Vol.2023-ARC-252 No.16
Vol.2023-SLDM-202 No.16
Vol.2023-EMB-62 No.16
2023/3/23

RISC-V SoCにおけるSecure Bootの実装と検証の高速化に向けた評価

齊木 昭大[1,a)]　大森 侑[1,b)]　木村 啓二[1,c)]

概要：OS のブートプロセスの信頼性保証はコンピューターシステムの信頼性保証に極めて重要である．そこで， デジタル署名やハッシュ値を用いてブートに用いられるバイナリイメージの検証をブート時に行う Secure Boot が広く用いられている．しかしながら，検証には高コストな計算を伴うため， ハードウェアアクセラレータ等を利用しない場合に Secure Boot の処理時間が長くなる．本稿では，ハードウェアアクセラレータを持たない RISC-V SoC 上で Secure Boot を U-Boot に実装し，そのオーバーヘッドを測定した．さらに，検証プロセスの一部の並列処理による高速化を検討し評価を行った結果を報告する．評価の結果，検証はブートプロセスの 26.08%を占めることがわかった．また，4 コアによる検証処理の並列化により最大 4.51 倍の速度向上が可能なことが確認できた．

1. はじめに

IoT デバイスの普及に伴い，各種エッジデバイスへのサイバー攻撃の危険性が問題となっている [1], [2]．また，IoT デバイスは物理的に無防備な環境に配置されることもあるため，OS やその上で動作するアプリケーションの脆弱性を悪用した攻撃のみならず，OS 起動以前のファームウェアを物理的なアクセスにより攻撃することも可能である．すなわち，ファームウェアの改竄が可能であり，システムを根幹から掌握するような攻撃が容易になり得る．

ブート前のファームウェアに対する攻撃は，OS やそれ以上のレイヤーのみで防御することは難しく，ハードウェアレベルから対策を施す必要がある．そこで，Secure Boot の利用が代表的な対策の 1 つとなる．

Secure Boot は，システムのハードウェア的に保護された箇所を信頼の基点 (Root of Trust: RoT) として，ファームウェアや OS 等のソフトウェアイメージに対し，連鎖的に検証を行うことでシステムのブートプロセス全体を保護する．システムのブートプロセスは，基本的に複数のステージから構成される．そのため，各ステージ間で連鎖的に検証を行うことで，信頼の連鎖 (Chain of Trust: CoT) を構築し，ブートプロセス全体の信頼性を担保する仕組みとなっている．

Secure Boot の実装に関して，RoT についてはハードウェアによる保護が望まれるが，それ以降の CoT の構築については様々な実装方法が存在している．ソフトウェアイメージの検証は，基本的にハッシュ値・デジタル署名を利用して行われるが，これらの計算コストは高くエッジデバイスにおけるソフトウェア処理はブート時の大きなオーバーヘッドとなる．

エッジデバイスにおける Secure Boot の軽量化・高速化についてはこれまでにも議論がなされているが，その多くがハードウェアを活用した実装を提案している [3], [4], [5]．また，ソフトウェア実装で Secure Boot について，実行時間やオーバーヘッドの程度について触れている研究も存在するが，ソフトウェアレベルでの解決策を提示するものは見られない．

本稿では，Secure Boot のソフトウェアによる検証プロセスを，RISC-V SoC を搭載した開発ボードである HiFive Unmatched [6] を対象として U-Boot 中に実装し，その性能評価を行った．また，評価の結果を踏まえ，ソフトウェアレベルでの Secure Boot の高速化手法を提案する．検証プロセスの中で最もボトルネックとなり得るハッシュ計算に対し，マルチコアプロセッサを活用した並列化を施すことで，プロセス全体としての高速化を実現した．

以下，2 節で Secure Boot や同等のセキュリティ機能を実装する既存研究・製品を概観し，3 節で HiFive Unmatched の概要を説明する．4 節で Secure Boot の実装について述べ，5 節で性能評価の結果を報告する．6 節で高速化手法

1 早稲田大学基幹理工学研究科
School of Fundamental Science and Engeneering, Waseda Univ.
a) saiki@kasahara.cs.waseda.ac.jp
b) oy@kasahara.cs.waseda.ac.jp
c) keiji@waseda.jp

