Design and Implementation of Secure Boot Architecture on RISC-V using FPGA

Loo, Tung Lun[1] and Mohamad Khairi Ishak[2]

[1] *School of Electrical and Electronic Engineering, Universiti Sains Malaysia, 14300, Nibong Tebal, Pulau Pinang, Malaysia.*

[2] *School of Electrical and Electronic Engineering, Universiti Sains Malaysia, 14300, Nibong Tebal, Pulau Pinang, Malaysia.*

Abstract

There are many well-known open-source bootloaders solutions available today such as UEFI/BIOS, Coreboot and Uboot. Recently, RISC-V as an open-source Instruction Set Architecture, has gained a lot of attention in new embedded products creation and academic research purpose. In this study, RISC-V Instruction Set Architecture boot flow and boot solutions are studied, simulated, experimented, and summarized. Security feature is implemented in firmware and measured against non-secured firmware to compare boot performance without security inclusion. A new proposed method to create a security block in Register Transfer Level to generate Secure Hash Algorithms 5 digest is implemented using Field Programmable Gate Array. The performance of this method is analyzed with the numbers of logic gate required and the execution time in software versus hardware. As a result of this study, it is observed that in simulated environment, secured firmware incurred 3.3 Megabytes of additional binary size and 747ms (35 %) additional boot time compared to non-secured firmware. A hardware implementation is proposed in Field Programmable Gate Array (FPGA) to reduce the need for a larger size firmware and longer boot time to implement security. The results of this implementation indicate a requirement of 32,048 gates to implement a SHA512 IP that reduce software execution time by 1132 %.

Keywords: riscv, security, firmware

1. Introduction

All compute devices today are powered by a few processors Instruction Set Architectures (ISAs), predominantly x86, AMD, ARM, and MIPS which is later converged to RISC-V in 2021 [1]. These ISAs provide flexibilities and extensibilities to the different engineering audiences, creating tremendous opportunities today that benefits consumer in many custom applications and use cases, especially in the booming edge devices in Internet of Things world. While having multiple ISA options are good, it is often difficult to make a good decision on which architecture to go for, because there are many factors that contribute to design decision. Several key elements of consideration while picking an ISA are as below.

- Time-To-Market (TTM)

  The TTM factor is about how easy it is to enable an embedded system with collaterals provided by the ISA provider. For example, the development time of an engineering team (often called OEM/ODM) taking a new 11th Generation Intel chip and providing a full solution with it. Several key factors that directly impact TTM are the availabilities of documentation, system level open-source references and manufacturing technology.

- Cost

  This factor includes cost of licensing, software, and hardware development cost that the OEM/ODM needs to pay to get the products released.

- Design flexibilities

  The design flexibilities revolve around two key questions of "How easy it is to include a new custom IP in a new design?" and "How easy it is to land firmware, driver, and software support of a new IP?"

An ideal SOC would not only needs to be functional, but also be protected since the very early initialization flow to ensure no malicious code can be injected at any point before arriving at user space applications. To achieve this, firmware architecture becomes an important topic of exploration to identify the security scheme offered with different ISA and how a generic security approach can be deployed to implement security in each of them. The gap of today's security scheme is the ease of deployment whereby the enablers and users would often end up disabling security just to improve the performance of the system, reduce the TTM and product price. The consequence of this problem will be more unsecured devices being in the market, causing risks to everyone in the IOT chain. Therefore, this research will focus on identifying the boot elements of each ISA, methodology to enable secure boot, and how a security IP block can be added to the register transfer and firmware level to facilitate security such that it does not significantly jeopardize system performance and is easy to enable without much additional software development.

The key objective identified for this research is to evaluate the secure firmware feature, measure it against boot performance, and propose security enhancement through Field Programmable Gate Array (FPGA) for firmware booting mechanism with the evaluated security features of an open-source ISA. This enhancement could be potentially scale to close-source ISA.

In this paper, section 1 describes the introduction, problems and objectives of the study. Section 2 describes the background and previous work related to RISC-V processor, boot flow, and security. Section 3 describes the proposed method, which includes secure boot in Software (QEMU) and hardware implementation through FPGA. Section 4 describes the experimental setup results and discussion. Section 5 concludes the study and identify future improvement opportunities.

2. Background and previous work

2.1. Secure Boot

UEFI secure boot is independent of ISA architecture. It is part of the UEFI specification definition. The main goal is to have system firmware acting as a trusted entity to load any untrusted 3rd party firmware code, which includes bootloaders, payloads, or even operating system. To ensure UEFI secure boot is functional, an end user will need to enroll the secure boot security database with authenticated variables and sign untrusted applications to ensure that they are considered secure to execute. In a typical workflow, a 3rd party code provider will need to sign their components with a private key and publish the public key. Then the OEM or users can enroll the public key to the database, which is stored in a UEFI authenticated variable region [**?**]. During firmware boot process, the image verification procedure will verify the 3rd party code according to the image security database, using a verification flow entailed in Figure 1.

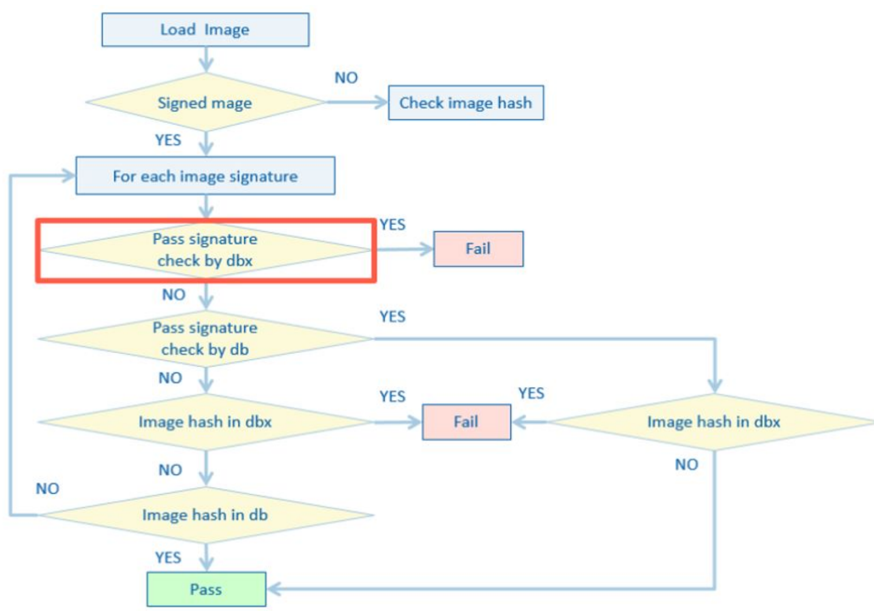

Figure 1: UEFI secure boot image verification flow [**?**]

Figure 2 shows the existing secure boot flow in firmware whereby CPU would validate the integrity of bootloader after jumping out of reset. If the bootloader data is identified to be invalid, the bootloader execution will halt, this flow is also applicable to booting OS. This mechanism ensures that an unauthorized application will not be able to execute, thus enable firmware to be a trusted entity from attack of malicious applications.

