Improving Hardware Security on Talos II Architecture Through Boot Image Encryption

Calvin Muramoto, Stephen Dunlap and Scott Graham
Air Force Institute of Technology, USA
Calvin.Muramoto@afit.edu
stephen.dunlap.ctr@afit.edu
Scott.Graham@afit.edu

Abstract: The OpenPOWER Foundation is an organization that promotes open-source high-performance hardware like the POWER9. OpenBMC is an OpenPower project that strives to produce an open-source firmware stack for Baseboard Management Controllers (BMCs). If hardware falls into the hands of competitors or bad actors, reverse engineering methods can be used to leak or manipulate sensitive information from the boot sequence. This represents a security concern because the root of trust can be invalidated. For example, since the Initial Program Load (IPL) data is frequently not encrypted and is sent over the Low Pin Count (LPC) bus, it is possible to intercept and conduct man-in-the-middle attacks to modify the boot process. The boot image flash chip could also be removed from the Talos II motherboard and examined by competing server architecture manufacturers 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 image chip. This paper demonstrates a method to encrypt sections of the boot image by encrypting a section of the image before flashing it onto the Talos II. The encrypted image will be decrypted during the boot sequence in the Level 3 cache of the POWER9, proving that it is possible to prevent adversaries from interfering with the IPL flow or obtaining details on firmware from the flash chip. This paper presents a novel method to improve the security of the boot image on Talos II architecture by encrypting the boot firmware image and decrypting it during the boot process. The proof of concept was executed on a Raptor Engineering Talos II system running a POWER9 processor with OpenBMC firmware on the ASPEED AST2500 BMC. This research claims that this unique method increases boot time security through firmware without altering hardware.

Keywords: Secure Boot, Hardware Security, Firmware Encryption, PNOR, Boot Image, Hostboot

1. Introduction

As the complexity of cyberattacks increase and evolve, the demand for increased security in computer systems also rises. Hardware attacks have become a concern in recent years because physical device hardware is not normally designed to be resistant to the increasing number of attack vectors. To counter this threat, trusted computing has become a focus for computing system development with security enhancements, like secure boot (Mitchell, 2005). Although secure boot can prevent adversaries from loading malicious firmware on a system, no known system encrypts the existing boot firmware.

Protecting boot firmware data from being leaked allows developers to conceal code that performs innovative functionality to the system. Hiding this low-level firmware also increases the difficulty of discovering zero-day attacks or vulnerabilities in the code. Tamper detection methods should be located in source code in the boot sequence to prevent intruders before the operating systems starts. Leaking the source code on intrusion prevention systems would allow bad actors to avoid the tamper checks and gain access to the system. Boot firmware is often left unencrypted in memory leaving the system open to hardware attacks like bus traffic recording or chip substitution attacks. Encryption adds an additional layer of protection from adversaries looking to reverse engineer firmware.

Although this proof of concept is developed specifically for the Talos II, it should be possible to implement boot firmware encryption on common systems with Intel and AMD processors. As long as the system uses a similar boot flow structure as OpenPOWER, encrypting sections of the boot firmware should be similar.

This paper demonstrates how encrypting of a section of the boot image is possible on an OpenPOWER workstation. As a proof-of-concept, this effort used a simple exclusive OR function as a proxy for encryption. Success will be measured by implementing the simple encryption function, flashing the modified image onto the Talos II, and verifying that the system can successfully boot consistently. This would protect the boot data from outsiders by only updating the firmware.

This paper is split into 6 sections to cover the research effort. Section 2 introduces the background information required to understand the nature of boot image encryption. In section 3, the methodology for this proof of concept is covered along with a discussion on feasibility. The experimental analysis of the methodology along with definition of success is contained in Section 4. The impact of this research is covered in Section 5 and the conclusion and discussion on future work is covered in section 6.

2. Background

2.1 Talos II Architecture

Talos II is a secure workstation designed by Raptor Computing Systems. It makes use of dual POWER9 CPUs, PCIE 4.0, and DDR 4.0. Most importantly, the Talos II also supports OpenBMC firmware, allowing developers to utilize open-source Baseboard Management Controller (BMC) firmware. This allows us to modify the booting process through firmware in this study. The BMC is a functional system processor that can monitor and control the system that it is implemented on.

