# SecureBoot implementation for PULPissimo RISC-V framework

ECE 751 Fall 2022

Presenters : Nitya Joshi, Pragna Pulakanti, Rakshith Macha Billava, Ujwal Ravichandra

# Background

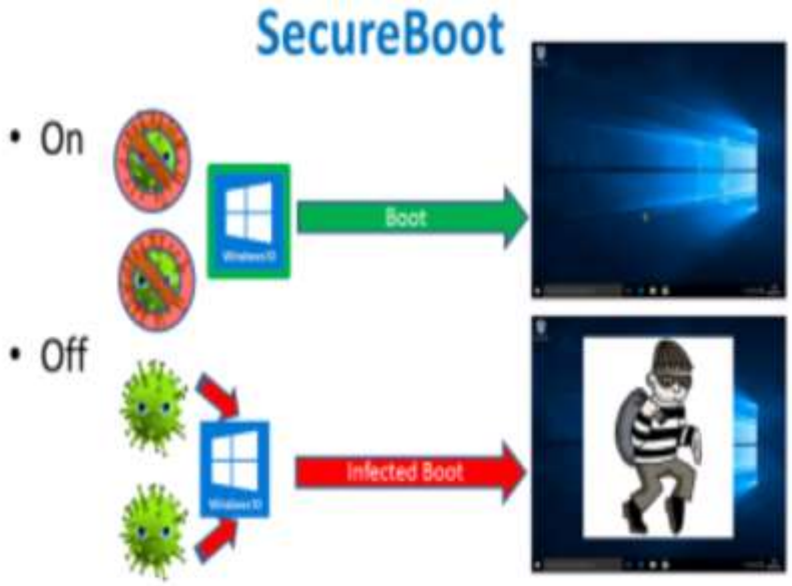


Fig 1: Secure Boot Example

# Secure Boot

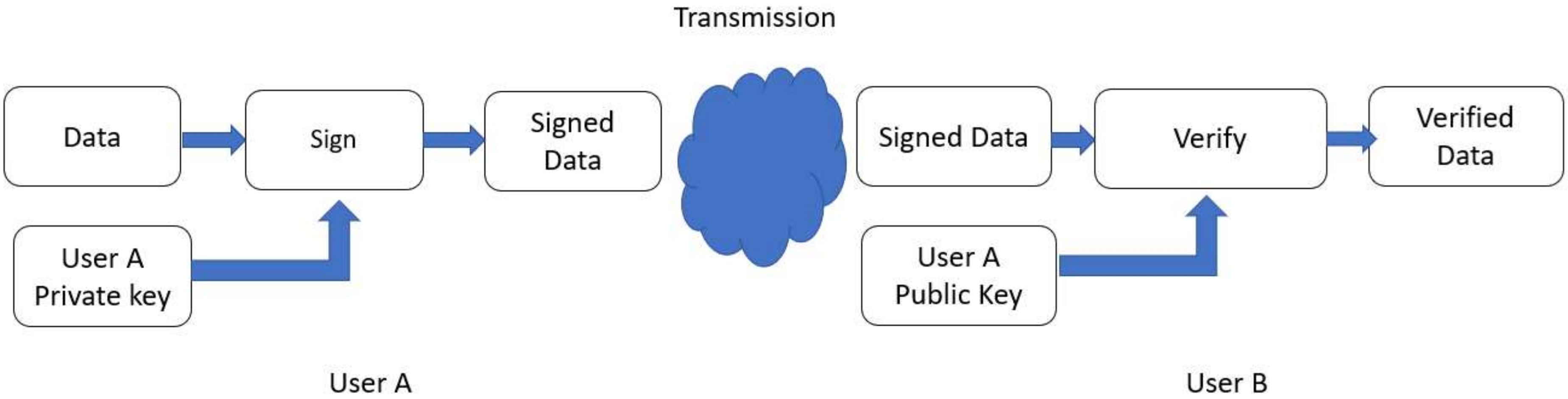


Fig 2: Sign and Verification of the data

# PULP platform

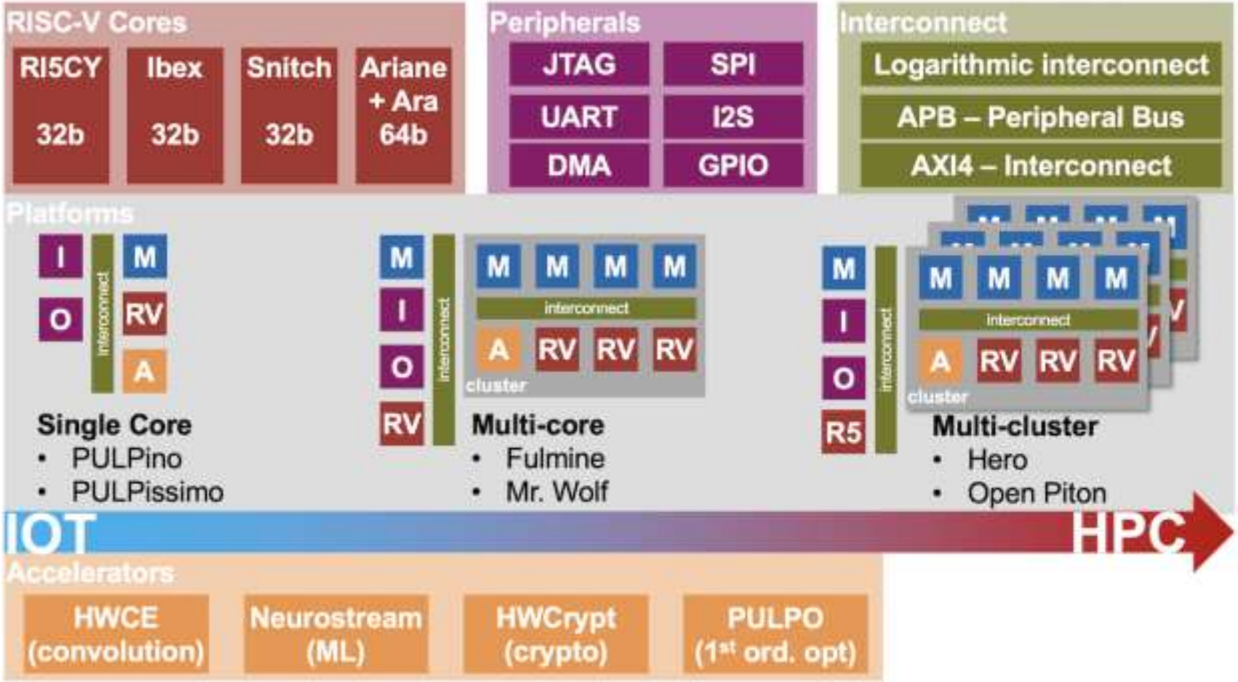


Fig 3: PULP platform family

# Motivation

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.

Bootloader firmware units (like wolfBoot) might provide a false sense of security, as attackers might be able to compromise such units.

# Objectives

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 user application before launching the application.
- 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.