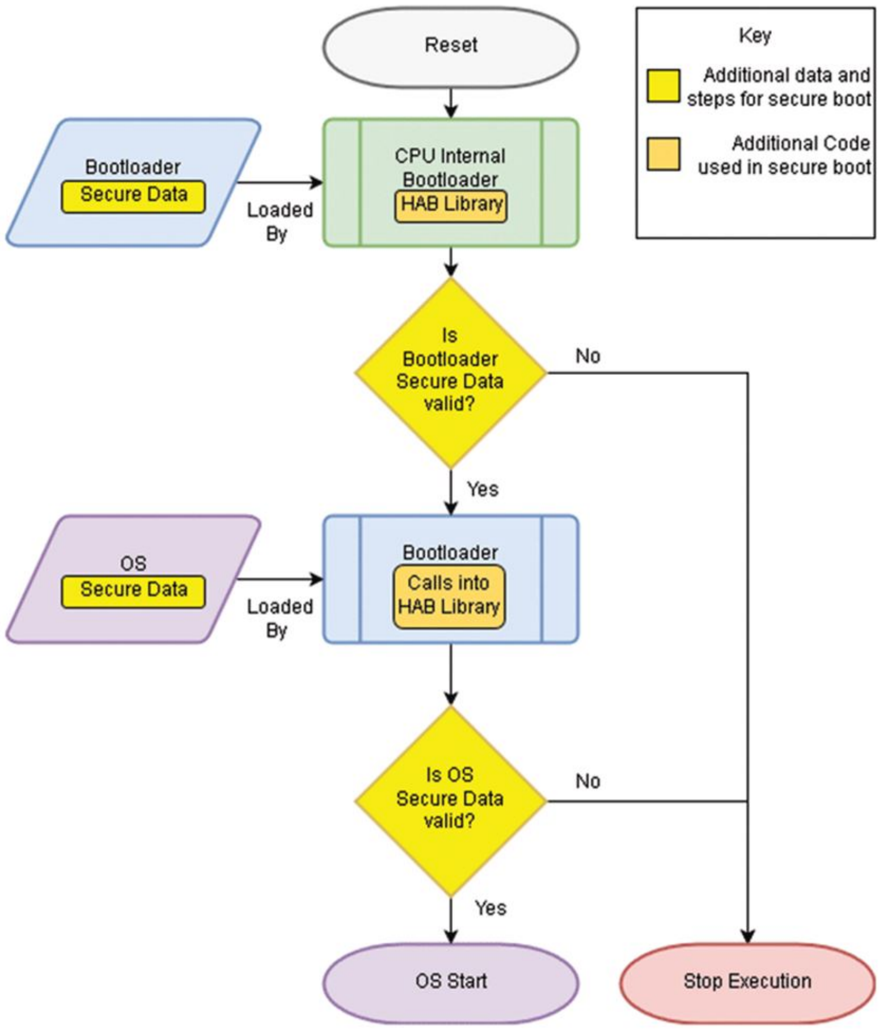

Figure 2: Existing Secure Boot Flow In Firmware [?]

Intel Boot Guard is a solution that extends the secure boot's root of trust from platform to the Platform Controller Hub (PCH). The mechanism contains a One-Time-Programmable (OTP) fuses that is burned to the chip during manufacturing process and that would be the hash of the master public key. This flow was described and productized in Dell EMC 14th generation server [**?**] . Figure 3 describes how to use a lower TCB (trusted computing base) to verify the stages from one to another. In this case, Authenticated Code Module (ACM) will verify PEI phase firmware volume, PEI will verify DXE phase firmware volume, and DXE will verify Operating System loader before hand over.

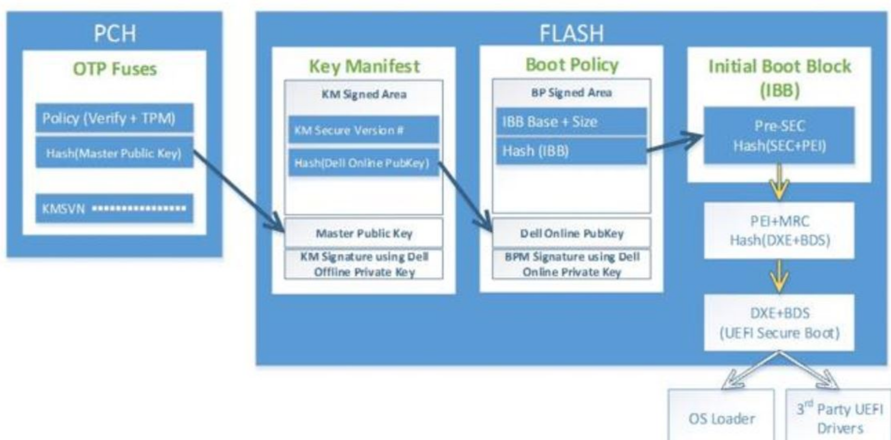

Figure 3: Intel Boot Guard [**?**]

Coreboot, an open source bootloader, implements verified boot, commonly known as vboot, which has very similar architecture as UEFI secure boot [**?**]. The root of trust is basically a read only portion of the SPI flash, which is commonly named as Google Binary Blob (GBB) area. This area contains a 4096- or 8192-bit public root RSA key that is used to verify the VBLOCK area to obtain the firmware signing key. During boot, the reset vector will copy the boot block in GBB and verify the next partition of the firmware code (FW_MAIN_A) to determine its legitimacy in executing it. The verification flow is demonstrated in Figure 4.

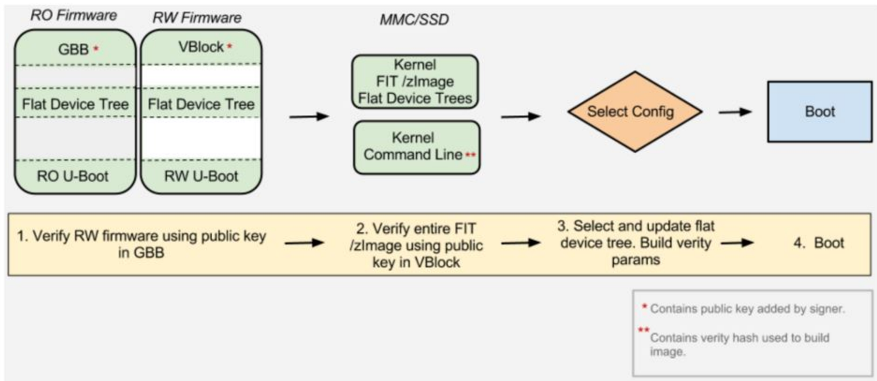

Figure 4: Coreboot vBoot [?]

Android Verified Boot (AVB) introduces the concept of LOCKED or UNLOCKED state. If a device is unlocked, the bootloader will proceed to boot even without any root of trust, where else in locked state, the bootloader would perform steps to verify boot using the pre-signed key hash in read only boot partition, which works in a very similar fashion as UEFI secure boot and Coreboot vboot. To change the device state, one can use “fastboot flashing [unlock — lock] command [?]. This is demonstrated in Figure 5.

