Secure Boot Solution for RISC-V

Nitya Joshi
Electrical and Computer Engineering Dept.
University of Wisconsin-Madison
Madison, Wisconsin
njoshi26@wisc.edu

Pragna Pulakanti
Electrical and Computer Engineering Dept.
University of Wisconsin-Madison
Madison, Wisconsin
ppulakanti@wisc.edu

Ujwal Ravichandra
Electrical and Computer Engineering Dept.
University of Wisconsin-Madison
Madison, Wisconsin
ravichandra@wisc.edu

Rakshith Macha Billava
Electrical and Computer Engineering Dept.
University of Wisconsin-Madison
Madison, Wisconsin
machabillava@wisc.edu

***Abstract*— With the ever-increasing pervasiveness of embedded systems, comes the question of how secure these devices are. Secure boot is one of the most basic safeguards to ensure against the installation of a malicious bootloader onto the device. We aim to implement a secure boot on a RISC-V based microcontroller architecture, to establish a chain of trust. For our implementation, we will be using the PULPissimo framework. PULPissimo is an open-source framework that is widely used for several academic projects. It is also on the advent of transitioning towards a broad market product. Most products developed, focus on the application rather than the security of the product. We hope to provide a pick-and-place secure boot to ensure that the code running on the processor is verified.**

Keywords—Secure boot, Hardware security, RISC-V Architecture, PULPissimo

I. Introduction

PULPissimo is the microcontroller architecture of the more recent PULP chips, part of the ongoing "PULP platform". With energy efficiency at the core of its design, it can lend itself readily to various kinds of embedded systems applications. With the omnipresent nature of embedded devices, ensuring device security is of even higher priority as malicious attackers might easily gain physical access to tamper with devices. Despite of the rapid growth of device number and market size, security has been overlooked for embedded systems due to the lagging security standards, inadequate investment in security development as well as the lack of security awareness. To improve its security robustness, we provide a basic secure boot setup that validates the flash before launching application. We validate the said implementation. that can aid further development of security features for PULPissimo.

II. Background Study

A. SecureBoot

Secure boot is a mechanism that establishes a Chain of Trust (CoT) on all system boot images. Secure boot relies on the public key cryptography to verify image signatures before their execution. A pair of public and private key is generated for image signing and verification. The private key is used to sign an image offline while the public key is used to verify the image signature before one image is executed. The whole secure boot process usually involves several images. The image of the former boot stage verifies the image of the next boot stage, which in turn forms a verification chain, known as the Chain of Trust. During the secure boot, a single signature verification failure can terminate the whole system booting process.

B. PULPissimo: Open source RISCV Architecture

PULPissimo is the microcontroller architecture of the more recent PULP chips, part of the ongoing "PULP platform" collaboration between ETH Zurich and the University of Bologna - started in 2013.

PULPissimo is a single-core platform. However, it represents a significant step ahead in terms of completeness and complexity - the PULPissimo system is used as the main System-on-Chip controller for all recent multi-core PULP chips, taking care of autonomous I/O, advanced data pre-processing, external interrupts, etc. The PULPissimo architecture includes a RI5CY core or the Ibex the one as main core, autonomous Input/Output subsystem (uDMA), support for Hardware Processing Engines (HWPEs), interrupt controller, etc.

III. Prior Work

A. ARM implementation of chain of trust

When the SoC hardware is powered on, the CPU automatically executes from the start address (such as the reset vector). The start address is configured in a predefined location in an immutable memory space like ROM. Hence this is the root of trust as this cannot be tampered with, providing assurance that the bootloader cannot be bypassed. The bootloader is divided into several stages, the first of which is the immutable bootloader. The later stages might be loaded from non-volatile storage into RAM and executed there or executed directly from flash.

B. WolfBoot Implementation of Secure Boot

WolfSSL [6] is an open-source project that provides a secure bootloader for firmware authentication. The firmware is verified at boot time using WolfCrypt ECDSA or ED25519. Hash algorithms used for the implementation are SHA-256 and SHA-3-384. Since the solution is provided as a software library it is loaded into the flash making it vulnerable to flash corruption. Additionally, if any parameters in the wolfBoot are modified, they cause a failure in initialization and leave the device vulnerable to attacks.

