Parallel Verification in RISC-V Secure Boot

Akihiro Saiki
Waseda University
Tokyo, Japan
saiki@kasahara.cs.waseda.ac.jp

Yu Omori
Waseda University
Tokyo, Japan
oy@kasahara.cs.waseda.ac.jp

Keiji Kimura
Waseda University
Tokyo, Japan
keiji@waseda.jp

***Abstract*—Assuring the reliability of the OS boot process is essential to realize reliable computer systems. Secure Boot enables it by introducing the verification of the boot image with digital signature and hash values. This can be the basis for various security mechanisms. However, Secure Boot requires a long boot time due to their expensive computation costs, resulting in extended downtime. In this paper, we first implement Secure Boot in an ordinary RISC-V boot process on U-Boot, a representative open-source bootloader, and clarify the overhead introduced by the verification process. Based on the insight obtained above, we propose a parallelization of the verification process on a multi-core in Secure Boot. It accelerates the boot process while securely authenticating the boot image. We implement the proposed parallel verification process on U-Boot. The evaluation on a HiFive Unmatched RISC-V board shows that the parallelized hash computation on four cores achieves 3.96 times better performance than the original.**

***Index Terms*—Parallelization, Multi-Core Processing, RISC-V, Secure Boot**

I. Introduction

With the spread of Internet of Things (IoT) technology and edge computing, security on edge devices and near-edge resources becomes important [1]–[3]. In edge computing-based IoT systems, gateways and computation nodes are located near the end users in addition to end devices [4]. Unlike centrally controlled systems like cloud servers, devices and servers near the end users may not be tightly managed. Therefore, they often have security issues, such as insecure placement, either physically or in terms of network, and unaddressed vulnerabilities in the system [5], [6]. In such cases, adversaries can attack not only through remote access but also through physical access to the devices easily. This means it is easy to take control of the system by tampering with the firmware or the OS kernel on the off-chip memory. Nevertheless, devices and servers at the edge may also handle sensitive data, such as machine learning models that contain personal information. Sensitive data must be processed securely, and devices and servers should have a mechanism to protect the computation process.

Secure Boot provides a security mechanism that assures the reliability of a system boot process and can be a countermeasure against tampering attacks on firmware and an OS kernel. Moreover, some security mechanisms, such as remote attestation for code integrity, require Secure Boot as a trusted basis [7]–[9]. In general, a system boots through multiple boot stages, including initializing hardware and peripherals, loading firmware, and executing the bootloader. Therefore, Secure Boot should ensure the reliability of the entire boot stages. It verifies the boot images of each stage by hash value and digital signature at the booting time. Its sequence begins with the trusted root, called Root of Trust (RoT). RoT is the trusted, unmodifiable data like a verification key for signature verification. Based on the data in the RoT, each boot stage verifies the subsequent boot image in a chain, constructing the Chain of Trust (CoT). The CoT provides proof of the reliability of the entire boot process.

There are various implementations of Secure Boot [10]. For security reasons, RoT should be hardware-protected and immutable. Therefore, in a typical implementation, the reliable part of the system, such as On-chip ROM, is often selected as RoT, and the subsequent process is configured in hardware or software.

Secure Boot schemes based on hardware typically use additional hardware resources or hardware-dependent functions, such as cryptographic engines, Trusted Platform Module (TPM), and so on [11]–[13]. The system boot state may be controlled by hardware. Hardware-based schemes are more secure and lightweight in terms of computation than software-based schemes. However, these schemes have restrictions on implementation targets and difficulties in modification.

Regarding the software-based schemes, the verification process is typically implemented in the bootloader [14]. Each boot stage in the boot process loads and verifies each boot image. However, expensive computation costs for signatures and hash values introduce significant boot overhead in such schemes. Additional hardware modules, such as cryptographic engines and accelerators, have sometimes been used with the software to mitigate this overhead.

In this paper, we focus on software-based schemes. We propose a parallelization method of the Secure Boot verification process as another acceleration technique. An ordinary hash calculation used in Secure Boot has difficulty in parallelization since it operates through the whole boot image from the beginning to the end in order, resulting in data dependency among operations. On the other hand, our approach divides the boot image into multiple chunks and calculates the hash value for each chunk with its location id to ensure the integrity of the whole image. The final hash value is calculated from the hash values of the chunks. Thus, multiple chunks can be calculated in parallel. The verification process in the boot time uses the newly calculated hash value. We implemented the parallelized Secure Boot to U-Boot, a representative open-

2771-3075/23/$31.00 ©2023 IEEE
DOI 10.1109/MCSoC60832.2023.00089

source bootloader [15]. Then, we evaluated the parallelization performance on a HiFive Unmatched RISC-V development board with four application cores [16].

The contributions of this paper are summarized as follows:

- We implement Secure Boot on U-Boot for RISC-V systems and clarify its overhead.
- We propose a parallelization method of hash calculation in the Secure Boot process.
- We implement the proposed parallel hash calculation in U-Boot.
- We evaluate the proposed parallel boot image verification technique on a HiFive Unmatched RISC-V board.

The rest of the paper is organized as follows: Section II reviews related works about implementations of Secure Boot, then Section III states the threat model, and Section IV presents the measurement result that motivates this paper. Section V proposes the parallelization method of verification. Section VI explains the implementation of parallelized Secure Boot to the bootloader. Section VII presents the performance evaluation results. Finally, Section VIII concludes this paper.