Initial Program Load (IPL) refers to the steps that OpenPOWER systems take from power on to the start of the hypervisor. During IPL, the Talos II utilizes several interfaces to transfer information between the PNOR flash chip, Serial electrically erasable programmable read-only memory (SEEPROM), Self-boot engine (SBE), and POWER9. The most important interfaces are the Low pin count (LPC) bus, pervasive interconnect bus, and serial communication. The LPC bus loads the PNOR flash image to the POWER9 and can perform memory, I/O, direct memory access, and bus master cycles. A trusted platform module on the Talos II uses the LPC to communicate with the SBE and POWER9. The pervasive interconnect bus connects and facilitates transactions between multiple pervasive interconnect bus master and slave units inside the POWER9 chip. It is mainly used by the SBE at the very beginning of the IPL for memory I/O. Finally, serial communication is a mechanism in the POWER9 chip pervasive logic that gives access to pervasive registers when the clocks are running. It is a specialized interface into the chip pervasive logic that allows specific latches to be updated while functionally in use.

An important consideration taken into this research was the size available on the PNOR flash chip, SEEPROM, and POWER9 Level 3 (L3) cache. Adding bloated code or having excessive memory usage during the IPL could cause a buffer overflow and crash the boot sequence. The PNOR flash chip is 4MB, the POWER9 SEEPROM is 64 KB, and POWER9 L3 cache is 10 MB. When allocating space for temporary data in the XOR function, we had to ensure that the space did not exceed the L3 cache size. Although the L3 cache has a memory space limitation, the cache provides fast data transfer speeds to the POWER9 CPU cores which helps reduce memory I/O instruction execution time. We also had to ensure that the compiled XOR code did not overflow the memory allocated for the Hostboot firmware on the PNOR.

2.2 PNOR Image Structure

The PNOR Image contains all the firmware needed to boot the Talos II. It is split into logical partitions, with each section containing a firmware module. Figure 1 shows a general overview of the architecture of the boot image used in OpenPOWER systems. The Talos II version of the PNOR is 4 MB and contains 31 sections with specific functions like Secure Boot, DIMM SPD Cache, and Module VPD Cache (Raptor CS, 2019). The Table of Contents (TOC) is located at the start and end of the image and contains important data like the name of the partition, physical offset, and physical size. It is queried often during IPL to inform the firmware of the information needed to load the following partitions. Before loading each section in the IPL, the SBE or POWER9 will look up the address and size of each partition to load into the L3 cache or DRAM. The sections relevant to this research are the Hostboot Base (HBB), Hostboot Extended (HBI), and OPAL sections.

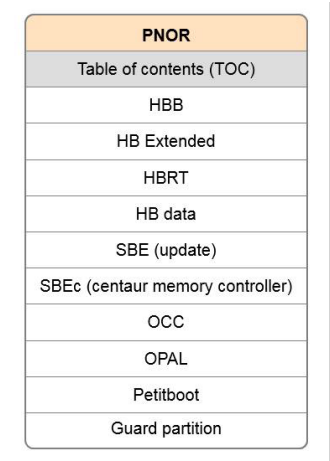

Figure 1: PNOR structure (IBM, 2019)

2.3 Initial Program Flow

IPL is a term that covers the boot process from power on to running the hypervisor. The Talos II follows the boot flow specified in Fig 2. Once the system is powered on, the OTPROM is loaded from the CPU die and executed on the SBE. The SBE is a specialized small auxiliary microprocessor that initializes the processor chip (IBM, 2019). The OTPROM is a section of SBE firmware that is permanently loaded through eFuses on the POWER9 silicon and loads the SEEPROM into the SBE pervasive interconnect memory (Raptor CS, 2021). The pervasive interconnect bus memory is a 64 KB SRAM attached to the pervasive interconnect bus and used by the SBE as its memory.