# PULPissimo RISC-V Architecture

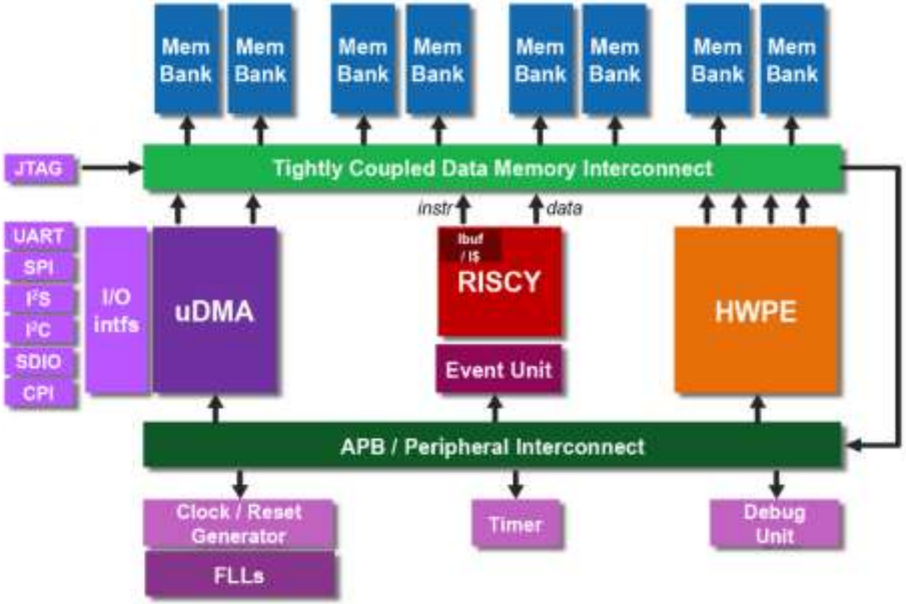


Fig 4: PULP RISC-V architecture

# PULP Architecture

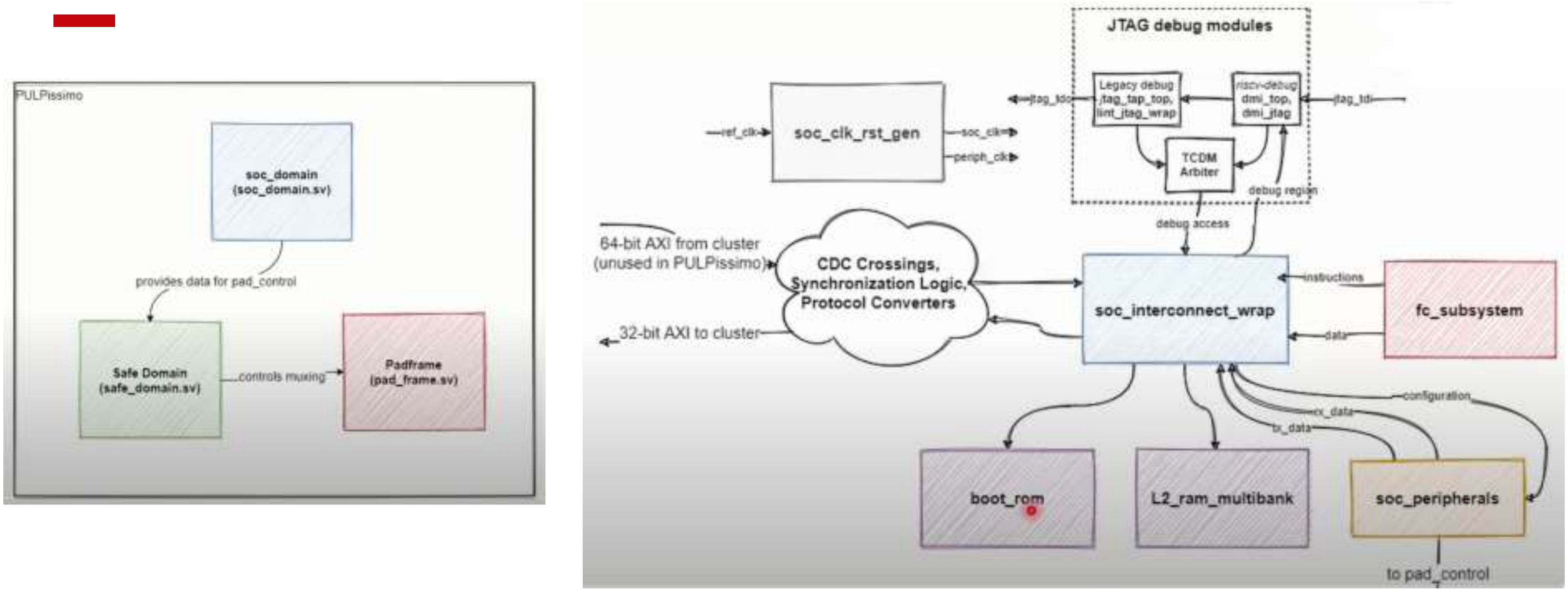


Fig 5: SoC Architecture

# PULPissimo Setup

Vivado 2019.2 and QuestaSim tool were used.

**Installation of PULP platform specific toolchain:**

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

**Simple Runtime:**

- Export the PULP RISC-V toolchain to our toolchain path.
- Update and use the pulp-runtime configuration.
- The runtime simulation environment is setup.

# PULPissimo Setup

**Software Development Kit:**

- For a more complete runtime (drivers, tasks etc.)
- Linux dependencies were built for the initialization of SDK
- 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.

