9/9009-42A?topic=9009-42A/p9ia9/p9ia9_secure_ipl_proc_concept.htm

6. "AVR231: AES Bootloader," 2017. [Online]. Available: http://ww1.microchip.com/downloads/en/AppNotes/00002462A.pdf

7. "OpenPOWER Firmware," 2021. [Online]. Available: https://wiki.raptorcs.com/wiki/OpenPOWER_Firmware

8. "Default Pnor Layout," Tech. Rep. [Online]. Available: https://git.raptorcs.com/git/pnor/tree/p9Layouts/defaultPnorLayout_64.xml [Accessed: 2021-09-20]

9. A. Jeffery, "Hacking Hostboot," 2018. [Online]. Available: https://amboar.github.io/notes/2018/08/17/hacking-hostboot.html [Accessed: 2021-09-20]

10. ——, "General Architecture of Hostboot," 2018. [Online]. Available: https://amboar.github.io/notes/2018/08/19/hostboot-architecture.html [Accessed: 2021-09-20]

11. R. Beaulieu, D. Shors, J. Smith, S. Treatman-Clark, B. Weeks, and L. Wingers, "Simon and Speck: Block Ciphers for the Internet of Things," *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15*, pp. 1–6, July 2015. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2744769.2747946

12. I. T. L. Computer Security Division, "Aes development - cryptographic standards and guidelines: Csrc," Aug 2021. [Online]. Available: https://csrc.nist.gov/projects/cryptographic-standards-and-guidelines/archived-crypto-projects/aes-development

13. M. E. Smid, "Development of the advanced encryption standard," *Journal of Research of the National Institute of Standards and Technology*, vol. 126, 2021.

14. "AN0060: Bootloader with AES Encryption," 2016. [Online]. Available: https://www.silabs.com/documents/public/application-notes/an0060-bootloader-with-aes-encryption.pdf

15. A. Fournaris, "Using Hardware Means to secure Critical Infrastructure Devices." [Online]. Available: https://www.cipsec.eu/content/using-hardware-means-secure-critical-infrastructure-devices

16. R. Nuqui and A. Phadke, "Phasor measurement unit placement techniques for complete and incomplete observability," *IEEE Transactions on Power Delivery*, vol. 20, no. 4, pp. 2381–2388, 2005.

17. "OpenPower-Firmware: SBE Questions," 2019. [Online]. Available: https://lists.ozlabs.org/pipermail/openpower-firmware/2019-July/000337.html

18. S. Smith, "OpenPower-Firmware SBE questions," 2019. [Online]. Available: https://lists.ozlabs.org/pipermail/openpower-firmware/2019-July/000337.html [Accessed: 2021-12-05]

19. "Red Hat Enterprise Linux 4: Reference Guide. Chapter 1. Boot Process, Init, and Shutdown." [Online]. Available: https://web.mit.edu/rhel-doc/4/RH-DOCS/rhel-rg-en-4/s1-boot-init-shutdown-process.html [Accessed: 2021-01-11]

20. "AMD Family 17h in coreboot." [Online]. Available: https://doc.coreboot.org/soc/amd/family17h.html?highlight=amd

21. "x86 System Initialization." [Online]. Available: https://wiki.osdev.org/System_Initialization_(x86) [Accessed: 2021-01-11]

22. "UEFI Applications in Detail." [Online]. Available: https://wiki.osdev.org/UEFI#UEFI_applications_in_detail [Accessed: 2021-01-11]

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

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

1. REPORT DATE *(DD–MM–YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From — To)*
24–03–2022	Master's Thesis	Sept 2020 — Mar 2022

4. TITLE AND SUBTITLE

Evaluating the use of Boot Image Encryption on Talos II Architecture

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Muramoto, Calvin M., 2d Lt, USAF

5d. PROJECT NUMBER

21G195

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-22-M-049

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

Sensitive devices operating in unprotected environments are vulnerable to hardware attacks like reverse engineering and side channel analysis. This represents a security concern because the root of trust can be invalidated through boot firmware manipulation. For example, boot data is rarely encrypted and typically travels across an accessible bus like the LPC bus, allowing data to be easily intercepted and possibly manipulated during system startup. The flash chip storing the boot data can also be removed from these devices and examined to reveal detailed boot information. This paper details an implementation of encrypting a section of the boot image and decrypting it during the IPL of the Talos II. During power-on, the encrypted image travels across the LPC bus into the POWER9 Level3 cache and is decrypted in the processor. This proves that it is possible to prevent adversaries from interfering with the IPL flow or obtaining details on firmware from the flash chip. The boot image encryption method is implemented with multiple levels of encryption and an evaluation of their efficiency is conducted to determine the performance impact for each algorithm.

15. SUBJECT TERMS

Secure Boot, Hardware Security, Firmware Encryption, Boot Security

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Dr. Scott Graham, AFIT/ENG
					19b. TELEPHONE NUMBER *(include area code)*
U	U	U	UU	101	(937) 255-6565 x4581; scott.graham@afit.edu