ⓒ2023 Information Processing Society of Japan

とその評価について報告し，7 節でまとめる．

2. 関連研究

Secure Boot 及びそれを構成するコンポーネントの実装は，その利用目的によって様々に存在する．

Jawad 等の Secure Boot アーキテクチャは，RISC-V SoC においてハードウェアベースの Secure Boot を実装する [3]．Trusted Computing Base (TCB) として，ブートステートを制御する Boot Sequencer と検証を行う Code Authentication Unit (CAU) を導入し，検証鍵の管理も専用のハードウェアユニットを用いることで，ソフトウェアに依存しない軽量な Secure Boot を実現している．CAU は暗号エンジンを搭載しており，演算性能と電力効率のいずれでもソフトウェアより優位な結果を残している．

Avani 等は，RISC-V SoC において Secure Boot とコードリカバリーの機能を提供するハードウェアモジュール *CARE* を提案している [4]．ブートイメージは *CARE* を経由して検証・リカバリープロセスを経た後に RISC-V Core へ送られる．標準的な Secure Boot は不正なイメージの検知行うのみで，正常な状態への回復には手動でのリストアが必要となるが，*CARE* は人手の介入なしにスタンドアロンで回復することを可能としている．

wolfBoot は wolfSSL 社の提供する，32bit マイクロコントローラ (MCU) をターゲットとした軽量なセキュアブートローダである [7]．ハッシュ・デジタル署名を用いた基本的なソフトウェアイメージ検証と，簡易的なファームウェアアップデート機能を備える．必要最低限のデバイスドライバと，同社が開発する軽量暗号ライブラリ wolfSSL [8] を組み込むことで，軽量化・高速化を実現している．

Ling 等は，ARM コアで利用可能な信頼実行環境 (TEE: Trusted Execution Environment) である TrustZone[9] を活用した，Secure Boot， Trusted Boot 及び Remote Attestation のハイブリッドなアーキテクチャを提案している [10]．本稿で扱うものと同等の Secure Boot は，TrustZone の Secure World における，Secure OS のブートで用いられており，複数のブートローダーが連鎖的に後続イメージの検証を行う．これは追加のハードウェアモジュールを利用しないソフトウェアベースの実装であるが，パフォーマンス評価において，Secure OS ブート時のハッシュ計算に多くの時間を消費することが指摘されている．これはイメージの総容量が大きいことに起因しているが，実運用環境では Secure OS のイメージが圧縮可能なため，ボトルネックが解消できるとしている．

ハードウェアベースの実装は，軽量で安全性の高い Secure Boot を構成できるが，適用対象が限定される，実装後の変更に柔軟に対応できないなどの欠点が挙げられる．*CARE* は適用対象を極小デバイスに限定しており，また頻繁なイメージ更新などは想定されていないと明言している．このような点から，文献 [10] のようなチップ製造時に On-Chip に格納される情報など，確実に信頼できる箇所を基点とした，ソフトウェアベースのアーキテクチャが用いられる．しかし近年では，エッジデバイスにおいても専用 OS ではなく汎用の OS イメージが利用されることがあり，使用ソフトウェアなどの環境条件に大きく依存せずにボトルネックの解消を行う必要性がある．

3. HiFive Unmatched

本節では，本稿が Secure Boot の実装・評価の対象とする HiFive Unmatched について，システム構成の概要を説明する．

3.1 ボード構成

SiFive Hifive Unmatched は，RISC-V コアをもつ SoC Freedom U740 を搭載した開発ボードである．Freedom U740 には，組み込みコアである SiFive S7 コアが 1 基，Linux が実行可能なアプリケーションコアである SiFive U74 が 4 基搭載されている [11]．On-Chip のメモリシステムとして，S7 コアが 16KiB の L1 I-Cache と 8KiB の Data Tightly-Integrated Memory (DTIM) を，U74 コアがそれぞれ 32KiB の L1 I-Cache/D-Chache を持つ．共有 L2 Cache は 2MiB 搭載する [12]．また，オンボードで 16GB の DDR4 DRAM と，32MiB の QSPI Flash を備える．外部ストレージとしては，SD カード・NVMe M.2 SSD が使用可能で，いずれもブートデバイスとして利用できる．

表 1 に各コアの仕様を，**図 1** に Freedom U740 SoC の概要構成図をそれぞれ示す．

表 1 Unmatched の各コアの仕様

コア	S7 Core	U74 Core
搭載数	1	4
ISA	RV64IMAC	RV64GC(IMAFDC)
Privilege Mode	M-Mode U-Mode	M-Mode S-Mode U-Mode
L1-I Cache	16KiB 2-way	32KiB 4-way
L1-D Cache	N/A	32KiB 8-way
DTIM	8KiB	N/A
L2 Cache	2MiB 16-way	