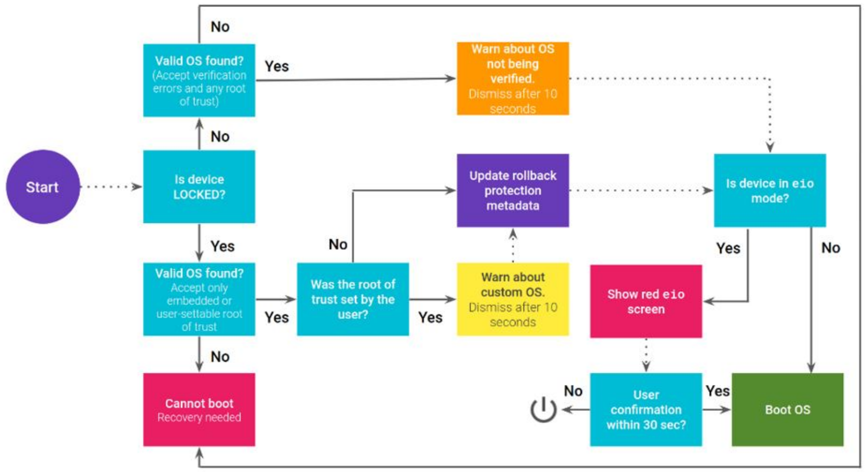

Figure 5: Android Verified Boot With Recovery Flow [?]

In this related topic of interest in RISC-V world, a lightweight secure boot architecture on SOC was introduced by [?]. This study introduced additional SHA3 hardware block to replace software-based authentication flow as demonstrated in Figure 6. With this approach, the comparison between hardware-based security scheme against software-based are highlighted in the paper.

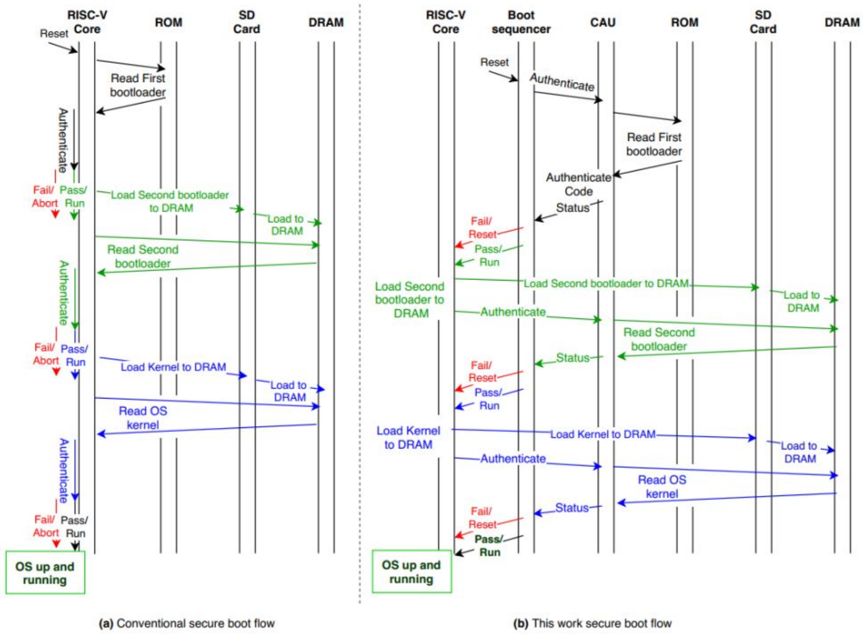

Figure 6: Authentication Flow To Implement Secure Boot [**?**]

2.2. RISC-V processor and security

RISC-V boot flow consists of ROM, loader, runtime, boot loader and OS, which aligns to the RISC processor modes that go from the most privileged mode to the least privileged mode as Figure 7 demonstrated.

Figure 7 also shows that all stages (Firmware, Hypervisor, OS, User space) are executed in sequence of exception levels like ARM64 fashion. ARM's EL3 has platform specific runtime firmware and has secure privileges, while RISC-V's M mode has platform specific firmware only and does not have secure privilege. ARM started with EL3, which is a secure world, while RISC-V starts from M mode, which is a bare metal machine code. The non-secure bootloaders in ARM uses ARM trusted firmware to switch to EL2, while RISC-V uses OpenSBI to switch into S-Mode from M-Mode. Also, ARM is close source while RISC-V is open source. Therefore, due to the open-source nature of RISC-V, the firmware stages between ROM code and OS (kernel space) are extremely flexible.

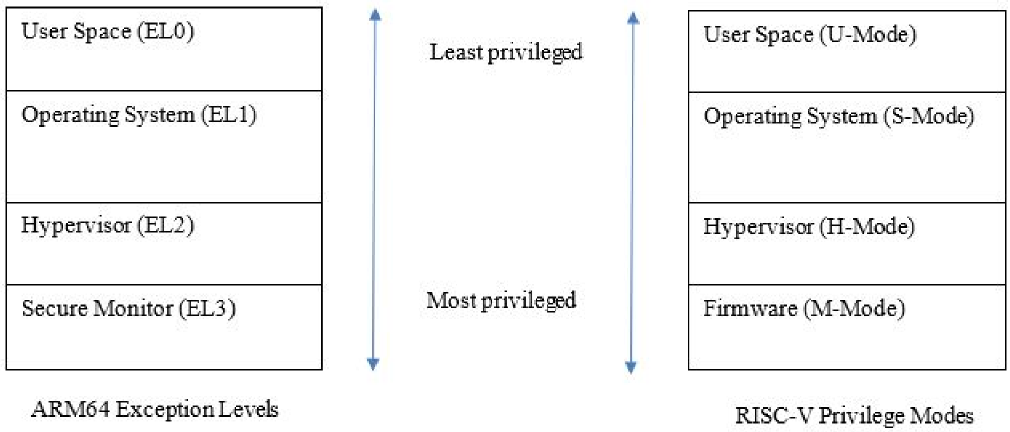

Figure 7: ARM and RISC-V Processor Modes Comparison

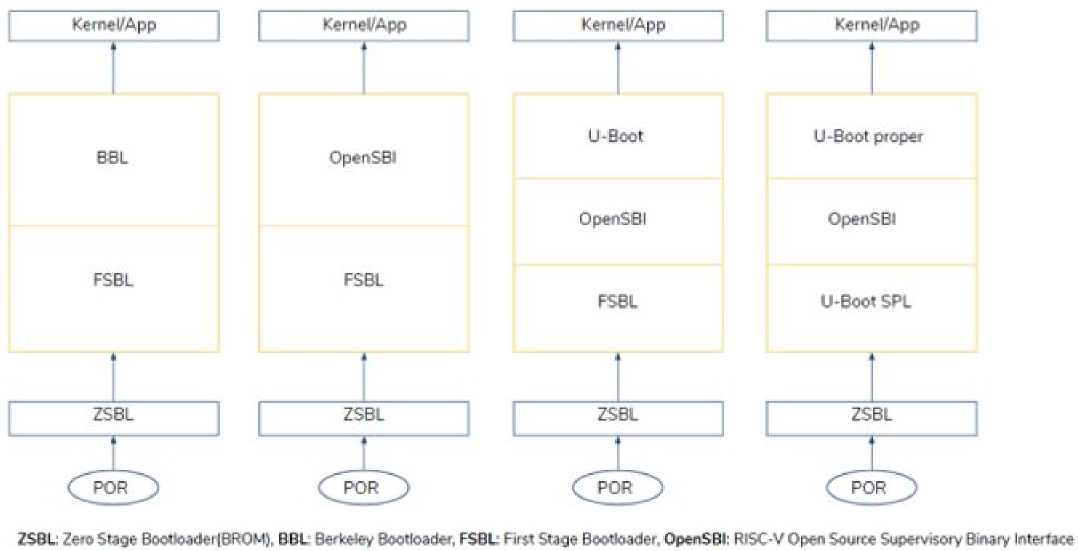

Figure 8: Different Firmware Flows For RISC-V [2]

Figure 8 shows some examples of different combinations possible after the "Zero stage Bootloader BROM", such that it contains a combination of U-boot, First Stage Bootloader, OpenSBI (RISC-V Open-Source supervisory binary interface) and BBL (Berkeley Bootloader). Even though having huge flexibility is good, this has eventually become a scalability issue if boot flow is not standardized, and all different RISC-V solutions adopt different methodologies. Maintenance and reusing existing source code and framework features become an issue.