C. Limitations

- In the Hack@DAC 2018 hardware security competition, it was exposed how the lack of secure boot implementation for PULPissimo leaves it vulnerable to boot code injection at privileged levels which can cause sensitive information to be compromised. [1]

- Bootloader firmware units like wolfBoot might provide a false sense of security, as an attacker able to compromise such units can gain full control over the target system [2]. It is up to the user to verify the security of the firmware and to ensure it does not contain any backdoors, which is impossible if the code is proprietary.

IV. Problem Statement

With the ease of physical access to consumer devices, embedded systems are especially vulnerable to physical attacks. While PULP-based frameworks are highly versatile, not having inbuilt secure boot implementations creates a space for third party bootloaders. While these bootloaders claim to be a plug and play solution to enable hardware security, their presence can create themselves be a complex security threat to the system they're added in. With the rise in popularity of RISC-V architectures, and the PULPissimo framework in particular, the necessity for an inbuilt secure boot solution arises too.

V. Proposed Approach

As mentioned previously, the PULPissimo framework and other open-source RISC-V architectures do not have a secure boot implementation thereby making any applications developed on the platform vulnerable to a multitude of attacks primarily including data (including firmware) theft or firmware corruption. Since RISC-V architecture and the PULPissimo framework in particular are gaining popularity, we believe this is required for the reliability of any application[14] that uses the PULPissimo framework. Through this project we aim to provide the following deliverables:

- Basic Secure Boot – A trusted application prior to execution of the user application that validates the flash before launching the application.
- Validation – Procedure to test the functionality of the implemented secure boot.
- A guide for the community aiming to implement secure boot onto the PULPissimo framework and for developing secure applications.

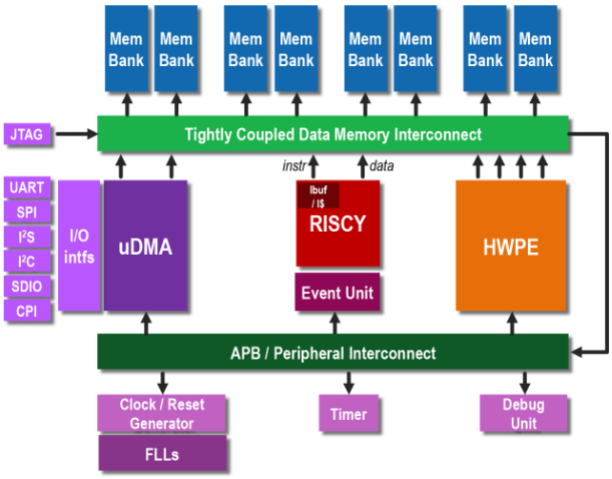

Fig 1: PULPissimo architecture

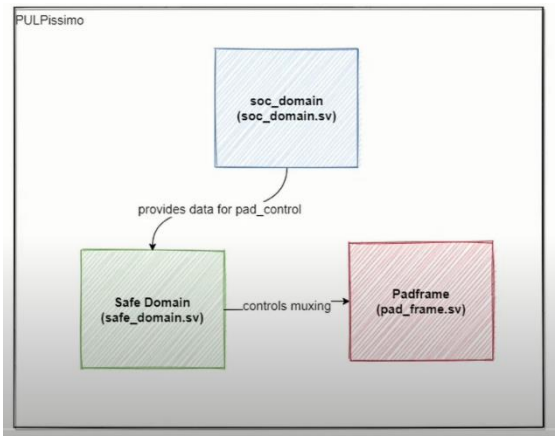

Fig 2: SoC Architecture

VI. PULPissimo Setup

A. Architecture Overview

PULP has an advanced microcontroller architecture with autonomous I/O, external interrupts, and a tightly coupled cluster of processors in which compute-intensive kernels can be offloaded from the main processor.

The architecture includes:

- RISC-V core
- Input/Output subsystem (uDMA)
- Memory subsystem
- Hardware Processing Engines (HWPEs)
- Interrupt controller
- Software Development Kit (SDK)

PULP has I/O interrupts like:

- SPI
- I2C
- UART
- JTAG

Hardware Processing Engines are hardware accelerators. They are used for individually handling the operation through memory for faster access.