3.2 ブートプロセス

HiFive Unmatched は，標準で Linux が起動可能な設計となっている．本稿では Ubuntu Server 22.04 をブート対象の OS として利用する．ブートステージは以下の構成をとる．

(1) ZSBL (Zeroth-Stage Boot Loader)
(2) U-Boot SPL (Second Program Loader)

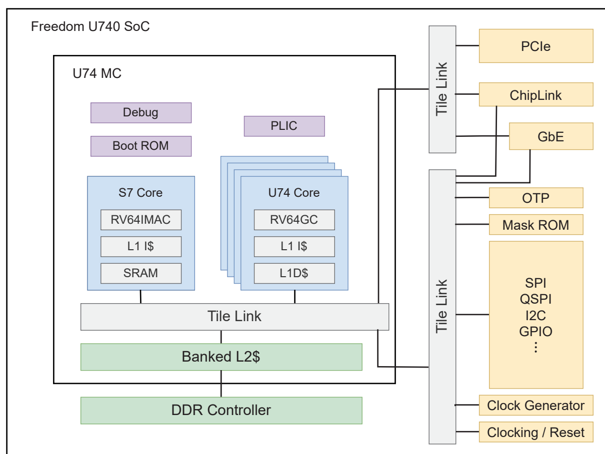

図 1 Freedom U740 SoC の構成 [11]

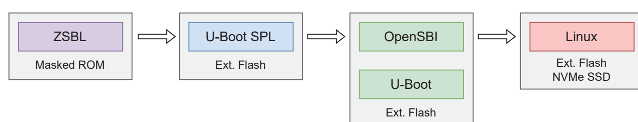

図 2 HiFive Unmatched の Linux ブートフロー [11]

(3) OpenSBI
(4) U-Boot
(5) Linux Kernel

HiFive Unmatched の標準 Linux ブートフローを**図 2** に示す．システムはリセット後，Reset Vector から最初のブートステージである ZSBL を実行する．各ステージの動作は以下の通りである．

ZSBL Masked ROM (Read Only) に格納されており，QSPI Flash もしくは SD カードの特定の領域から次のブートイメージである U-Boot SPL をロードし，無効化状態の L2 Cache 領域に展開・実行する．

U-Boot SPL 後続の U-Boot の一部で，起動直後の無効化されている L2 Cache 領域から実行され，CPU のクロックアップや各種ペリフェラルの初期化を行う．このプロセスで DRAM が使用可能になり，共有 L2 Cache も有効化される．初期化完了後は後続イメージを DRAM 上に展開し，処理を移す．U-Boot SPL がロードするイメージは以下の 3 種類である．
- OpenSBI
- U-Boot
- Flattened Device Tree (FDT)

OpenSBI RISC-V Supervisor Binary Interface を実装したファームウェアである．Physical Memory Protection (PMP) の初期化や Priviledge Mode の切り替えを行い，Machine Mode (M-Mode) から 1 つ低位の Supervisor Mode (S-Mode) で U-Boot へ処理を移す．OpenSBI の使用する領域は PMP により保護されるため，S-Mode から直接アクセスすることはできない．

U-Boot オープンソースのブートローダーであり，OS カーネルやそれに付随するイメージを DRAM に展開し，OS を起動する．本稿で使用する Ubuntu Server 22.04 では以下のイメージをロードする．
- Linux Kernel
- Initial Ram Filesystem (initramfs)
- Flattened Device Tree (FDT)

4. Secure Boot の実装

本節では，HiFive Unmatched に対して実装した Secure Boot について，Threat Model と設計，検証プロセスの実装の詳細を述べる．

4.1 Threat Model

本稿では，IoT エッジデバイスを攻撃対象とし，攻撃者がデバイスに物理的にアクセス可能であると想定する．攻撃者は，停止しているデバイスの Flash Memory や外部ストレージに格納されているファームウェア，ソフトウェアイメージを直接改竄することが可能である．ただし，サイドチャネル攻撃など，イメージ改竄以外の高度なハードウェア攻撃は本稿の想定範囲外とする．また，Masked ROM に格納されているプログラムコードは読み取り専用であり，構造上改変が不可能なため，暗黙的に信頼できるものとする．