Therefore, in 2020, the boot stage is further standardized to use U-Boot and OpenSBI as the only open source accepted methodology. Figure 9 shows the up-stream boot flow where OpenSBI sits right in the middle of boot phase between M-mode (firmware) and S-mode (U-boot) to provide all runtime services.

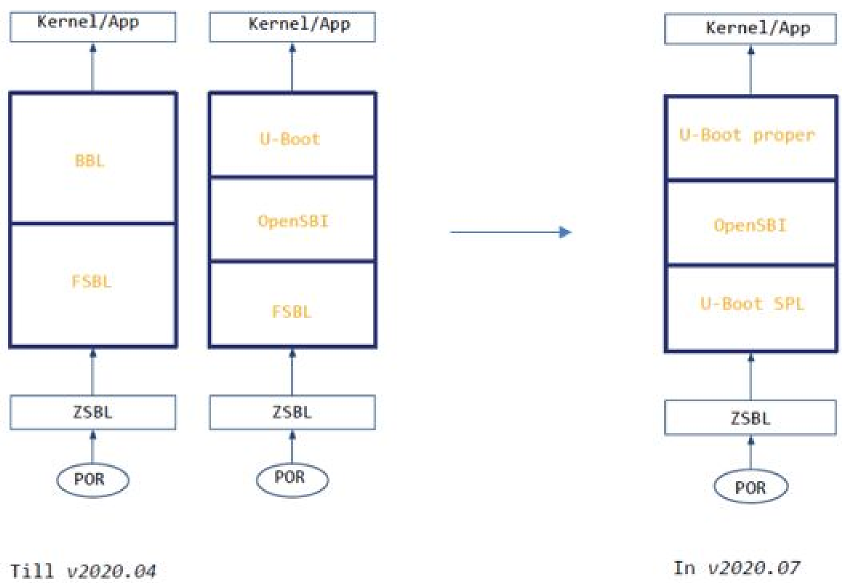

Figure 9: RISC-V Standard Boot Flow [2]

Figure 9 from Atish et al. also demonstrated that FSBL, which was SiFive specific, will be replaced by Coreboot/U-boot SPL. U-boot will then act as the last stage boot loader before Linux. OpenSBI standard started as an ingredient that is specific only to RISC-V, which makes it important to understand what it does and how it evolved over time. Jagan presented in China RISC-V Forum 2019 shows the evolving of RISC-V Supervisor Binary Interface (SBI) to Open-Source Supervisor Binary Interface (OpenSBI). In summary of the specification changes, the system calls type interface layer between firmware runtime, M mode and S mode were made modular, scalable, and extendable between all CPU and Silicon specific hardware configuration. OpenSBI now contains platform independent and dependent libraries, which support SiFive U540, Andes AE350, Ariane FPGA, Kendryte K210 and QEMU [2].

The current RISCV boot stage ported to UEFI is initiated by Hewlett Packard Enterprise since 2015 [3]. It described some architectural changes with OpenSBI as a platform structure layer that is callable by services during UEFI boot flow, for example during SEC phase, SEC module would call the OpenSBI initialization and platform initialization code and then return to PEI core once done. In PEI phase, PEI module extracts device tree information constructed in OpenSBI to be further consumed in DXE driver. In DXE and OS run time, supervisor, and hypervisor ecall interface is made available for any run time service required from OpenSBI. This flow is simplified in Figure 10.

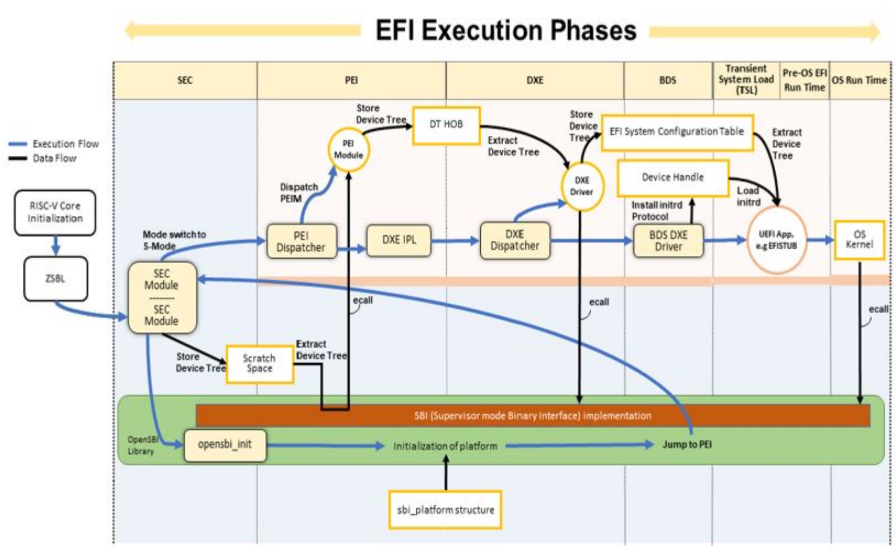

Figure 10: High Level Overview Of EFI Execution Phases with RISC-V [3]

The boot phase porting from RISCV essential services to UEFI framework involves more than just adding OpenSBI libraries as another underlayer service.

It also entails a volume top file (VTF) that generates a reset vector for UEFI bootloader to jump into, binding processor, converting RISCV ELF format to PE COFF, and porting of other UEFI libraries such as base memory, DXE real time clock, CPU arch, timer arch, reset protocol and CSR (control status registers). A MSCRATCH CSR is used to maintain a V machine mode trap handler in each of the boot phases (SEC, PEI, DXE core). More details of these work in each boot phase are simplified in Figure 11.

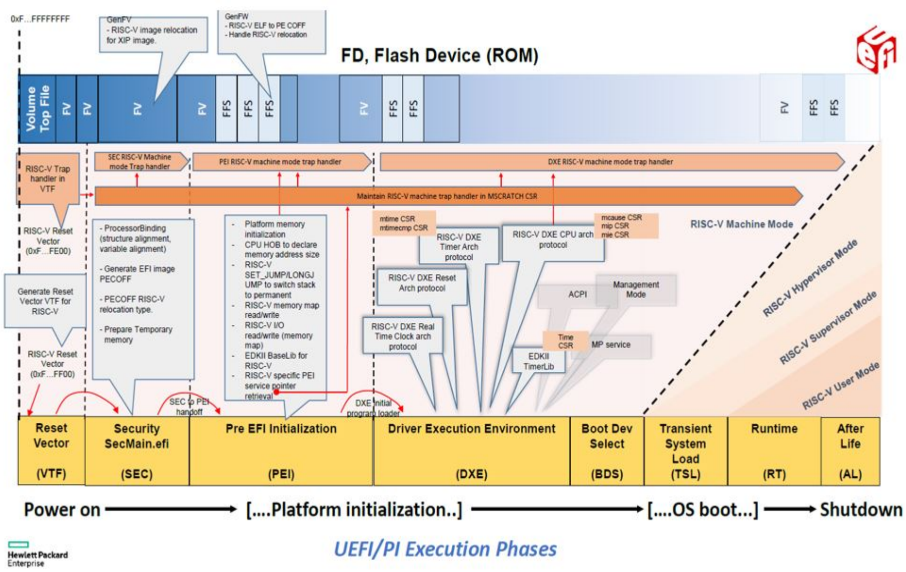

Figure 11: Detailed EFI Execution Phases with RISC-V [3]

In RISC-V, two security architectures stood out in addressing the challenges in x86 and ARM world today, namely Sanctum [4] and HECTOR-V [5]. Sanctum provides a similar enclave like concept as Intel SGX. Enclave page table registers and walker/transform logic are added on top of the LLC cache logic. On top of that, it adds measurement root of trust in a temper-resistant hardware. The goal of Sanctum is to target side channel attack that was claimed not covered in Intel SGX. This is achieved by adding page entry transformation logic. The root of trust resides in the CPU ROM where the code reads security monitor from untrusted flash memory and generate key based on monitor's hash. The software stack of an enclave is designed in a way that user's sanctum-aware runtime code and data communicate directly to the security monitor in machine's measurement root as Figure 12.