II. Related Works

Implementations of Secure Boot can fall into two types: hardware-based and software-based [17]. There are also hybrid methods that introduce both of them in the verification.

Hardware-based Secure Boot methods for RISC-V SoC were proposed in [11], [12]. They reduce the Secure Boot time by introducing hardware cryptographic modules to calculate hash values and signatures. While they are more lightweight in computation and robust in terms of security, they also have many hardware limitations.

Hardware TPM is also used as a hardware-based implementation [13]. TPM is a hardware co-processor that can be used as RoT and has a strictly protected register called Platform Configuration Register (PCR) [18]. The PCR value can be updated only with the hash value of the combination of the current PCR value and the input value and cannot be deleted and replaced. This can build the CoT and assure the reliability of the entire boot process. However, the overheads of the TPM initialization and the PCR operation can be highly significant, delaying the entire boot process [13].

A Hybrid Secure Boot method, which uses hardware-specific features and software-based verification, is proposed in [8]. This method uses ARM TrustZone, the isolation technology implemented in the ARM processors [19], to protect the verification process and provide Trusted Execution Environment (TEE). Secure Boot proceeds on the Secure World (SW), a physically isolated secure execution environment. In addition, Secure Boot is used as a trusted base for remote attestation of the OS and the filesystem image used in Normal World (NW). Although TrustZone makes this approach more secure than software-only implementations, the performance problems are remaining because the verification of Secure Boot is still a software implementation.

The wolfBoot is a bootloader that implements software-based verification [14]. It provides a simple verification process for boot images and firmware updates. The main target of the wolfBoot is microcontrollers (MCUs), and it uses wolfSSL, a lightweight cryptography library for embedded systems [20]. Such software-based Secure Boot can be applied by simply installing the bootloader, making it easy to deploy on several devices. However, the verification can introduce a considerable boot overhead without hardware acceleration. In this paper, we make a simple implementation of verification based on the wolfBoot in U-Boot and accelerate the software-level verification by parallelization. Note that though the proposed method is software-based, it is orthogonal to the hardware-based methods. We can expect further performance by combining it with the hardware cryptographic modules.

III. Threat Model

This paper assumes that edge devices and near-edge resources, such as IoT devices, edge servers, and IoT gateways, are the target of attacks. The assumed attacks are the tampering attacks targeting firmware and OS kernel in the following two cases. In both cases, the component selected as the RoT is assumed to be immutable and implicitly trusted.

In the first case, adversaries have physical access to the target device. They can physically access non-volatile storage for firmware, tamper with it, and install malicious programs. This attack is expected to be performed while the device is down or before the device is deployed.

In the second case, adversaries can access the targets through the network. If the target system has potential security vulnerabilities, such as Remote Code Execution (RCE) and Privilege Escalation, adversaries can exploit these vulnerabilities remotely via the network. The exploitation enables adversaries to execute malicious code for tampering with firmware. These devices may also have the function to update the firmware over the network. If this feature is not implemented securely, adversaries can use a malicious image for update [21], [22].

Secure Boot assures integrity and authenticity of binary images at a booting time, thus cannot protect from tampering with unprotected memory sections at run-time. To protect against tampering at the application's run-time, we can use TEE such as RISC-V Keystone [23] and the methods that provide Remote Attestation (RA) [7]. In this paper, we use the OpenSBI [24], which implemented the security monitor of RISC-V Keystone though we do not use Keystone here. Therefore, the protection of tampering at run-time is out of the scope.

IV. Motivation

The computation cost of software-based verification is expensive, resulting in an extended downtime at the system reboot. This becomes a critical problem for systems that require high availability. To clarify the actual effects of the verification during Secure Boot, we implemented a simple image verification process in U-Boot by referring wolfBoot [14], then evaluated it on the HiFive Unmatched board [16]. The detail of this board is described in Section VI.

TABLE I
PERFORMANCE EVALUATION OF SECURE BOOT

Boot Stage	Verification [ms]	Stage total [ms]	Ratio [%]
U-Boot SPL	1286	3051	42.14
OpenSBI	-	211.0	-
U-Boot Proper	4492	6625	67.81
Total	5778	9888	58.44

Originally, the HiFive Unmatched board boots Linux through four boot stages: Zeroth-Stage Bootloader (ZSBL), U-Boot SPL, OpenSBI [24], and U-Boot Proper. Among these boot stages, ZSBL, U-Boot SPL, and U-Boot Proper act as bootloaders to load the images, and OpenSBI is an open-source implementation of RISC-V Supervisor Binary Interface (SBI) [25]. ZSBL has special features: it cannot be modified and runs at an entirely slow clock frequency. Therefore, ZSBL is outside this evaluation's scope, and the Secure Boot process begins at U-Boot SPL. The binary images loaded by each bootloader are as follows:

- U-Boot SPL
  - OpenSBI
  - U-Boot Proper
  - Flattened Device Tree (FDT) [26]
- U-Boot Proper
  - Initial RAM Filesystem (initramfs)
  - Linux Kernel
  - FDT

The original boot process does not have Secure Boot or equivalent functionality. Thus, we introduce Secure Boot to the HiFive Unmatched by implementing the verification process in U-Boot.