The main entry (pulpissimo.sv) to the PULP core is mainly divided into 3 components:

1. SoC Domain / PULP SoC
   This is the heart of the SoC. This domain maintains the connectivity of all the subsystems in the SoC. It wraps the actual heart of the SoC – The pulp SoC IP. The main SoC fabric consists of 32-bit PULP chip. Most of the signals are used when additional multi-core clusters are present.

2. Safe Domain
   This domain contains logic that must be power gated. In this domain, power management is controlled. Pad_control module multiplexes functionalities of IO pads. Rst_gen module synchronizes the reset signal to the reference clock.

This is only used for modules within the safe domain.

3. Padframe
   This contains the tech-independent wrappers of the IO pads. All the bits except output enable are H/W configured.

SoC Schematic Overview:

There are two clock domains: SoC clk & Peripheral clk
There are about 32 interrupts in Pulp RISCV.
APB SoC control consists of the register file with global configuration.

It follows the TCDM protocol which is a single-cycle latency protocol. It is used for the connection between the core and memories. But it doesn't allow multiple transactions.

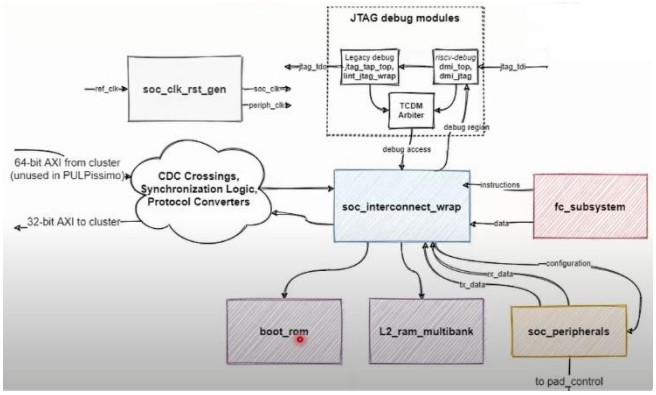

Fig 3: Schematic Overview

PULPissimo environment setup:

The entire setup is done on an Ubuntu machine. QuestaSim and Vivado 2019.2 tool were installed on Ubuntu and used for the application.

Installation of PULP platform specific toolchain:

- PULP RISC-V GNU Compiler Toolchain
- On Ubuntu there are several packages that are required to build the toolchain.

Simple Runtime:

- Using simple runtime we could run and write programs.
- Firstly, we had to export the pulp riscv toolchain to our toolchain path.
- Updated the pulp-runtime configuration.
- Completed the runtime simulation environment setup.

Software Development Kit:

- For a more complete runtime (drivers, tasks etc.), we installed the software development kit for PULP/PULPissimo.
- Linux dependencies were built for the initialization of SDK
- The config and build scripts were updated.
- The GNU toolchain was built along with the python dependencies.
- Use of virtual environment was essential for python packages.
- After building the pulp-sdk, the necessary environment variables were set.

RTL simulation platform:

- QuestaSim installation was required for the RTL simulation platform.
- The latest version of the IP's were checked out composing of the PULP system.
- The required scripts were generated and updated. Dependencies were resolved.

After having finished these tasks for the setup, now it is possible to develop our own RTL, adding new IP to PULPissimo or running a set of examples. Building and using the virtual platform is also possible. For our application, we needed bitstream generation.

FPGA setup for PULPissimo:

Bitstream generation (generated for ZedBoard):

- .bit the bitstream file for JTAG configuration
- .bin the binary configuration file to flash to a non-volatile configuration memory

Bitstream Flashing:

- Programmed ZedBoard using Vivado.

GDB and OpenOCD:

- The binary has to be loaded into PULPissimo's L2 memory.
- We need OpenOCD in parallel with GDB to setup communication with internal RISC-V debug module.
- JTAG is used as a communication channel between OpenOCD and the Core

VII. Crytpo Block

A. Data Hashing

As mentioned previously, the PULPissimo framework and other open-source RISC-V architectures do not have a secure boot implementation thereby making any applications developed on the platform vulnerable to a multitude of attacks