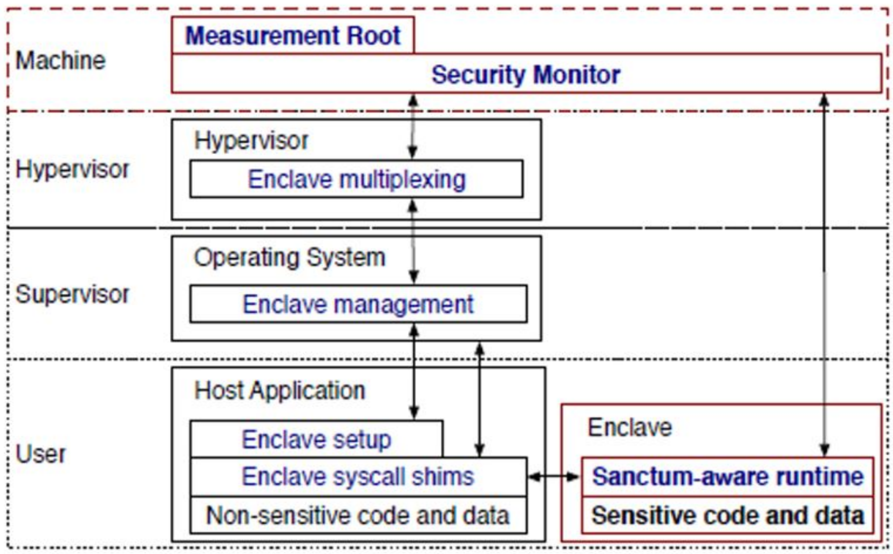

Figure 12: Sanctum Software Stack TCB [5]

Another popular piece that explores using heterogenous multicore architecture to realize a secure TEE design in RISC-V is HECTOR-V. It has RISC-V Secure Co Processor (RVSCP) embedded to application processor that enable HECTOR-V with mechanism to establish secure communication channels between multiple devices connected to the CPU. Its architecture is described in Figure 13. With this proposed heterogenous architecture, RVSCP provides hardware enforced control flow integrity and restricts I/O accesses to certain execution states. Concurrently, SiFive also developed WorldGuard architecture. In this architecture, each core gets assigned a world ID and process of the core is annotated with process ID. This ID is transported using the interconnect and requests from participants are filtered by peripherals, the memory, and the caches. This is mostly similar to HECTOR-V's design. The only difference is that WorldGuard transfer the security monitor ownership dynamically to any party for flexible use case, as compared to HECTOR that has concrete secure processor.

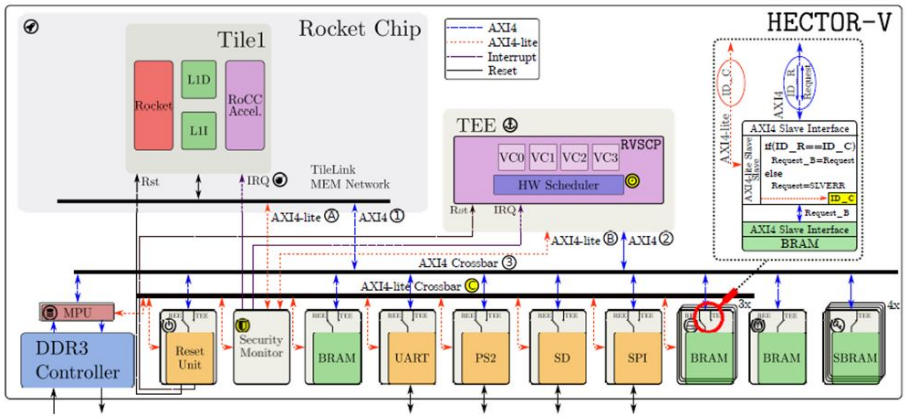

Figure 13: Hector-V architecture

2.3. Other related work

In the past, many researches related to security features available in different ISA had been carried out. Ning et al analyzed several hardware assisted TEE such as Intel SGX, ARM TrustZone Technology and AMD SEV, and evaluated the feasibility of deploying them on edge compute infrastructure by evaluating the performance overhead [6]. More security and performance benchmark study had been carried out by Christian et al and it has been discovered that AMD SEV has the best benchmark with memory protection mechanism at execution speed of near native speed compared to Intel SGX [7]. Other than performance evaluation, there are also comparative study on multiple ISAs. Geraldine et al summarized some key challenges of security features, short coming of ARM TrustZone and Intel SGX, and proposed countermeasures in RISC-V architecture that address its defined thread models [8]. In respective targeted technology domain, Pascal et al proposed HECTOR-V, a RISC-V based architecture to improve on the flexibilities of peripherals' permission management [5].Victor et al on the other hand, proposed Sanctum, a RISC-V based TEE to improve on software isolation with comparison to Intel SGX [4]. There are also several researches that provide deep dive study and survey on existing technologies, for example, Intel SGX was deeply explained from different aspects from the technology use cases to vulnerabilities by Victor et al [9], an ARM TrustZone comprehensive survey was carried out by Sandro et al [10] and multiple System Management Mode usage model for security purposes was analyzed by William in his PHD report [11].

3. Proposed Method

In this section, secure boot implementation is proposed for RISC-V with firmware implementation in software emulated environment (QEMU) and enhanced with RTL implementation in FPGA hardware environment.

3.1. Secure Boot in Software/QEMU

Figure 14 describes the potential to apply UEFI secure boot to the existing RISC-V boot with the existing UEFI framework on x86 QEMU. The idea is to inject security stack in UEFI PEI and DXE phase so that the RISCV UEFI boot flow can have secure boot encapsulated. This topic was flagged as a potential enabling item by RISC-V presentation [12]. The security stack by UEFI services had been made available with OpenSSL as the underlayer library and a comprehensive technical report of this describing how to sign and incorporate the keys has been created by [13]. The algorithm to verify the hash of a firmware region is as illustrated as Algorithm 1.