We measure the time taken for verification and the total processing time for each boot stage. The time taken for each detailed verification process is also measured. The booting target is Ubuntu Server 22.04. We conducted the measurements five times and took the average times. Details of the evaluation environment are shown in Section VII.

The results shown in Table I indicate that the bootloader's verification process can be an overhead. In particular, approximately 4.5 seconds are consumed for verification with the U-Boot Proper, which is a significant factor in increased boot time. Furthermore, Table II shows that the hash calculations for *initramfs* and *kernel* are the most significant cause of overhead of the U-Boot Proper and the overall boot process (underlined in the table). One of the direct factors of that is the large size of these images. The *initramfs* is 97MiB zstd compressed image, and the *kernel* is 29MiB bzimage.

According to these results, we consider accelerating hash calculation performed in the Secure Boot. On a multi-core machine, parallelization is a fundamental method to accelerate computation at the software level. However, the original implementation of the hash calculation cannot be parallelized as it is. Therefore, we propose a parallelizable hash calculation method presented in Section V.

TABLE II
PERFORMANCE DETAILS OF EACH VERIFICATION PROCESS

Loader	Load Target	Calculation	Time [ms]
U-Boot SPL	OpenSBI	Hash	29.16
		Signature	384.4
	U-Boot	Hash	100.7
		Signature	384.4
	FDT	Hash	2.840
		Signature	384.4
U-Boot Proper	initramfs	Hash	<u>3268</u>
		Signature	90.38
	kernel	Hash	<u>952.8</u>
		Signature	90.31
	FDT	Hash	0.3392
		Signature	90.34

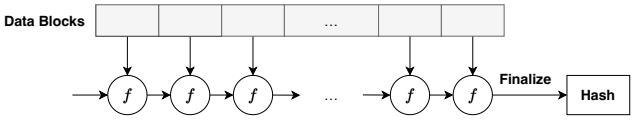

Fig. 1. Model of general cryptographic hash function

V. PARALLELIZATION APPROACH

In this section, we describe our approach to parallel hash calculation. We parallelize it by dividing the entire data into multiple chunks. Then, we calculate a sub-hash value for each chunk and obtain the total hash value by accumulating sub-hash values. This is a variant of Merkle tree-based calculation.

A. Ordinary Hash Calculation

Typically, a hash value is calculated by inputting the entire file directly into a hash function. This means that a hash value calculation of a file accumulates data by sequentially accessing the file from the top to the end (Figure 1). While this calculation order of the hash value ensures file integrity, it introduces data dependency among operations. Thus, the original hash calculation cannot be parallelized without changing its calculation method. To parallelize a hash calculation, we need a method that allows data decomposition and independent decomposed data calculation while ensuring data integrity. Therefore, we focused on Merkle tree, which can verify the whole object from data split into chunks using a tree structure [27].

B. Merkle Tree

Merkle tree [27] is a hash-based tree structure mainly used to verify the integrity of distributed data efficiently. It is usually constructed with a binary tree or an n-ary tree. In a Merkle tree, the whole data is split into chunks. A tree has leaves, each of which has a hash value obtained from a chunk of the data. Similarly, a non-leaf node in the tree has a hash value obtained from its children nodes. The value of the top node of the tree is called *Root Hash*, representing the entire data. Only *Root Hash* must be stored securely for verification.

Figure 2 depicts an example structure of a Merkle tree constructed as a binary tree with eight leaves. In this figure,

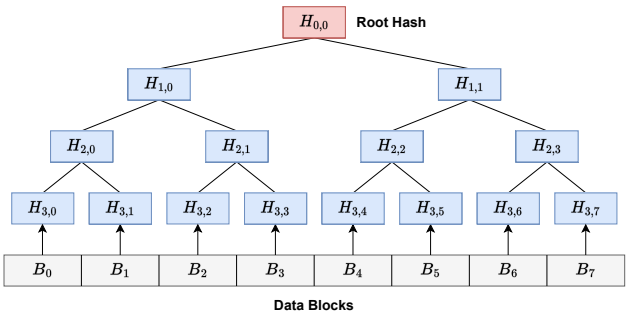

Fig. 2. Merkle hash tree constructed from data blocks

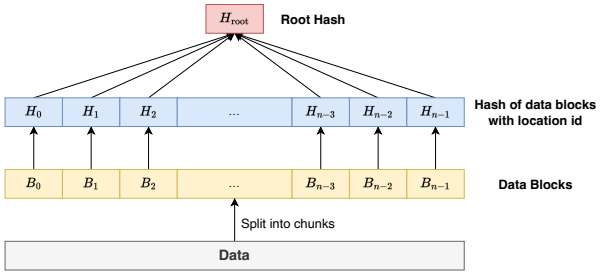

Fig. 3. Hashing scheme using one-level n-ary tree

$H_{i,j}$ represents a hash value of the j-th node at the i-th level. A leaf node at the bottom of the tree has a hash value, $H_{3,j}$, obtained from the corresponding data block B_j. For the other nodes than leaves, the hash value $H_{i,j}$ is calculated with the hash function h as follows:

$$H_{i,j} = h(H_{i+1,2j}||H_{i+1,2j+1})$$
$$(|| : \text{string combination})$$

If the values of all children nodes are known, the node's value can be calculated independently from other nodes at the same level. Finally, *Root Hash*, the value of the tree's root, is calculated from its leaves.