# FPGA setup for Pulp

- 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

# Data Hashing

- Map data of any size to a fixed length ->Hash Digest
- One way cryptographic function.
- Common hashing algorithms include **MD5, SHA-1, SHA-2, NTLM,** and **LANMAN**
- A good hash function must have

  1. Irreversibility

  2. Deterministic Property

  3. Has an Avalanche effect

# SHA256 : Pre-Processing

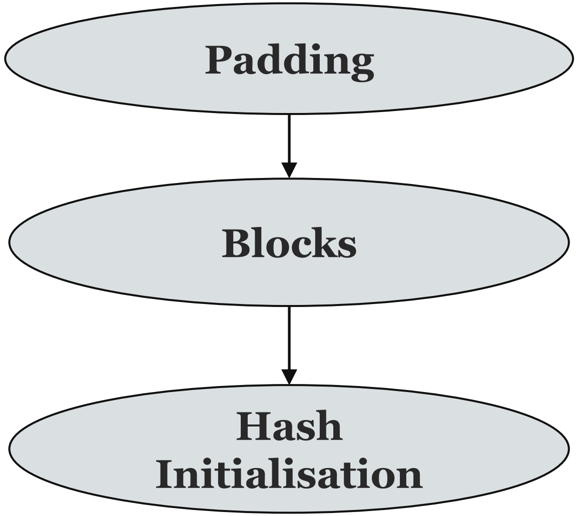


Fig 6: Pre-processing the input data

$$H_0^{(0)} = \texttt{6a09e667}$$
$$H_1^{(0)} = \texttt{bb67ae85}$$
$$H_2^{(0)} = \texttt{3c6ef372}$$
$$H_3^{(0)} = \texttt{a54ff53a}$$
$$H_4^{(0)} = \texttt{510e527f}$$
$$H_5^{(0)} = \texttt{9b05688c}$$
$$H_6^{(0)} = \texttt{1f83d9ab}$$
$$H_7^{(0)} = \texttt{5be0cd19}$$

Fig 7: square roots of the first eight prime numbers

# SHA256 : Algorithm

.

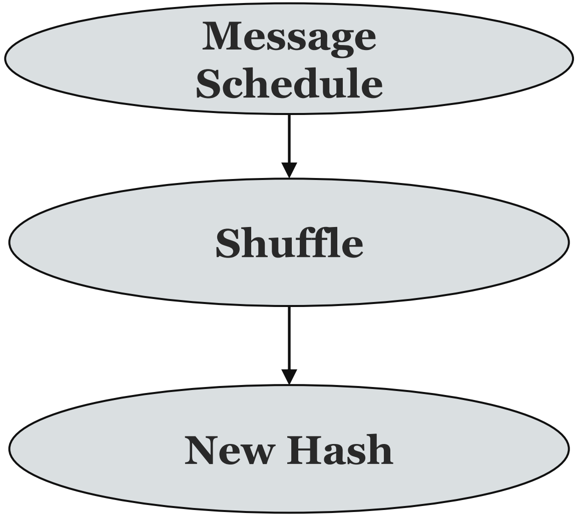


Fig 8: Generation of Hash Value

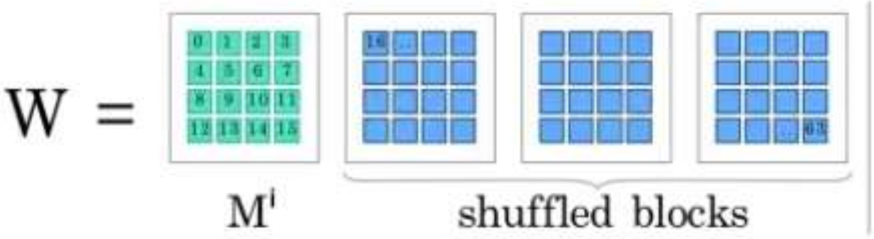


Fig 9. Message schedule, Wi, 512-bits

# Encryption

- Input data to cipher text.
- Can only be decrypted by intended users.
- Commonly used encryption algorithms **AES, RSA, ECC**