3.2. Hardware implementation through FPGA

Another part of the methodology is to propose a method to replace these secure boot services with RTL instead of bootloader code to effectively reduce flash size and improve boot performance. To achieve this, an open-source RISCV processor (NEORV32) is used as an initial environment. SHA512 digest generation block is then added to the RTL of NEORV32, and the digest generated is passed to bootloader via a new read only register block through the custom

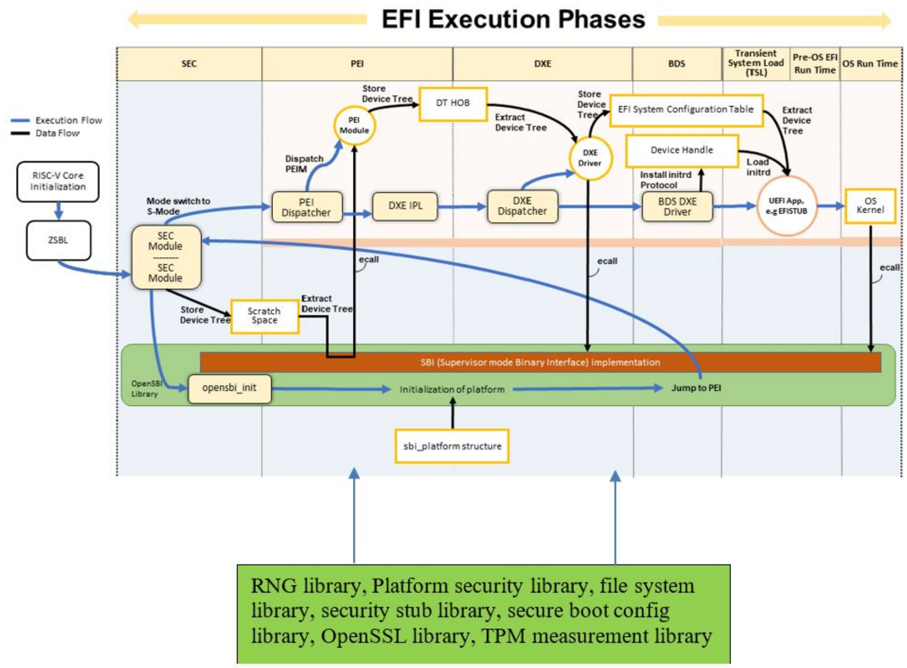

Figure 14: UEFI Secured Boot on RISC-V

functions subsystem (CFS) IP. With this implementation, bootloader will no longer need to contain and execute security code to achieve security purpose. A bird-eye view of what is being added is illustrated in green boxes of Figure 15. The IP are customized to introduce an additional arbitration block with state machine that is capable to map the boot rom content to be sent to SHA512 security block to produce digest. Once digest data is produced, the arbitration block will notify custom functions subsystem block with a status complete bit together with 512-bit SHA information.

Figure 16 demonstrates the detailed comparison of system port map after Arbiter and SHA512 core is being added to NEOV32 CPU processor. The details of how the SHA512 core and arbiter block from signals level and how they are being consumed is described subsequently.

Based on the connections shown in Figure 16, during normal boot up process, NEORV32 CPU fetches instructions from bootloader ROM to execute. The map the ROM code to a memory region;

get the memory address pointer and size;

call SHA512_INIT();

call SHA512_Update() with the pointer and size;

call SHA512_Final() to get the digest.;

compare the digest with a pre-saved digest in hardware root of trust;

if *comparison matches* **then**

  continues the boot process and runs the next firmware code;

else

  halt the boot process due to security violation;

end

Algorithm 1: Flow illustration of secure boot in firmware

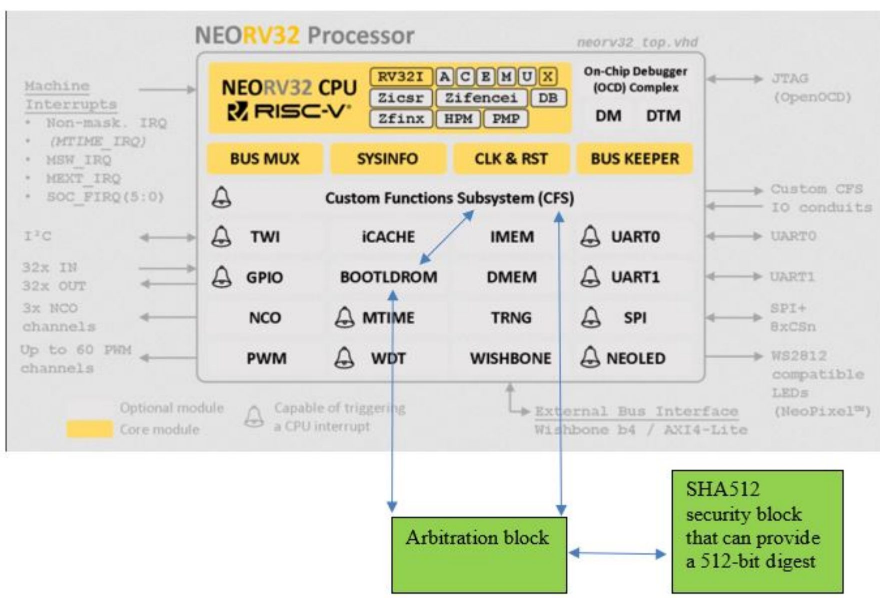

Figure 15: Hardware Secure Boot Block

amount of memory mapped IO and functionalities depend heavily on how the CPU is connected to data bus and in this case, the custom functions system block. The custom functions system block defines an interface consist of offset of each data that bootloader can read and write data from. The SHA512 Core block is responsible for taking in blocks of data to hash, and then provide output of the digest once completed. Arbiter is the middleman which controls the operation of taking ROM data and send to SHA512 Core to be hashed. Once the hash operation is completed, the digest and the completion status will then be shared with custom functions system to be accessible by CPU, which translate to software accessible registers.

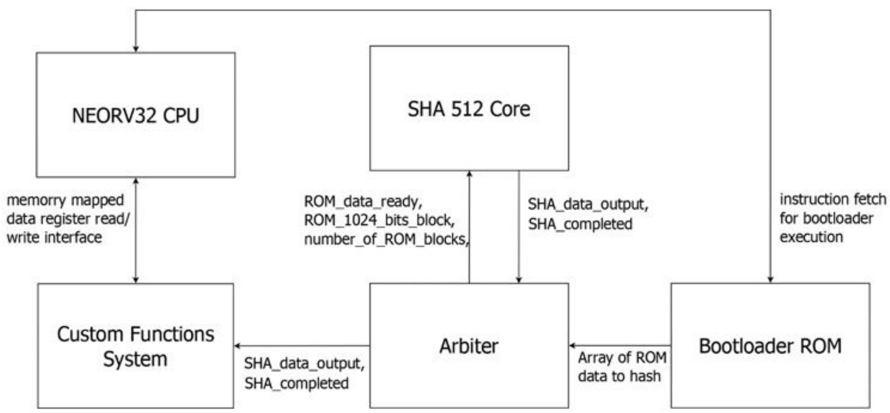

Figure 16: System port map of components to implement additional security block

The state machine of the arbiter is further designed as Figure 17. It begins with Idle state when everything is initialized to 0. A counter is implemented to keep track of the ROM blocks left to transfer from ROM block to SHA512 core. In send data state, the arbiter will set ROM_data_ready for SHA512 block to consume that block of 1024 bits data, then transition to toggle data ready bit state to toggle ROM_data_ready bit to 0 and decrement the counter so that the next send data state will transfer next chunk of data to SHA512 core to be processed. Once all blocks had been sent to SHA512 core. It will wait for SHA512 block to respond with SHA_complete. Once SHA_complete is set to 1, it will enter a complete state and update the SHA data to custom functions system block together with the SHA_complete status bit. The algorithm to shift the firmware ROM in sequent to the SHA512 core is as illustrated as Algorithm