When one data block is modified, it changes the value of the parent node, and the change propagates up to the tree's root through intermediate nodes. Thus, verifying a partial modification of the data can be done by properly tracking back to the root node. This is less expensive than recalculating the hash value of the entire data and is one of the advantages of utilizing the Merkle tree for integrity verification. This feature is particularly effective in distributed systems.

Merkle tree is built from a chain of hash functions, and its security proof is equivalent to that of the hash function itself [28]. Therefore, its security strength depends on the hash function used.

For parallel processing of hash calculation, it is important that the original data is split into chunks and their hash values are independently calculated each other. This resolves the data-wide dependence of ordinary hash calculation.

C. Our Approach

As described above, Merkle Tree enables parallel hash calculation by dividing the input data into multiple chunks and lightweight hash recalculation at a partial data update by constructing a hash tree. The parallelization of a hash calculation at Secure Boot does not require hash recalculation since it must always verify the entire binary image every boot time. It just needs parallelism at the bottom of the tree instead of constructing a multi-level tree. Therefore, we introduce a one-level n-ary tree for parallel hash calculation in our Secure Boot as depicted in Figure 3.

This scheme constructs only a one-level subtree of an n-ary Merkle tree, where n is the number of data blocks. First, the original file image is split into smaller data blocks. Then, the hash values of the data blocks are calculated in parallel. They become the leaf nodes of the tree. When an image is split into blocks, it is possible that the hash values from different blocks coincidentally become the same, for example, at the padding. Hash values in different blocks should not collide from a security perspective; thus, the index number of the blocks is added to the contents as a location identifier before hashing. Finally, the *Root Hash* is calculated sequentially after calculating all leaf nodes. In the *Root Hash* calculation, the input size is a multiplication of the number of blocks and the length of the hash digest. Since hash digests are short, fixed-length values, the computational complexity of *Root Hash* is less than that of the ordinary sequential processing of the image. Denoting the content of the i-th block as B_i, the identifier of i-th block as id_i, the hash value of the i-th block as H_i, and the *Root Hash* as H_{root}, the entire calculation with the hash function h is expressed as follows:

$$H_i = h(id_i||B_i)\ (0 \leq i \leq n-1) \quad (1)$$
$$H_{root} = h(H_0||H_1||\cdots||H_{n-2}||H_{n-1}) \quad (2)$$

The hash value calculated by the proposed method differs from the sequentially calculated original one. It also takes different values depending on the size of the blocks. Therefore, after determining the size of data blocks for the target image to be divided, the hash value and signature of the image by this hashing scheme must be pre-computed.

It is worth mentioning that the security strength of the proposed parallel hash calculation is the same as that of Merkle Tree since it is a one-level Merkle Tree.

VI. Implementation

A. Overview

We implement the proposed Parallel Secure Boot for a HiFive Unmatched RISC-V board. Table III shows the specification of the HiFive Unmatched board. It has a Freedom U740 SoC with a SiFive S7 embedded core and four SiFive U74 application cores.

TABLE III
SPECIFICATION OF HIFIVE UNMATCHED

Board	SiFive HiFive Unamtched	
SoC	SiFive Freedom U740	
CPU Core	1 x S7 Core	4 x U74 Core
ISA	RV64IMAC	RV64GC(IMAFDC)
Privileged Mode [29]	M-Mode U-Mode	M-Mode S-Mode U-Mode
Frequency	1.2GHz (26MHz before PLL init)	
L1-I Cache	16KiB/core 2-way	32KiB/core 4-way
L1-D Cache	N/A	32KiB/core 8-way
DTIM	8KiB	N/A
L2 Cache	2MiB 16-way	
DRAM	DDR4 16GiB	

As mentioned in Section IV, the Linux boot sequence of a HiFive Unmatched board consists of four stages: ZSBL, U-Boot SPL, OpenSBI, and U-Boot Proper. We first implement the Secure Boot verification process in this boot sequence by referring wolfBoot. Then, we parallelize a part of the verification process.

B. Secure Boot

The overall structure of the Secure Boot implemented on the HiFive Unmatched board is depicted in Figure 4. The implementation details are described below from three points of view: Root of Trust, Chain of Trust, and Verification Process.

1) Root of Trust (RoT): As mentioned in Section IV, the first bootloader ZSBL is excluded from this Secure Boot process, and the process begins at U-Boot SPL, the second bootloader. Therefore, we assume that U-Boot SPL is stored in unmodifiable memory, such as on-chip ROM, and it is the RoT. We implicitly trust U-Boot SPL binary and its FDT, including the public key for verification, and establish the CoT based on it.

U-Boot SPL is stored in external storage in the evaluation environment because of its hardware specification. The main objective of this paper is performance evaluation. Thus we do not pursue security strictness, assuming that the U-Boot SPL is trusted.

2) Chain of Trust (CoT): According to the assumption of the RoT above, the CoT can be established from U-Boot SPL. As depicted in Figure 4, the prior loader verifies and executes the successor images in order if each verification passes. The boot sequence is immediately stopped when the verification fails, and the invalid program execution is prevented. All target images to be verified must have hash values and signatures. Such information used in verification is appended as an image header ("Sig." in Figure 4). For signature verification, each bootloader has a public key. The key is installed in the FDT, which is implicitly trusted or verified before it is used in the Secure Boot sequence.