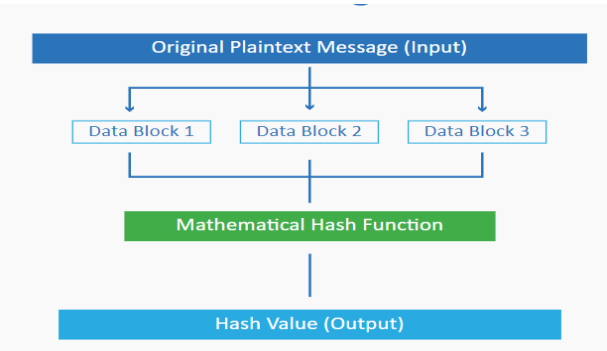

Fig 4. Hashing of Data

primarily including is a process that allows you to take data of any size and apply a mathematical process to it that creates an output that's a unique string of characters and numbers of the same length. The hash value generated by a hash algorithm is not reversible i.e., the input data cannot be obtained from the generated hash value making it a one-way cryptographic function. Hashing is used in Digital signature verification, Password hashing, SSL Handshake, and Integrity checks.

The following properties of a hash function make it useful in data authentication: deliverables:

- **Avalanche Effect**: Even the smallest change in the input results in a significant change in the hash value output.
- **Irreversibility**: Can easily convert the input to hash but cannot derive hash from the input.
- **Determinism**: Output hash length is independent of the size of input data and only depends on the hash algorithm used.

It is important to note that hash doesn't prevent intruders from snooping on the data it only ensures the data integrity. Some common hashing algorithms include MD5, SHA-1, SHA-2, NTLM, and LANMAN. SHA256[4] algorithm, is one of the most widely used hash algorithms.

We use SHA256[4] to generate hash for the firmware. A section is created in the device to save the hash and upon initial boot the hash is calculated and verified against the saved value to validate the integrity of the firmware.

B. Data Encryption

An encryption algorithm is **the method used to transform data into ciphertext**. The ciphertext can only be decoded by the user with the correct decryption key. One of the widely used encryption algorithm is RSA[3]. As seen in Fig A. public and a private key pair are created, with public key being open to anyone while private key would be held by the creator.

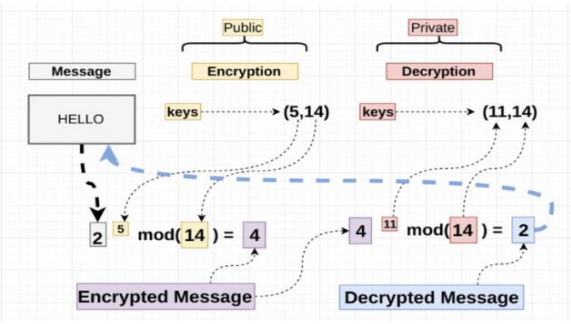

Fig 5: Encryption and Decryption using public

VIII. Implementation

The first obstacle to the implementation of this project is the understanding of the PULPissimo software and make architecture which was our initial step. The main installation directory of the PULPissimo software stack will be referenced as *<PULPissimo_home>* for the rest of this section. The PULPissimo make picks up the examples from the *<PULPissimo_home>/pulp-rt-examples*. A copy of the existing example was used for our implementation.

A. Boot Validation

The secure_boot_launcher() is the entry point for the boot code which then performs certain steps prior to successfully launching the user application. The rom_boot_validate_flash_boot() is called to validate the boot section. This function first computes the SHA256 hash of the flash code. This is then compared with the user hash (discussed shortly) to confirm the integrity of the program. As hash is a one-way function capable of detecting any changes, we can catch any issues resulting in a binary mismatch. The user application is launched only if the hash matches.

B. User Hash

C. Once the application is built, the SHA256 hash of the entire binary file needs to be appended to the binary file at a specific location that can be read by the boot. However, the PULPissimo build outputs an elf file instead of a hex file which makes the placement of the hash at a specific location hard. To overcome this, the elf file is first converted to a hex file using the PULP RISCV Toolchain. The calculated hash is appended to the hex file and the hex file is again converted back to the elf file for the simulation/build flow to pick up. The hash needs to be placed in a specific location. For this, a section is created in the linker script at the very end. This section holds the hash and other flags queried by the secure boot.Version and Magic Number

The user can also add the version of the executable being loaded onto the device. The version of the previous executable (v1) is stored by the secure boot. When validating the incoming application the version of that application (v2) is compared with v1. If v1 is lesser than v2 or if it is equal, the application is launched. If not, the application is not loaded. This is because an attacker can try to load the executables of previous versions to exploit any vulnerabilities that are patched in the newer releases.