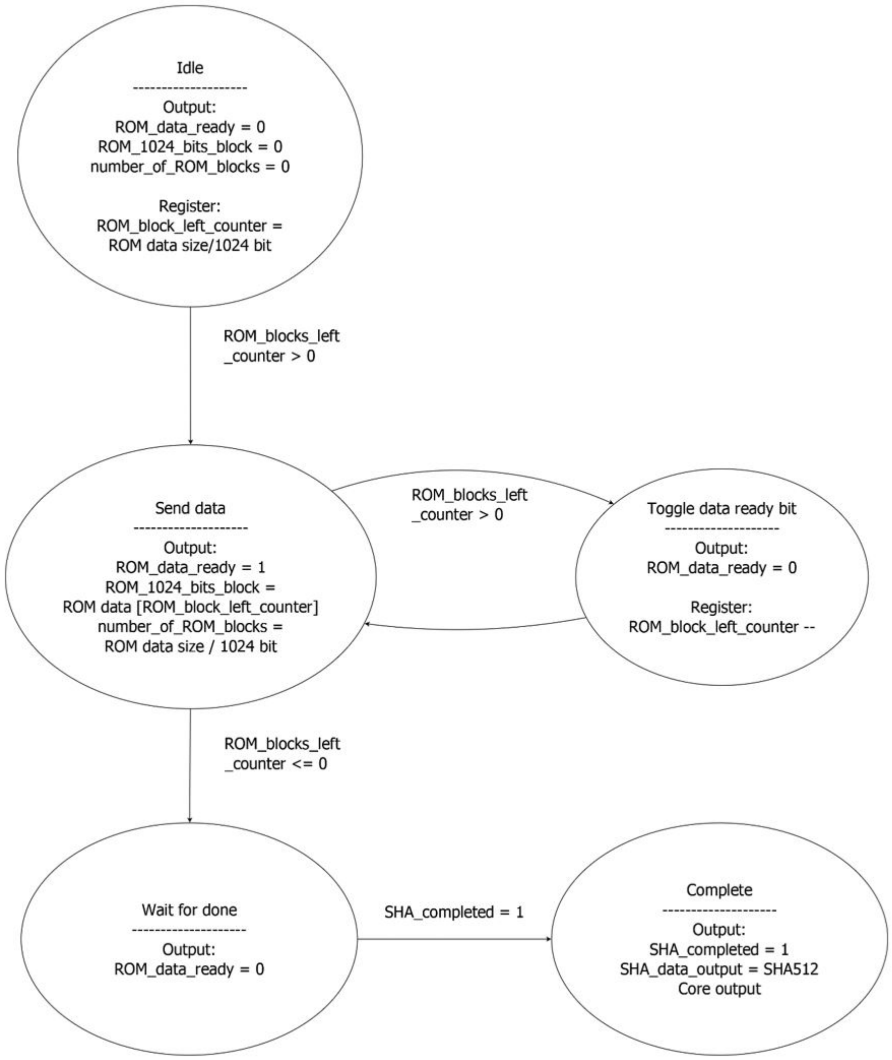

Figure 17: State machine of arbitration block

2.

3.3. Proposed experiments

The experiment proposed to evaluate the methodology includes 3 parameters.

1. Boot performance (time used to boot the firmware)

Data: compiled Bootloader ROM

Result: 512-bit SHA5 digest

while *not the end of bootloader* **do**

  shift current 1024-bit bootloader blocks to SHA5;

  wait for SHA5 block to acknowledge;

  move to next 1024-bit block;

end

Algorithm 2: Simple illustration of arbitration block

2. Firmware size (size of the compiled binary)
3. Number of logic gates consumed in RTL (FPGA resource consumed)

To prove secured boot implementation, negative testing will be performed using unauthorized EFI applications that mimics malicious software to demonstrate that only signed applications can be executed. Therefore, the flow of experiments for software and hardware is planned respectively.

For Software QEMU, a normal unsecured UEFI firmware for x86 and RISC-V is compiled. Then, the UEFI firmware in QEMU is executed by booting to Shell and capturing the boot log. After this is achieved, a secured UEFI firmware is compiled. Then, the secured UEFI firmware in QEMU is executed by booting to Shell and capturing the boot log. With the secured UEFI firmware, negative testing is performeced with signed and unsigned EFI application. The boot time of secured and unsecured firmware are captured. With this and by comparing the results, the impact to boot time after incorporated security can be benchmarked. The binary size of secured and unsecured firmware are captured. With this and by comparing the results, the impact to firmware binary size after incorporated security can be benchmarked.

For Hardware FPGA, the SHA512 block that interacts with other components in NEORV32 RISC-V processor is implemented. The SHA512 digest generated by the SHA512 block is captured and compare with the output of software execution to verify the correctness in functionality. The SHA512 digest generation time is captured to be compared with software execution time to verify the performance. The boot log of NEORV32 firmware is captured to verify the ability to access SHA512 digest generated by RTL in bootloaders.

4. Results/Discussion

From functional correctness perspective, secured boot is configured as Figure 18. With secure boot enabled, only application software that is being signed with the same keys will be able to execute. To validate this behavior, an unsigned "Hello World" application and a signed "Hello World" application is attempted to execute in EFI Shell environment. The commit ID used to produce this result is 392836a for efitools repository and 75e9154f81 for EDK2 repository.