3) Verification Process: The overall signing and verification flow is depicted in Figure 5. Before deploying the image to the device, the target image must be signed with the generated key

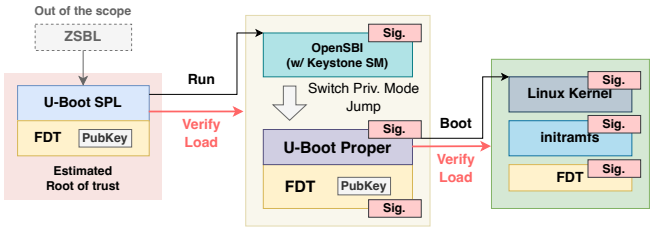

Fig. 4. Structure of Secure Boot

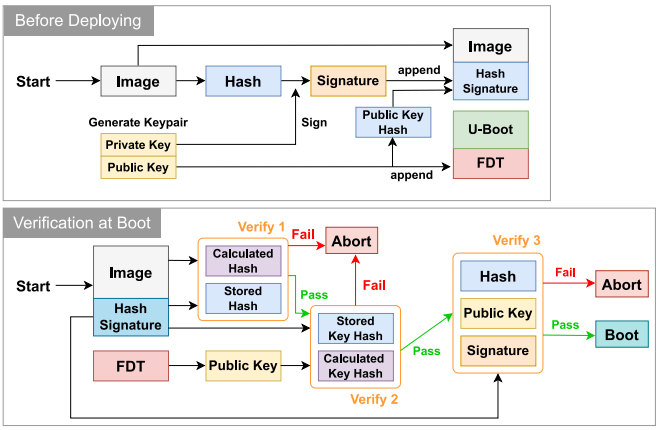

Fig. 5. Image signing and verification flow

pair. The private key is used for signing, and the public key is stored in the FDT of each bootloader. The pre-calculated hashes and signatures of the binary images are attached to the images as image headers. The hash value of the public key is also attached to the image to detect the appropriate key for verification.

The verification at boot time proceeds as follows:

1) Calculate and verify the hash value of the image
2) Calculate and verify the hash value of the public key
3) Verify the signature

The image header contains information related to the boot process and the verification other than hash values and signatures. The hash value is calculated to protect them, including a part of the image header. Denoting the hash value of the image header as H_{hdr}, Equation (2) becomes

$$H_{root} = h(H_{hdr}||H_0||H_1||\cdots||H_{n-2}||H_{n-1}). \quad (3)$$

C. Parallel Processing

In this Secure Boot implementation, hash calculation is performed in U-Boot SPL and U-Boot Proper. As described in Section IV, the overhead from the hash calculation of the initramfs and the Linux kernel, the verification targets of U-Boot Proper, is significant. Therefore, in this paper, parallelization is implemented to be performed in U-Boot Proper.

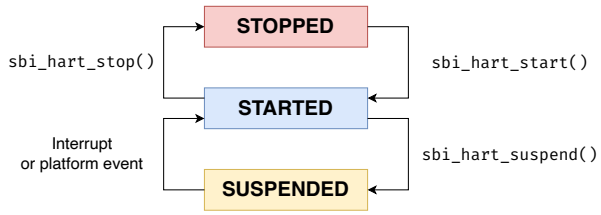

Fig. 6. State diagram of the HSM Extension omitting pending states [25]

D. Harts Control

U-Boot Proper runs under S-Mode, one of the RISC-V Privileged Mode [29]. In S-Mode, all hardware threads (harts) are controlled by the firmware that implements SBI. An S-Mode application can control harts via SBI Hart State Management (HSM) Extension [25]. The HSM Extension introduces a set of hart states and provides SBI ecall functions that enable S-Mode applications to control hart states. The hart states consist of three main states: STARTED, STOPPED, and SUSPENDED, and four pending states between each of them. Figure 6 shows the state transitions of the HSM Extension without pending states. When OpenSBI switches the privileged mode to S-Mode and jumps to U-Boot Proper, only one hart, called boot hart, is in the STARTED state, and the others are in the STOPPED state. The boot hart proceeds the boot process alone and wakes up other harts after initialization in the Linux kernel. Non-boot harts must be woken up before Linux booting and set stack pointer to point to the allocated stack area to use them for parallel processing in U-Boot. Therefore, U-Boot Proper needs to wake up non-boot harts to initialize the stack for each hart and then put non-boot harts on standby.

U-Boot does not have functions to control harts using HSM Extension because parallel processing is generally not performed in the bootloader. In addition, Linux kernel 5.6 or later is fully compatible with HSM Extension and does not work correctly if the non-boot harts are not in the STOPPED state at the kernel booting time. Considering the above, we modify U-Boot to realize parallel processing on multiple harts as described below.

Figure 7 depicts the harts control flow implemented in U-Boot Proper. In the initialization phase, the boot hart calls the ecall function `sbi_hart_start()` to wake up the non-boot harts, and each hart initializes its stack region. Then, the non-boot harts wait for interrupts with the `wfi` instruction. This enables the boot hart to control the non-boot harts utilizing Inter-Processor Interruption (IPI). The S-Mode applications can generate IPI calling the ecall `sbi_send_ipi()`. Upon receiving IPI, the non-boot hart executes the given task and waits for another interrupt. Before booting Linux, the boot hart sends IPI, and the non-boot harts call the ecall `sbi_hart_stop()` to become the STOPPED state.