The SEEPROM firmware on the SBE core initializes a designated CPU core, allows access to the PNOR flash memory, and then loads the HBBL from SEEPROM. SEEPROM is an EEPROM memory device that can only be read but can also be reprogrammed through the IPL process (Geissler, 2015). The stages that follow are all included in the PNOR and are covered in the following sections.

The PNOR flash chip is a NOR memory device where all firmware is stored for the latter portion of the IPL. It contains Hostboot, OPAL, the OCC, and Bootloader firmware. It is connected to the master or alternate master processor through the SPI bus. The image loaded into the PNOR consists of several sections of data that correspond to stages in the initial program load. This paper focuses on the Hostboot Extended (HBI) image and the payload.

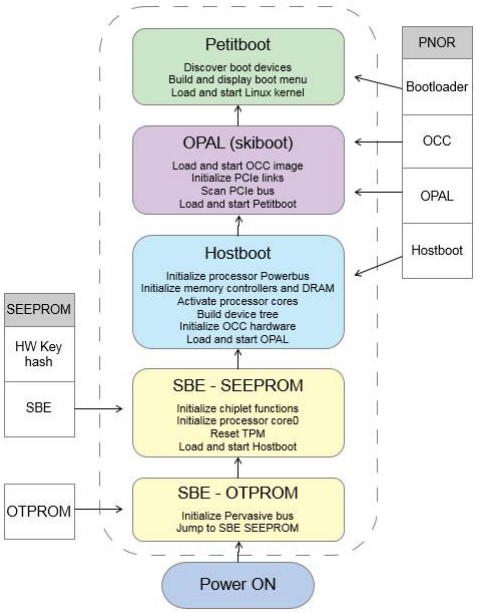

Figure 2: IPL Flow (IBM, 2019)

2.4 Hostboot

The Hostboot stage of the boot process is extremely important as it initializes a series of capabilities that are needed for later boot stages. It is loaded and started in ISTEP5, and covers ISTEPs 6 through 21. Throughout the course of these ISTEPs, the processor bus, memory controllers, DRAM, and PSI/Node are all initialized. The most important partitions are the HBBL, HBB, and HBI. Hostboot has its own cache-contained operating system that separates a user and kernel space along with task management for HBI (Jeffery, 2018). Since the memory requirements are larger than the 10 MB available on the L3 cache, a virtual memory and filesystem layer is used. This requires demand-paging to bring code and data in from the PNOR when needed (Jeffery, 2018).

The HBBL firmware is stored and run from SEEPROM but is also part of the PNOR image. This is because the HBBL on SEEPROM can be updated through a successful boot process. HBBL handles locating the HBB partition, ECC verification, loading into the L3 cache, and execution on the first core. This firmware is the first code that will run on the main CPU cores after power on. Although the HBBL source code is located in the Hostboot repository, it is stored on the SEEPROM SBE image where the secure boot root of trust hash is located. This is because the HBBL is responsible for cryptographically verifying the integrity of the HBB and HBI when trusted boot is enabled. The switch from HBBL to HBB is handled by the *bl_start.S* code.

HBB contains the hostboot kernel and handles key low-level modules like serial communication, LPC, PNOR access, and Virtual File System (VFS). HBB takes care of the required services to read and write to the PNOR. It also functions as the foundation for the rest of the HBI startup.

Execution of the HBI is done through the VFS. The HBI phase of the boot process consists of a list of ISTEPS and services that need to be executed. The *initservice* module runs through the list of services and sends a start message for each service. The Virtual File System Resource Provider (VFSRP) utilizes a watcher that handles each message by requesting the relevant memory addresses and responding. HBI is also responsible for chain-loading the subsequent firmware payload in ISTEP 20 and 21. When complete, Hostboot passes control of the CPU to the payload firmware and Skiboot is run.

2.5 Payload

On all OpenPOWER systems, the firmware payload is Skiboot. Skiboot handles the initialization of PCIE, device trees, real-time clock, NVlink, and sensors. These sensors are required when Skiboot loads the OCC firmware and begins its execution. The OCC manages CPU temperatures, power measurement, and other chassis-specific data. Finally, the OpenPOWER Abstraction Layer (OPAL) is started for OS runtime services. OPAL resides in RAM after OS boot. Runtime API implemented by firmware is called into by the host operating system to invoke platform-specific functionality. Skiboot then chain loads Skiroot and Petitboot, which act as bootloaders for Linux.