Secure Boot はブート時点でのソフトウェア整合性・信頼性を保証するものであり，ランタイムにおける，保護領域外へのマルウェアの注入や改竄は対象外となるが，Keystone[13] などの TEE や他 Remote Attestation を行う手法を併用することでランタイムでの保護を行うことも可能である．本稿は Keystone の Security Monitor を実装した OpenSBI をファームウェアとして利用するが，Keystone の機能については利用していないため，これを対象外とする．

4.2 設計

4.2.1 Root of Turst

RoT は，ブートプロセスの開始点であり，暗黙的に信頼できる領域に存在する ZSBL とする．RoT には本来，製造過程で組み込まれるような信頼できる情報を用いて，後続イメージ検証するような仕組みが必要となる．文献 [10] では，改変が困難な On-Chip ROM 内のブートローダーと，改変不可能な eFuse に格納された公開鍵を利用して RoT を構築している．しかし，今回使用する HiFive Unmatched にそのようなセキュリティ的な要素は存在しない．本稿の目的は，ハードプロセッサにおける Secure Boot プロセスの性能評価であるため，ZSBL が文献 [10] と同様の検証プロセスを持つと仮定した上で進める．

4.2.2 Chain of Trust

CoT は，前述の仮定の上で，ZSBL をベースとして構築

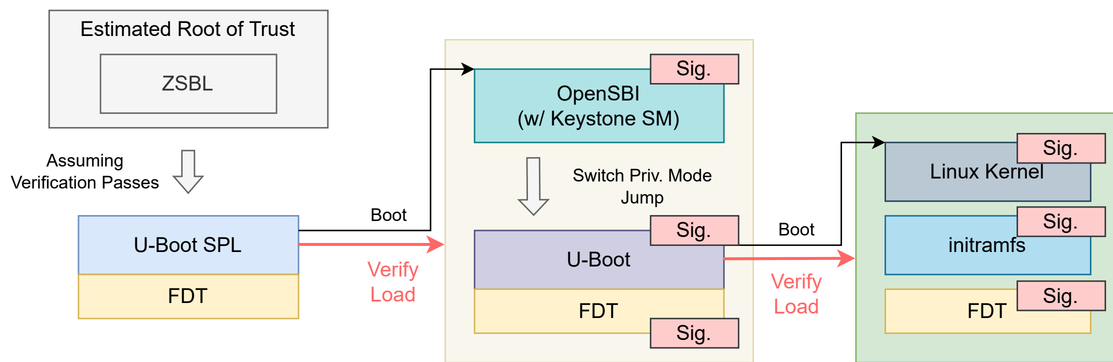

図 3 Chain of Trust の構築フロー

される．CoT の構築フローを**図 3** に示す．基本動作として，先のブートローダーが 1 つ後のステージのイメージ検証を行い，パスした後に処理を移すという流れをとる．検証されるイメージは，全て事前に適切なプロセスでハッシュ・署名が付加されている必要がある．ハッシュ・署名は，他のブートに必要な情報と共にイメージヘッダとして付加される（図中の Sig.）．本稿で用いるイメージヘッダについては 4.2.3 節で述べる．

OpenSBI と U-Boot について，これらは独立したソフトウェアだが U-Boot SPL によって同時に検証・ロードされる．U-Boot へは OpenSBI から処理が移ることになるが，U-Boot についての情報は前ステージの U-Boot SPL から OpenSBI に渡されるため，CoT の構築においてはまとめて 1 段階として扱う．

4.2.3 検証プロセスの実装

図 3 より，検証プロセスが実装されるのは U-Boot SPL，U-Boot の 2 種類のブートローダーである．U-Boot リポジトリ [14] の v2022.07 をベースに実装を行った．検証プロセスでは，ロードする各イメージに対して

- イメージヘッダの読み取り
- イメージハッシュ値の検証
- 公開鍵のハッシュ値の検証
- 署名の検証

を行う．いずれかの検証に失敗した場合は，その時点でシステムをハングアップさせブートを停止する．

検証対象のイメージは，事前にハッシュ・署名の計算がされ，他のロードに必要な情報とともにイメージヘッダとして付加されている．イメージヘッダには以下の情報が必要に応じて格納されている．

- イメージサイズ
- イメージタイプ
- ハッシュアルゴリズム
- 署名アルゴリズム
- タイムスタンプ
- ロードアドレス
- イメージのハッシュ値
- 公開鍵のハッシュ値
- 署名