E. Parallel Task Control

In parallelized block hashing, the hash value calculation for each block is defined as a parallel task. Since there is

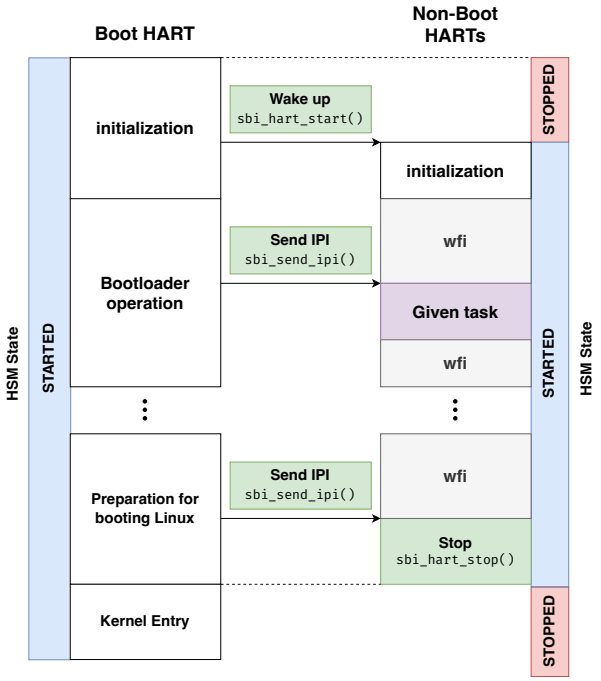

Fig. 7. Harts control flow using HSM Extension

no significant difference in the calculation of each block, the number of blocks handled by each thread is allocated evenly.

Task execution control is implemented simply. The boot hart acts as a master, and non-boot harts act as workers. The master allocates tasks to workers, executes its tasks, and then monitors whether the workers have finished the tasks. Task allocation is performed by utilizing IPI; the master stores task information in RAM and generates IPI to the workers. It is impossible to pass any arguments via IPI, and the location of task information must be pre-defined. When the workers receive interrupts, they execute the IPI handler, designed to collect task information and jump to tasks. After the worker complete tasks, they mark the flags indicating the status of their tasks located pre-defined position in RAM as completed. The master monitors these flags and waits to calculate *Root Hash* until all threads finish their tasks. These flags read/write instructions are performed atomically, using `fence` instructions.

Figure 8 depicts an example of 4-harts task control flow. Before hash calculation, the master prepares parallel task information to pass to the workers, and all the workers are waiting for interruption by executing `wfi` instruction. After the master sends IPI to the workers and the workers receive it, hash calculation is started on all harts. In this example, for N data blocks $B_0 \dots B_{N-1}$, the master and worker 1-3 perform hash calculations of $B_0 \dots B_{i-1}$, $B_i \dots B_{j-1}$, $B_j \dots B_{k-1}$, $B_k \dots B_{N-1}$, respectively $(0 < i < j < k < N)$. The master waits until all workers have finished their task, and the workers wait for another interruption again. Finally, the master calculates *Root Hash* and verifies the signature.

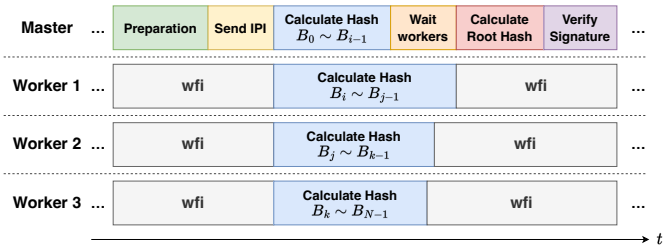

Fig. 8. Example of 4-threads parallel hash calculation flow

TABLE IV
SOFTWARE USED FOR EVALUATION

Firmware	OpenSBI v1.1 (w/Keystone SM)
Bootloader	U-Boot v2022.07 (Modified)
Crypto Lib.	wolfSSL v5.5.0-stable
Algorithm (Hash/Signature)	SHA3-384 / ED25519
Boot Target	Ubuntu Server 22.04 Linux 5.15.0
OS for pre-evaluation	Freedom U SDK 2022.12.00 Linux 5.19.14

VII. EXPERIMENTAL EVALUATION

In this section, we first describe the preliminary evaluation of the parallelized hash calculation. Then, we demonstrate the performance of the parallelized Secure Boot. Table IV shows a set of software used for the evaluation. SHA3-384 and ED25519 are selected as hash and signature algorithms, respectively. Parallelization is configured to use four U74 application cores shown in Table III, and initramfs and Linux kernel are selected as parallelization target images. All results are the averages of five times measurements.

A. Preliminary Evaluation

First, we implement the parallelized hash calculation as a Linux user application to determine the appropriate size of the data blocks (block size). We evaluate the execution time for hash calculation, varying the block size from 1KiB to 160KiB.

Table V shows the measurement results of our parallelized hash calculation. The top row is the result of the original implementation for sequential calculation. The computation time decreases as block size increases, and almost converges around 80KiB (initramfs: 885 ms, kernel: 260 ms). Compared to the original sequential one, the computation times of initramfs and kernel are 3.96 and 3.92 times faster, respectively, in the 80KiB blocks (initramfs: 3503 ms $\rightarrow$ 885.0 ms / kernel: 1018 ms $\rightarrow$ 259.9 ms).