The secure boot also checks for a specific value stored which is a sequence of alternate 1s and 0s called the magic number. Only if the magic number exists, the application is loaded. There also exists an is_valid flag which the bootloader enables once the validation is successful. Only if both the magic number and the is_valid flag match, the application is executed.

The secure boot on every launch invalidates the magic number and the is_valid flag. This is also enabled and checked every time the application is validated, and the version is legal. As these values are updated on every boot, there will not be any stale values that can be exploited. Additionally, the boot code is split into multiple sections, each validating a small part that provides additional security against clock gating which results in any single check being skipped.

IX. Verification

The secure boot is simulated for the ZedBoard platform using the QuestaSim tool.

```
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] ************************************************************
# [STDOUT-CL31_PE0] *     ECE 751 - Implemntation of Secure Boot on RISC-V     *
# [STDOUT-CL31_PE0] ************************************************************
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: Invalidating flash boot flags
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: Validating the boot version
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: Initiating flash boot validation
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: HASH MATCH - 4b0584877256164a2825676a6851098ae8599f2d7b7efe3e970404fbb6fcf034
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: Flash Boot validation successful
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: Enabling all flash boot flags and magic number
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: Launching user application
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] **   Reached User Code   **
```

Fig 6: Successful validation of secure boot

The secure boot is tested for its functionality with all the configurations matching with no corruption in the flash contents. The output as shown in Figure 6 demonstrates the usual flow of the secure boot when the validation is successful.

For verifying if the secure boot can actually detect any corruption in the contents of the flash, we can manually modify the contents of the hex file while converting back to the elf file or by injecting a code that changes a random flash value. For the purpose of this project, we have used the latter implementation wherein the malicious code can be introduced by enabling a macro either by using the makefile or within the code itself. Figure 7 shows the simulation logs when the malicious code is enabled.

Similarly, the version number and the magic number is tested to ensure the complete functionality of the secure boot implementation.Setup and debug of pulp environment (Simple Runtime, Software Development Kit) for simulations.

X. Discussion

- According to the Pulp owners, PULPissimo has been implemented on ZedBoard. However, we saw a lot of challenges in synthesising the design for the ZedBoard. The main bottleneck here was the limited number of LUT's for ZedBoard. After few iterations of debug, it was seen that version 7 of pulp cannot be flashed onto the ZedBoard due to the limited slices wrt to design in version 7. However, the version 6 design with no HWPE modules were able to fit onto the slices available in ZedBoard. Bit file was successfully generated for the ZedBoard FPGA.
- The PULPissimo stack also consists of a boot_code directory which provides the source of the code for the ROM boot. However, when tested with the final generated binary file, the binary file did not contain any symbols pertaining to the boot_code indicating the final binary does not use the provided source code. The core also is configured to initiate execution directly from the flash boot. As a result, the secure boot is implemented as the first code to be launched as part of a normal boot. However, the code is modular and is a pick-and-place solution to be implemented in the ROM boot.
- The PULPissimo framework's crypto IP is not open source which is why we could not integrate it with our implementation. Therefore, due to the software implementation of the crypto algorithms, the processing time is higher. Additionally, a digital signature algorithm is not included in the implementation due to its need for computational capability and memory requirements.

XI. Conclusion

As part of this project, we have implemented a basic secure boot for the PULPissimo platform. We have also validated the secure boot functionality by manually corrupting the device data.

```
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] ************************************************************
# [STDOUT-CL31_PE0] *     ECE 751 - Implemntation of Secure Boot on RISC-V     *
# [STDOUT-CL31_PE0] ************************************************************
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: Invalidating flash boot flags
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] WARN: Corrupting device code
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: Validating the boot version
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: Initiating flash boot validation
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: Device Hash - dbefea4d026f4fd025c0605f7ea252cfbdddb2dfe71e73a8a2ec931b55cbf1cf
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] NOTE: User Hash - 7312e60d5da6b25d72fa4572cbde1ba1269eef32c28c08ed69ab983dd1d88425
# [STDOUT-CL31_PE0]
# [STDOUT-CL31_PE0] ERROR: Flash Boot validation failed. Integrity check failed
```

Fig 7:Manual data Corruption

I. INDIVIUAL CONTRIBUTION

Ujwal:

- Understanding the PULPissimo platform and Pulp RISC-V architecture.
- Tool installations, GNU Compiler Toolchain and environment setup for build flow.
- FPGA setup and implementation for Pulpissimo.
- ZedBoard configurations and bit file synthesis.