ヘッダにはハッシュ値・署名の他に，ブートに関わる情報が含まれるため，イメージのハッシュ値は，ヘッダのハッシュ値・署名以外の部分を含めて計算する．これにより，イメージヘッダの改竄による意図しない動作から保護することが可能となる．

また，ブートローダーは署名検証に用いる公開鍵を保持している必要がある．これは，各ローダーで利用する FDT に格納している．この方法は，Chromium OS 向けに実装された U-Boot Verified Boot [15] で用いられている．

ハッシュアルゴリズムには SHA3-384 を，署名アルゴリズムには ED25519 を採用した．これらの計算には，wolfBoot でも利用されていたライブラリ wolfSSL の，GPL-2.0 でライセンスされたオープンソース版を用いた．ライブラリを利用する計算部分の実装については，wolfBoot での実装を参考にしている．

5. Secure Boot の性能評価

本節では，4 節で述べた Secure Boot を実装したブートプロセスの性能評価の結果について述べる．**表 2** に評価で使用した環境・ソフトウェアを示す．各種計測値は，3 回平均値を示す．

表 2 評価環境

Board	SiFive HiFive Unmatched A00
SoC	Freedom U740-c000
Core	SiFive S7 / SiFive U74
DRAM	DDR4-2400 16GB
Storage	SanDisk microSDHC 32GB (QSPI) Samsung SSD 970EVO 250GB (NVMe)
Firmware	OpenSBI v1.1 (w/Keystone SM)
Bootloader	U-Boot v2022.07-dirty
Crypto Lib.	wolfSSL v5.5.0-stable
OS	Ubuntu Server 22.04 Linux 5.15.0-1007-generic

©2023 Information Processing Society of Japan

5.1 評価方法

評価は，各コアに搭載される Hardware Performance Monitor (HPM) を用いて実行サイクルを計測することにより行う．実行サイクルは Control and Status Register (CSR) の mcycle (cycle) レジスタでカウントされており，これを読み出すことで計測可能である．また，計測は各ブートステージ全体についてと，Secure Boot の各検証プロセスを独立に行う．なお，HiFive Unmatched はブートプロセスの途中で使用コアや動作周波数が変化するため，評価結果は計測したクロックサイクル数と動作周波数から導出した実時間で示す．

5.2 評価結果

5.2.1 ブートステージ全体

表 3 に計測した各ブートステージの処理クロックサイクルから導出した処理時間を示す．

表 3 ブートステージ全体の処理時間

Boot Stage		Time [s]	Freq.
ZSBL		7.112	26MHz
U-Boot SPL		3.460	1.2GHz
OpenSBI	_start → sbi_init()	0.1703	1.2GHz
	init_coldboot()	0.1826	1.2GHz
U-Boot		14.20	1.2GHz
TOTAL		25.12	

5.2.2 検証プロセス

表 4 に Secure Boot の検証プロセスの処理時間を示す．表 4 より，U-Boot SPL においては，各イメージの署名検証プロセスが合計で約 89%を，U-Boot においては，initramfs と Kernel のハッシュ計算プロセスが合計で約 95%を占めている．特に initramfs と Kernel のハッシュ計算については，検証プロセスの全体と比較した場合でも約 72%を占めており，大きなオーバーヘッドとなっているとわかる．また，署名検証プロセスは一定長の入力に対しての計算であるため，イメージに依らず一定の実行時間となるが，ハッシュ計算はその特性上，イメージのサイズに比例して実行時間が増大する．検証対象イメージのサイズを**表 5** に示す．

5.2.3 検証プロセスの占める割合

表 3，表 4 を利用し，Secure Boot の検証プロセスがブート全体に占める割合を導出した．各ブートローダーに対する割合を**表 6** に，ブートプロセス全体での割合を**表 7** にそれぞれ示す．

表 6 において，U-Boot SPL では 45.01%，U-Boot では 35.12%を検証プロセスが占める結果となっている．U-Boot についてはユーザーインターフェースを備えており，その待機時間が少なからず含まれている．これを除いた純粋な実行時間であれば，検証プロセスはより大きな割合を占めると考えられる．また，表 7 では，ブートプロセス全体に対して，検証プロセスが 26.08%を占める結果となっている．