Although we have no clear criteria for determining block size, several observations exist. The improvement of computation time between larger blocks is negligible. In addition, larger blocks can significantly differ in computational complexity for a single block, which can worsen parallelization efficiency. However, smaller blocks can increase memory consumption. In this implementation, the hash values of all blocks are stored in the memory until the *Root Hash* is calculated. We need to consider the available resources and computing performance to choose an appropriate block size.

TABLE V
PERFORMANCE EVALUATION OF BLOCK HASHING

Block Size [KiB]	Time [ms]			
	initramfs		kernel	
	Sequential	Parallel	Sequential	Parallel
-	3503	-	1018	-
1	4335	1281	1262	374.1
2	3935	1086	1145	317.9
4	3659	968.9	1065	284.2
8	3601	931.3	1048	272.6
16	3530	903.2	1027	264.8
32	3516	894.5	1023	261.7
64	3502	889.9	1019	260.5
80	3500	885.0	1019	259.9
96	3497	888.0	1018	261.3
112	3497	884.3	1018	260.0
128	3495	889.0	1018	261.2
144	3495	886.5	1017	261.8
160	3493	881.4	1017	262.4

B. Parallelized Secure Boot

Based on the preliminary evaluation results above, we set the block size to 80KiB and measure the entire parallelized Secure Boot process.

The performance evaluation in the bootloader is performed using Hardware Performance Monitor (HPM) [30]. The HPM has Control and Status Register (CSR) that counts executed cycles in each hart. We can measure elapsed cycle simply by reading `mcycle` (M-Mode) register and `cycle` (other privilege modes) register. After measuring elapsed cycles, the elapsed time can be calculated from the clock frequency (1.2GHz). The measurement is performed on the boot hart, which monitors the worker harts.

Table VI shows the measurement result of the overall boot stages, and Table VII shows the measurement result of each verification process. Comparing Table VI with Table I, the total verification time becomes 2.24 times faster (5778 ms $\rightarrow$ 2625 ms), and the overall time taken for boot becomes 1.41 times faster (9888 ms $\rightarrow$ 6987 ms). The ratio of the verification process in the boot sequence has also been reduced to 37.57% from 58.44%, which means that the overhead has been reduced.

Concerning the verification details, both the initramfs and kernel hash calculation, which applied parallelization, are less than a second (underlined in Table VII). This indicates a speedup of 3.96$\times$ (3268 ms $\rightarrow$ 824.9 ms) and 3.92$\times$ (952.8 ms $\rightarrow$ 242.8 ms), respectively, compared to the simple sequential implementation measured before. The acceleration ratios are almost equal to the results obtained in the preliminary evaluation and are also close to the number of threads (four). This means that the effect of parallelization is sufficient, as the impact of the sequential *Root Hash* calculation and other overhead is minimal.

VIII. CONCLUSION

In this paper, we have proposed a parallelization method of hash calculation in Secure Boot and implemented the method on a HiFive Unmatched RISC-V board with four application

TABLE VI
PERFORMANCE EVALUATION OF PARALLELIZED SECURE BOOT

Boot Stage	Verification [ms]	Stage total [ms]	Ratio [%]
U-Boot SPL	1286	3024	42.52
OpenSBI	-	211.3	-
U-Boot Proper	1339	3752	35.69
Total	2625	6987	37.57

TABLE VII
PERFORMANCE DETAILS OF PARALLELIZED VERIFICATION PROCESS

Loader	Load Target	Calculation	Time [ms]
U-Boot SPL	OpenSBI	Hash	29.16
		Signature	384.3
	U-Boot	Hash	100.8
		Signature	384.4
	FDT	Hash	2.840
		Signature	384.5
U-Boot Proper	initramfs	Hash(Parallel)	824.9
		Signature	90.27
	Kernel	Hash(Parallel)	242.8
		Signature	90.43
	FDT	Hash	0.3405
		Signature	90.25

cores. This method divides the target images into multiple blocks, each of which has an independent hash value. The hash calculation for each divided block includes its id number and body to ensure the integrity of the target image. This makes it possible to calculate the hash value of each block in parallel.

The evaluation of four-thread parallelization on a HiFive Unmatched board confirmed that the hash calculation could be up to 3.96 times faster, and the overall boot process could be 1.41 times faster than the original implementation.

IX. ACKNOWLEDGEMENT

We would like to thank Professor Tatsuya Mori and Professor Kazue Sako for the valuable discussion about parallel hash calculation. A part of this paper is supported by JSPS KAKENHI Grant Number JP23K11040.

REFERENCES

[1] I. Lee, "Internet of things (iot) cybersecurity: Literature review and iot cyber risk management," *Future Internet*, vol. 12, no. 9, 2020. [Online]. Available: https://www.mdpi.com/1999-5903/12/9/157

[2] A. Corallo, M. Lazoi, M. Lezzi, and A. Luperto, "Cybersecurity awareness in the context of the industrial internet of things: A systematic literature review," *Computers in Industry*, vol. 137, p. 103614, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0166361522000094

[3] Y. Xiao, Y. Jia, C. Liu, X. Cheng, J. Yu, and W. Lv, "Edge computing security: State of the art and challenges," *Proceedings of the IEEE*, 2019.

[4] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A survey on the edge computing for the internet of things," *IEEE Access*, vol. 6, pp. 6900–6919, 11 2017.