3. Methodology

The methodology for this proof of concept is complex due to the multiple stages required to make firmware changes. Before the PNOR is flashed onto the Talos II, a section of memory needs to be identified for encryption. The decryption function is 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 is then compiled and the address space identified earlier is encrypted. The PNOR with the encrypted section and decryption function is flashed onto the Talos II. Multiple successful boot sequences would be proof that the implementation was successful.

3.1 Proof of Concept Development Discussion

After analyzing the IPL firmware, several points in the code were identified where the decryption code could execute successfully. The first possibility was the Hostboot Bootloader (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, so machine code is used to communicate with the LPC to transfer data. This also means that the console output (print out and log commands) would not be accessible, making it difficult to verify the correct address and size of the partition for encryption.

Another problem was the risk in 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 POWER9 could get “bricked”.

The second location in code possible was for the HBI. Since HBB handles the modules for starting the HBI, we were able to narrow down the location to the *vfsrp.C*, which is the virtual file system resource provider. 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 POWER9 to load in each task and initialize them. Each task sends a MSG_MM_RP_READ, which executes a *memcpy()* which is used to load the relevant data into the local execution space. The problem with using 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.

Finally, the last point in code is located at 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 is only run one time to load the payload from the PNOR into local memory space. The only possible problem with working in this code would be 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.

The plan was to allocate a space to store the encrypted payload, as shown on line 270 in Figure 3, and copy the data from the PNOR to the local memory space. The encrypted payload was loaded in 4 KB chunks, as shown in lines 277 to 284 in Figure 4. After all the encrypted data was stored in the *temp_buf*, the data was XORed in place on lines 289 to 292 in Fig 4. The XORed data was then passed into the XZ decompression function. The *5isplay* 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.

```
250         // Display a banner on the status of PNOR compression
251         CONSOLE::displayf(NULL, "call_host_load_payload.C: PNOR is XZ compressed\n");
252         CONSOLE::flush();
253
254         struct xz_buf b;
255         struct xz_dec *s;
256         enum xz_ret ret;
257
258         xz_crc32_init();
259         s = xz_dec_init(XZ_SINGLE, 0);
260         if(s == NULL)
261         {
262             TRACFCOMP(ISTEPS_TRACE::g_trac_isteps_trace,ERR_MRK
263                       "load_pnor_section: XZ Embedded Initialization failed");
264             return err;
265         }
266
267         CONSOLE::displayf(NULL, "call_host_load_payload.C: Starting malloc\n");
268         CONSOLE::flush();
269
270         uint8_t* temp_buf = (uint8_t*) malloc(originalPayloadSize);
271
272         if ( ! temp_buf) {
273             CONSOLE::displayf(NULL, "call_host_load_payload.C: Malloc failed\n");
274             CONSOLE::flush();
275         }
```

Figure 1*:* XOR code showing the XZ compression code and the allocated space for decryption.

```
277         const uint32_t BLOCK_SIZE = 4096;
278         for ( uint32_t i = 0; i < originalPayloadSize; i += BLOCK_SIZE )
279         {
280             memcpy( reinterpret_cast<void*>(
281                       reinterpret_cast<uint64_t>(temp_buf) + i ),
282                     reinterpret_cast<void*>( pnorSectionInfo.vaddr + i ),
283                     std::min( originalPayloadSize - i, BLOCK_SIZE ) );
284         }
285
286         CONSOLE::displayf(NULL, "call_host_load_payload.C: Done copying to local buffer\n");
287         CONSOLE::flush();
288
289         for ( uint64_t i = 0; i < originalPayloadSize; i++ )
290         {
291             temp_buf[i] ^= 0x55;
292         }
293
294         for (uint32_t i = 0; i < 16; i++){
295             CONSOLE::displayf(NULL, "Header %d = %02x\n", i, temp_buf[i]);
296             CONSOLE::flush();
297         }
298         CONSOLE::displayf(NULL, "call_host_load_payload.C: Done XORing\n");
299         CONSOLE::flush();
```

Figure 2*:* XOR code showing the memcpy() into the allocated space to be XORed with the key.