Pragna:

- Researched the existing implementation of secure boot.
- Verified SHA256 implementation and helped in integration with the secure boot process.
- Verified RSA implementation.
- Contributed to the presentation and report components of the project.

Rakshith –

- Understanding the PULP software architecture and the build flow.
- Compiling and understanding the PULP-specific programming tools like RISC-V OpenOCD, GDB, and the RISC-V Compilers.
- Designing and implementing the secure boot code flow
- Binary file editing and script file generation
- Validation of the implemented secure boot

Nitya–

- Researched existing secure boot implementations
- Understanding the PULPissimo architecture
- Designing the secure boot code flow
- Contributed to the presentation and report components of the project.

REFERENCES

[1] NJ2 Dessouky, Ghada & Gens, David & Haney, Patrick & Persyn, Garrett & Kanuparthi, Arun & Khattri, Hareesh & Fung, Jason & Sadeghi, Ahmad-Reza & Rajendran, Jeyavijayan. (2018). When a Patch is Not Enough - HardFails: Software-Exploitable Hardware Bugs.

[2] NJ1 L. Morel and D. Couroussé, "Idols with Feet of Clay: On the Security of Bootloaders and Firmware Updaters for the IoT," 2019 17th IEEE International New Circuits and Systems Conference (NEWCAS), 2019, pp. 1-4, doi: 10.1109/NEWCAS44328.2019.8961216.

[3] Xin Zhou and Xiaofei Tang, "Research and implementation of RSA algorithm for encryption and decryption," Proceedings of 2011 6th International Forum on Strategic Technology, 2011, pp. 1118-1121, doi: 10.1109/IFOST.2011.6021216.

[4] A. L. Selvakumar and C. S. Ganadhas, "The Evaluation Report of SHA-256 Crypt Analysis Hash Function," 2009 International Conference on Communication Software and Networks, 2009, pp. 588-592, doi: 10.1109/ICCSN.2009.50.

[5] On the Path to a Secure Boot Solution for RISC-V [https://www.veridify.com/path-secure-boot-solution-risc-v/]

[6] WolfBoot secure Bootloader [https://www.wolfssl.com/products/wolfboot/]

[7] J. Haj-Yahya, M. M. Wong, V. Pudi, S. Bhasin and A. Chattopadhyay, ”Lightweight Secure-Boot Architecture for RISC-V System-on-Chip,” 20th International Symposium on Quality Electronic Design (ISQED), 2019, pp. 216-223, doi: 10.1109/ISQED.2019.8697657.

[8] Platform Security Boot Guide – Arm [https://developer.arm.com/documentation/den0072/latest]

[9] A Survey on RISC-V Security: Hardware and Architecture [https://arxiv.org/pdf/2107.04175v1.pdf]

[10] The whys and hows of secure boot [https://www.embedded.com/the-whys-and-hows-of-secure-boot/]

[11] 7. Byung-Chul Choi, Seoung-Hyeon Lee, Jung-Chan Na, Jong-Hyouk Lee, “Secure firmware validation and update for consumer devices in home networking”, IEEE Transactions on Consumer Electronics (Volume: 62, Issue: 1, February 2016), doi: 10.1109/TCE.2016.744856

[12] PULP Platform [https://pulp-platform.org/]

[13] PULP based chips [https://pulp-platform.org/docs/hipeac/acaces2020/04 PULP Chips.pdf]

[14] Online SHA256 tool [https://emn178.github.io/online-tools/sha256.html]

[15] RISC-V Instruction Set Manual [https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf]

[16] PULPissimo README [https://github.com/pulp-platform/pulpissimo/blob/master/README.md]

[17] Configure and run PULPissimo [https://singularitykchen.github.io/blog/2020/12/20/Tutorial-Configure-and-Run-Pulpissimo/]

[18] A Deep Dive into HW/SW Development with PULP [https://www.youtube.com/watch?v=B7BtaYh3VqI]