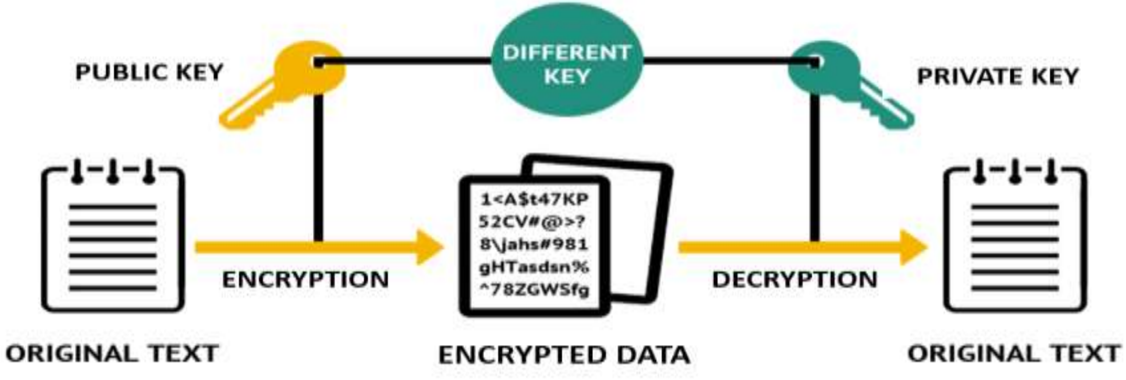


Fig 10: Encryption

# RSA Encryption

- Two prime numbers P and Q
- n=p*q and ϕ=(p-1)*(q-1)
- Public key(n,e)
- Private key(d,e)

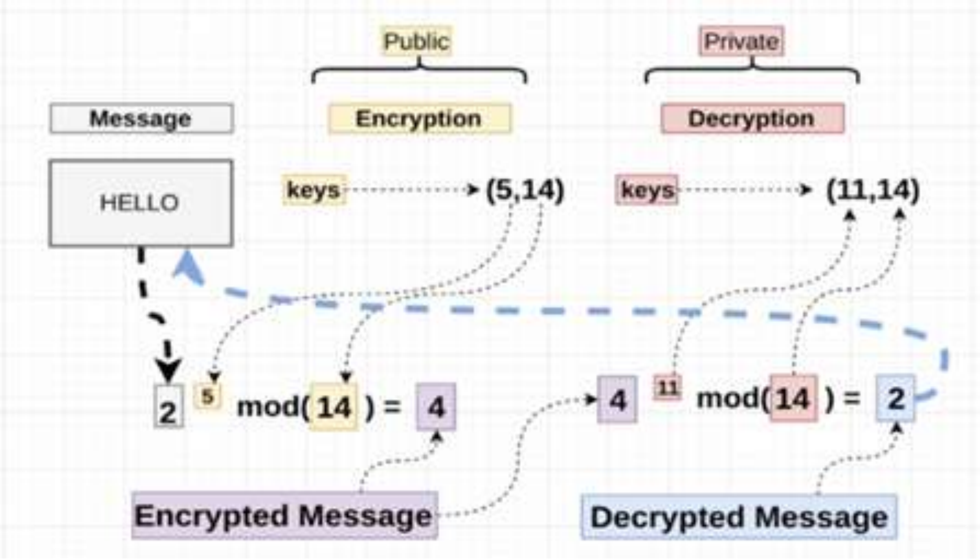


Fig 11: RSA Encryption

# Secure Boot Implementation

# Boot Validation

- Compute SHA256 hash of the device flash code
    - Software implementation of SHA256
- Compare with the user hash stored in the device
- Hash should match to launch the user code

# User Hash

- Calculated using the original binary
- riscv32-unknown-elf-objcopy tool is used
- Convert executable -> elf -> hex
- Add hash
    - Scripting
    - Manual - Intel HEX format
- Convert edited hex -> executable
- Linker script edits for custom section

```
:020000041C07D7
:10FEE00012345678901234500000000000000000D8
:10FEF0000000000000000000000000000000000002
:10FF000000000000000000000000000000000000F1
:10FF100000000000000000000000000000000000E1
:10FF200000000000000000000000000000000000D1
:10FF300000000000000000000000000000000000C1
:10FF400000000000000000000000000000000000B1
```

Fig 12 - Intel Hex format editor

# Secure Boot Flags

- Version Number
  - Do not launch if incoming version number is smaller
  - Prevent exploitation of vulnerabilities in previous versions
- Magic Number
  - Specific sequence of alternate 1s and 0s
  - To prevent unintended edits to boot flow
- Is_Valid
  - Save the state of the validation

# Verification - Regular Secure Boot Flow

```
# [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   ***
```

# Verification - Manual Data Corruption

```
# [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
```

# Conclusion

- Basic secure boot for the PULPissimo RISC-V platform
- Validation of the implemented secure boot
- A guide for the community aiming to implement secure boot

# THANK YOU