3.2 Assumptions and Limitations

Since this research was a proof of concept to show that the PNOR image could be encrypted, a simple XOR was used to 'encrypt' the boot firmware. This is a limitation in this proof of concept because a simple XOR is not secure. A key of "0x55" was used when XORing the payload because it should flip every single bit in the section, making it easy to verify in a hex editor. For the purposes of this demonstration, a simple XOR is representative of an encryption scheme.

This implementation is also only executed on the payload of the boot sequence, meaning that only the payload would be secure. The decryption code in Hostboot is also in plaintext, along with the key used to XOR the partition. These limitations will be addressed in future research.

This implementation requires a Talos II system running a single POWER9 without secure boot. It also requires the PNOR image to be compiled on a separate system and XORed before being flashed onto the system.

Finally, the key of "0x55" was stored unprotected in the Hostboot firmware; mechanisms to properly secure such a key will be explored in future research.

4. Experimental Analysis

There was a problem when initially working 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, we planned to allocate a space in memory to manually store the PNOR image, decrypt the compressed image, and pass the decrypted data into the decompression function.

A few attempts were made to boot the Talos II with an encrypted payload PNOR successfully. Initially, the XOR decryption function was used directly on the data pointed to by the *pnorSectionInfo.vaddr*. We quickly discovered that the virtual address specified for each PNOR section is memory mapped to an address space on the LPC bus so the system did not boot successfully. This is a problem because we were attempting to XOR in place, and this would attempt to write over the PNOR section over the LPC bus. We resorted to an implementation with a higher cost to memory by allocating a 1 MB block that would copy memory from PNOR and XOR the data in place. This memory space was then passed into the decompression function and led to 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 OPAL and the following boot process. Figure 5 shows the OBMC console client logs where the print statements in Figures 3 and 4 are shown. The figure also shows ISTEP 21.3 on line 98 and the following log on line 99 show that OPAL Skiboot was able to start, meaning that the XOR code ran successfully. After running the boot sequence several times and cycling the power, we verified that the Talos II was able to boot Linux without issue.

```
33.84818|ISTEP 20. 1 - host_load_payload
33.87024|call_host_load_payload.C: Starting load_pnor_section()
33.87144|call_host_load_payload.C: PNOR is XZ compressed
33.87391|call_host_load_payload.C: Starting malloc
34.53205|call_host_load_payload.C: Done copying to local buffer
34.53735|Header 0 = fd
34.53027|Header 1 = 37
34.53836|Header 2 = 7a
34.53878|Header 3 = 58
34.53926|Header 4 = 5a
34.53970|Header 5 = 00
34.54022|Header 6 = 00
34.54065|Header 7 = 01
34.54111|Header 8 = 69
34.54157|Header 9 = 22
34.54202|Header 10 = de
34.54250|Header 11 = 36
34.54295|Header 12 = 02
34.54348|Header 13 = 00
34.54393|Header 14 = 21
34.54442|Header 15 = 01
34.54484|call_host_load_payload.C: Done XORing
34.54563|Running Decompression
34.66993|Freeing buffer
34.67050|call_host_load_payload.C: Ending load_pnor_section()
34.67252|ISTEP 20. 2 - host_load_hdat
35.55996|ISTEP 21. 1 - host_runtime_setup
41.29561|htmgt|OCCs are now running in ACTIVE state
46.02538|ISTEP 21. 2 - host_verify_hdat
46.05073|ISTEP 21. 3 - host_start_payload
[   46.323404859,5] OPAL skiboot-9858186 starting...
```

Figure 3: Boot log messages and successful OPAL start message

5. Research Impact

The successful implementation of the PNOR encryption methodology in this research shows 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 POWER9 processor and stored in the L3 cache, making it extremely difficult to reveal the plaintext firmware. This also proves that it is possible to manipulate the boot data if the allocated space does not overflow memory. The ability to control boot data during the IPL shows that the capabilities of the POWER9 are only limited by the memory space available during the early stages of the boot process.