[5] H. H. Kim and J. Yoo, "Analysis of security vulnerabilities for iot devices," *Journal of Information Processing Systems*, vol. 18, pp. 489–499, 8 2022. [Online]. Available: www.kips.or.kr

[6] S. Rajendran and R. M. Lourde, "Security threats of embedded systems in iot environment," *Lecture Notes in Networks and Systems*, vol. 89, pp. 745–754, 2020. [Online]. Available: https://link.springer.com/chapter/10.1007/978-981-15-0146-3_70

[7] A. S. Banks, M. Kisiel, and P. Korsholm, "Remote attestation: A literature review," *CoRR*, vol. abs/2105.02466, 5 2021. [Online]. Available: http://arxiv.org/abs/2105.02466

[8] Z. Ling, H. Yan, X. Shao, J. Luo, Y. Xu, B. Pearson, and X. Fu, "Secure boot, trusted boot and remote attestation for arm trustzone-based iot nodes," *Journal of Systems Architecture*, vol. 119, p. 102240, 2021. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762121001661

[9] I. Lebedev, K. Hogan, and S. Devadas, "Invited paper: Secure boot and remote attestation in the sanctum processor," in *2018 IEEE 31st Computer Security Foundations Symposium (CSF)*, 2018, pp. 46–60.

[10] R. Wang and Y. Yan, "A survey of secure boot schemes for embedded devices," in *2022 24th International Conference on Advanced Communication Technology (ICACT)*, 2022, pp. 224–227.

[11] A. Dave, N. Banerjee, and C. Patel, "Care: Lightweight attack resilient secure boot architecture with onboard recovery for risc-v based soc," in *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, 2021, pp. 516–521.

[12] J. Haj-Yahya, M. M. Wong, V. Pudi, S. Bhasin, and A. Chattopadhyay, "Lightweight Secure-Boot Architecture for RISC-V System-on-Chip," in *20th International Symposium on Quality Electronic Design (ISQED)*, 2019, pp. 216–223.

[13] C. Profentzas, M. Günes, Y. Nikolakopoulos, O. Landsiedel, and M. Almgren, "Performance of secure boot in embedded systems," in *2019 15th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2019, pp. 198–204.

[14] wolfSSL. (2023) wolfboot: wolfssl secure bootloader. [Online]. Available: https://github.com/wolfSSL/wolfBoot

[15] DENX Software Engineering. (2023) U-boot. [Online]. Available: https://source.denx.de/u-boot/u-boot

[16] SiFive. (2023) Sifive hifive unmatched. [Online]. Available: https://www.sifive.com/boards/hifive-unmatched

[17] R. Wang and Y. Yan, "A survey of secure boot schemes for embedded devices," in *2022 24th International Conference on Advanced Communication Technology (ICACT)*, vol. 2022-February. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 224–227.

[18] (2019) Trusted platform module library specification. Trusted Computing Group. [Online]. Available: https://trustedcomputinggroup.org/work-groups/trusted-platform-module/

[19] ARM. Building a secure system using trustzone®technology. [Online]. Available: https://documentation-service.arm.com/static/5f212796500e883ab8e74531

[20] wolfSSL. (2023) wolfssl embedded ssl/tls library. [Online]. Available: https://github.com/wolfSSL/wolfssl

[21] K. Liu, M. Yang, Z. Ling, H. Yan, Y. Zhang, X. Fu, and W. Zhao, "On manually reverse engineering communication protocols of linux based iot systems," *IEEE Internet of Things Journal*, vol. 8, pp. 6815–6827, 7 2020. [Online]. Available: https://arxiv.org/abs/2007.11981v2

[22] M. Bettayeb, Q. Nasir, and M. A. Talib, "Firmware update attacks and security for iot devices survey," *ACM International Conference Proceeding Series*, 3 2019. [Online]. Available: https://dl.acm.org/doi/10.1145/3333165.3333169

[23] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanovic, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the Fifteenth European Conference on Computer Systems*, ser. EuroSys '20, 2020.

[24] Risc-v open source supervisor binary interface (opensbi). [Online]. Available: https://github.com/riscv-software-src/opensbi

[25] riscv.org. (2023) Risc-v supervisor binary interface specification. RISC-V Platform Specification Task Group. [Online]. Available: https://github.com/riscv-non-isa/riscv-sbi-doc/releases/download/v1.0.0/riscv-sbi.pdf

[26] devicetree.org. Devicetree specification release v0.4. [Online]. Available: https://github.com/devicetree-org/devicetree-specification/releases/download/v0.4/devicetree-specification-v0.4.pdf

[27] R. C. Merkle, "Protocols for public key cryptosystems," *Proceedings - IEEE Symposium on Security and Privacy*, pp. 122–134, 7 1980.

[28] ——, "A certified digital signature," in *Advances in Cryptology — CRYPTO' 89 Proceedings*, G. Brassard, Ed. New York, NY: Springer New York, 1990, pp. 218–238.

[29] A. Waterman, K. Asanović, and J. Hauser. (2021) The risc-v instruction set manual volume ii: Privileged architecture. RISC-V International. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[30] SiFive. (2022) Sifive fu740-c000 manual v1p6. [Online]. Available: https://sifive.cdn.prismic.io/sifive/1a82e600-1f93-4f41-b2d8-86ed8b16acba_fu740-c000-manual-v1p6.pdf