Figure 18: Secure boot configuration

From firmware size perspective, results collected indicates that non-secure UEFI firmware has a total of 7,602,384 bytes, while secure UEFI firmware has a total of 10,895,864 bytes. Therefore, it is deduced that from software perspec-

```
Shell> perf
Loader Performance Info
=======================

 Id   | Time (ms)  | Delta (ms)
------+------------+-----------
 1000 |       7 ms |       7 ms
 1010 |      24 ms |      17 ms
 1040 |      27 ms |       3 ms
 1060 |      41 ms |      14 ms
 1080 |     168 ms |     127 ms
 10A0 |     261 ms |      93 ms
 10B0 |     264 ms |       3 ms
 2000 |     265 ms |       1 ms
 2020 |     885 ms |     620 ms
 2030 |    1021 ms |     136 ms
 2040 |    1169 ms |     148 ms
 2050 |    1228 ms |      59 ms
 2060 |    1239 ms |      11 ms
 2070 |    1240 ms |       1 ms
 2080 |    1281 ms |      41 ms
 2090 |    1287 ms |       6 ms
 20A0 |    1419 ms |     132 ms
 20B0 |    1423 ms |       4 ms
 20C0 |    1424 ms |       1 ms
 20D0 |    1425 ms |       1 ms
 3000 |    1445 ms |      20 ms
 3010 |    1952 ms |     507 ms
 3020 |    1955 ms |       3 ms
 3030 |    2016 ms |      61 ms
 3040 |    2020 ms |       4 ms
 3050 |    2219 ms |     199 ms
 3060 |    2238 ms |      19 ms
 3080 |    2294 ms |      56 ms
 3090 |    2294 ms |       0 ms
 30A0 |    2328 ms |      34 ms
 30B0 |    2328 ms |       0 ms
 30C0 |    2335 ms |       7 ms
 30D0 |    2471 ms |     136 ms
 30E0 |    2478 ms |       7 ms
 3100 |    2502 ms |      24 ms
 3110 |    2530 ms |      28 ms
 3120 |    2550 ms |      20 ms
 3130 |    2716 ms |     166 ms
 3140 |    2826 ms |     110 ms
 3150 |    2826 ms |       0 ms
 31A0 |    2832 ms |       6 ms
 31B0 |    2838 ms |       6 ms
 31F0 |    2839 ms |       1 ms
------+------------+-----------
```

```
Shell> perf
Loader Performance Info
=======================

 Id   | Time (ms)  | Delta (ms)
------+------------+-----------
 1000 |       5 ms |       5 ms
 1010 |      24 ms |      19 ms
 1040 |      27 ms |       3 ms
 1060 |      40 ms |      13 ms
 1080 |     154 ms |     114 ms
 10B0 |     157 ms |       3 ms
 2000 |     158 ms |       1 ms
 2020 |     521 ms |     363 ms
 2030 |     638 ms |     117 ms
 2040 |     798 ms |     160 ms
 2050 |     859 ms |      61 ms
 2060 |     881 ms |      22 ms
 2070 |     881 ms |       0 ms
 2080 |     913 ms |      32 ms
 2090 |     919 ms |       6 ms
 20A0 |     919 ms |       0 ms
 20B0 |     924 ms |       5 ms
 20C0 |     924 ms |       0 ms
 20D0 |     925 ms |       1 ms
 3000 |     970 ms |      45 ms
 3010 |    1347 ms |     377 ms
 3020 |    1350 ms |       3 ms
 3030 |    1460 ms |     110 ms
 3040 |    1462 ms |       2 ms
 3050 |    1644 ms |     182 ms
 3060 |    1668 ms |      24 ms
 3080 |    1713 ms |      45 ms
 3090 |    1713 ms |       0 ms
 30A0 |    1775 ms |      62 ms
 30B0 |    1775 ms |       0 ms
 30C0 |    1785 ms |      10 ms
 30D0 |    1899 ms |     114 ms
 30E0 |    1903 ms |       4 ms
 3100 |    1929 ms |      26 ms
 3110 |    1935 ms |       6 ms
 3120 |    1955 ms |      20 ms
 3130 |    1956 ms |       1 ms
 3140 |    2076 ms |     120 ms
 3150 |    2077 ms |       1 ms
 31A0 |    2083 ms |       6 ms
 31B0 |    2092 ms |       9 ms
 31F0 |    2092 ms |       0 ms
------+------------+-----------
```

Figure 19: Comparison of Secured boot performance with 2839ms boot time against Normal boot with 2092ms

tive, implementing security in firmware will add additional 3.292 Megabytes (10.895M – 7.602M) of additional binary size.

In terms of boot speed, it is observed that non-secure firmware took 2092 milliseconds to boot while secure firmware took 2839 milliseconds to boot in QEMU. Therefore, it is deduced that there are an additional 747 milliseconds (2839ms – 2092ms) additional boot time that secured firmware has in extra, compared to non-secured firmware, which is 747ms/2092ms * 100 = 35.7%.

From hardware perspective, results collected indicates that a RISC-V based NEORV32 without any additional security implementation will consume 19,785 logic gates, while a NEORV32 with the addition of arbiter and SHA512 block consumes 51,833 logic gates. Therefore, it is deduced that implementing security in RTL will add 32,048 logic gates.

In terms of security execution speed comparison, according to Table 2, it is observed that producing a SHA512 digest for 896 bits data will take 257us with software while RTL implementation takes 227ns. The performance advantage is therefore 257u/227n * 100 = 1132%.

Table 1: Hardware and Software Performance Comparison

	Execution Time with 2.2GHz frequency CPU using same set of data
Software execution	257 us
Hardware execution	227 ns

5. Conclusion

The objective of this research, which is to study firmware security schemes, identify and evaluate boot performance with different secure boot scheme, and propose a security enhancement mechanism with open-source ISA, is accomplished. The boot time and boot size impact of implementing hashing for firmware is highlighted in boot performance comparison, which indicates that secured firmware incurred 3.3 Megabytes of additional binary size and 747ms (35%) additional boot time compared to non-secured firmware. The hardware implementation also indicates that it requires an additional 32,048 logic gates to implement a SHA512 IP that reduce software execution time by 1132%.

Although this paper has demonstrated the secure boot implementations with QEMU and FPGA hardware, there are some enhancement to be done to enable security with minimal firmware or software involvement, driven by the initial problem statement. One example is how the configuration to update the RTL security scheme at runtime can be provided for better user experience. A suggestion is through a network IP with manageability mechanism for Over-The-Air (OTA) update, connecting with RISC-V network on chip cores, such as the OpenPiton Network on Chip (NOC) project and have a secure channel to modify the key hashes. This is another topic of research area that can be proposed and presented with industrial use cases with business opportunities to introduce such features on IOT secured devices.

Acknowledgment

The authors would like to thank USM and Intel for the support that leads to the completion of this work.

References

References

[1] J. Turley, Wait, what? mips becomes risc-v, `https://www.eejournal.com/article/wait-what-mips-becomes-risc-v/`, accessed on 2021-8-30 (March 2021).

[2] A. S. Jagan Teki, An introduction to risc-v boot flow: Overview, blob vs blobfree standards, China RISC-V Forum.

[3] C. Abner, W. Dong, Uefi and risc-v, 3rd RISC-V Workshop.

[4] V. Costan., I. Lebedev., S. Devadas., M. CSAIL, Sanctum: Minimal hardware extensions for strong software isolation, USENIX The Advanced Computing System Association.

[5] P. Nasahl., R. Schilling., M. Werner., S. Mangard, Hector-v: A heterogeneous cpu architecture for a secure risc-v execution environment, 2021 ACM Asia Conference on Computer and Communications Security.

[6] Z. Ning, J. Liao, F. Zhang, W. Shi, Preliminary study of trusted execution environments on heterogeneous edge platforms, Third ACM/IEEE Symposium on Edge Computing.

[7] C. Göttel., R. Pires., I. Rocha., S. Vaucher., P. Felber., M. P. V. Schiavoni, Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms, 2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS), Salvador, Brazil.

[8] N. G. Shirley., G. Yutian., S. Fareena, A survey and analysis on soc platform security in arm, intel and risc-v architecture, 2020 IEEE 63rd International Midwest Symposium on Circuits and Systems (MWSCAS).

[9] V. Costan., S. Devadas, Intel sgx explained, Computer Science and Artificial Intelligence Laboratory Massachusetts Institute of Technology.

[10] S. Pinti., N. Santos, Demystifying arm trustzone: A comprehensive survey, ACM Computing Survey Volume 51, Issue 6.

[11] d. S. William Augusto Rodrigues, On using the system management mode for security purpose, University of London, Doctor of Philosophy.

[12] D. Schaefer, Edk2 uefi on risc-v, https://fosdem.org/2021/schedule/event/firmware_uor/attachments/slides/4492/export/events/attachments/firmware_uor/slides/4492/EDK2_on_RISC_V.pdf, accessed on 2021-8-30 (February 2021).

[13] N. S. Agency, Uefi secure boot customization, https://media.defense.gov/2020/Sep/15/2002497594/-1/-1/0/CTR-UEFI-SECURE-BOOT-CUSTOMIZATION-20200915.PDF/CTR-UEFI-SECURE-BOOT-CUSTOMIZATION-20200915.PDF, accessed on 2021-8-30 (September 2020).