This approach is expected to be applicable on server architectures utilizing other processors, such as Intel and AMD. It should be possible if the boot structure follows a similar model to OpenPOWER's format. This would allow every consumer computer system to support encrypted boot firmware which would help protect against hardware attacks.The boot firmware encryption technique could also be applied to BIOS and firmware updates, permitting new boot firmware to be securely flashed onto a computer system without the risk of source code being leaked.

6. Conclusions and Future Work

Since this research was a proof of concept to show that the PNOR image could be encrypted, a simple XOR was used to 'encrypt' the boot firmware. The implementation of the XOR can be changed to a secure encryption like SPECK, AES or DES in the future. SPECK is a lightweight encryption protocol developed by NSA optimized for firmware implementation (Beaulieu, 2015).

This research also only focused on implementing the boot image encryption on the OPAL section of the PNOR. In the future, this could be expanded to multiple partitions based on how the firmware stage is loaded into memory. The multi-section encryption could be accomplished by programming the decryption method into each partition and encrypting the respective portions.

Future research will be to incorporate the PNOR encryption scheme with Secure Boot enabled. IBM's Secure boot allows for the verification of Hostboot and stores the source code on the SBE SEEPROM with the goal of ensuring that only legitimate firmware is loaded into the flash chip (Heller, 2019). This should be relatively simple as secure boot uses a chain of trust system where each firmware component of the boot process verifies the next section before being allowed to run.

Finally, key management would need to be solved as the current key exists unprotected in the HBI firmware. In future research, the key should be saved in SEEPROM with the verification and bootloader signatures.

Disclaimer: The views expressed in this paper are those of the authors, and do not reflect the official policy or position of the United States Air Force, Department of Defense, or the U.S. Government. This document has been approved for public release; distribution unlimited, case # 88ABW-2021-0886.

References

Beaulieu, R., Shors, D. and Smith, J. (2015) "Simon and Speck: Block Ciphers for the Internet of Things". National Security Agency. https://csrc.nist.gov/csrc/media/events/lightweight-cryptography-workshop-2015/documents/papers/session1-shors-paper.pdf.

Geissler, A. (2015) "Hostboot POWER Systems Initialization Firmware". https://github.com/open-power/hostboot. Accessed 20 Sep 21.

Jeffery, A. (2018) "General Architecture of Hostboot". https://amboar.github.io/notes/2018/08/19/hostboot-architecture.html. Accessed 20 Sep 21.

Jeffery, A. (2018) "Hacking Hostboot". https://amboar.github.io/notes/2018/08/17/hacking-hostboot.html. Accessed 20 Sep 21.

Heller, D and Sastry, N. (2019) "OpenPower secure and trusted boot, Part 2: Protecting system firmware with OpenPOWER secure boot". https://developer.ibm.com/articles/protect-system-firmware-openpower/. Accessed 20 Sep 21.

IBM (2020) "Secure Initial Program Load (IPL) process". https://www.ibm.com/docs/en/power9/9009-42A?topic=9009-42A/p9ia9/p9ia9_secure_ipl_proc_concept.htm. Accessed 20 Sep 21.

Microchip (2017) "AVR231: AES Bootloader". http://ww1.microchip.com/downloads/en/AppNotes/00002462A.pdf. Accessed 5 Dec 21.

Mitchell, C. (2005) "Trusted Computing". The Institution of Engineering and Technology, London, United Kingdom. p. 1-6

OpenPOWER (2021) "Hostboot". https://github.com/open-power/hostboot. Accessed 20 Sep 21.

Raptor Computing Systems (2019) "index : pnor". https://git.raptorcs.com/git/pnor/tree/p9Layouts/defaultPnorLayout_64.xml. Accessed 20 Sep 21.

Raptor Computing Systems Wiki (2021) "OpenPOWER Firmware". https://wiki.raptorcs.com/wiki/OpenPOWER_Firmware. Accessed 20 Sep 21.