表 4 検証プロセスの処理時間

Loader	Target	Process	Time [s]	Freq.
U-Boot SPL	OpenSBI	Hash	0.03034	1.2GHz
		Signature	0.3845	
	U-Boot	Hash	0.1042	
		Signature	0.3847	
	FDT	Hash	2.934e-3	
		Signature	0.3846	
U-Boot	initramfs	Hash	3.653	1.2GHz
		Signature	0.09042	
	Kernel	Hash	1.064	
		Signature	0.09033	
	FDT	Hash	3.683e-4	
		Signature	0.08982	
TOTAL			6.552	

表 5 各イメージのサイズ

Image	Size [bytes]
OpenSBI	187,512
U-Boot	646,672
U-Boot FDT	21,987
initramfs	101,511,746
Kernel	29,521,920
Linux FDT	10,565

表 6 検証プロセスがブートローダーに占める割合

U-Boot SPL	検証プロセス [s]	1.564
	プロセス全体 (sec)	3.460
	割合 [%]	45.20
U-Boot	検証プロセス [s]	4.987
	プロセス全体 (sec)	14.20
	割合 [%]	35.12

表 7 検証プロセスがブートプロセス全体に占める割合

検証プロセス [s]	6.552
プロセス全体 [s]	25.12
割合 [%]	26.08

6. 高速化手法とその評価

5 節において，サイズの大きいイメージのハッシュ計算が，ブートプロセスのオーバーヘッドの大きな要因となることを確認した．本節では，この結果を踏まえ，検証プロセスを高速化する手法を提案する．

6.1 並列処理による高速化

サイズの大きいイメージの検証がボトルネックとなることは，文献 [10] の Secure OS に関して指摘されていたが，イメージサイズの削減が可能であることから問題とされて

いなかった．本稿では，ハッシュ計算のプロセスそのものに改良を加えることで，ボトルネックの解消を実現する．

本稿で使用した HiFive Unmatched は．アプリケーションコアである SiFive U74 を 4 コア搭載したマルチコアのマシンである．近年ではエッジデバイスにおいてもマルチコアが普及しつつあるため，マルチコアを使用した並列処理による高速化を提案する．

6.1.1 従来のハッシュ計算

4.2.3 節で述べたように，従来の実装は wolfBoot での実装を参考にしている．wolfSSL は OpenSSL と同様の API を持っており，init / update / final 関数を呼び出すことで計算を行う．イメージ先頭から 128bytes ブロックごとに逐次 update 関数を通し，1 つのハッシュ値を得る一般的な実装である．模式図を**図 4** に示す．

6.1.2 並列化手法

従来のイメージ先頭から逐次計算で 1 つのハッシュ値を得る方法では，前後のブロックに依存性がある．この依存性は，ブロックごとに 1 つのハッシュ値を持たせることで解消可能である．これにより，並列処理自体は可能となるが，ブロック数分のハッシュ値を保持する必要がある．文献 [16] では，イメージヘッダにブロック数分のハッシュ値を保持しているが，これはイメージヘッダの肥大化を招くため望ましくない．そこで，ブロックごとのハッシュ値を，先頭から再度ハッシュ関数を通すことで 1 つのハッシュ値を得る方針とした．ここでは，このハッシュ値を *Root Hash* と呼ぶ．

ブロックごとのハッシュ値は並列計算可能だが，*Root Hash* の計算は逐次で行う必要がある．しかし，これは高々ハッシュ長 × ブロック数の計算であるため，適切なブロックサイズを選択することで，元イメージの逐次ハッシュ計算と比較しても計算量は少なくなる．

ブロック単位でハッシュ値を計算する場合，内容が一致するブロックが存在した場合にハッシュ値の衝突が起こる．これはセキュリティ的に望ましくない事象のため，各ブロックにブロック番号を付与した上でハッシュ値を計算することで防止する．

n 個のブロックを持つ場合について，i 番目のブロックを B_i，i 番目のハッシュ値を H_i，*Root Hash* を H_{root}，ハッシュ関数を h とすると，この計算は以下の式で表される．

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

模式図を**図 5** に示す．

6.1.3 並列化手法の予備評価

6.1.2 節で述べた並列化手法の効果について，Linux アプリケーションとして実装を行い，予備評価を行った．予備評価の対象は，ロードしたイメージに対してブロック単位でハッシュを計算し，最終的に *Root Hash* を得るまでのプロセスである．比較のため，同一環境で従来実装の評価も行う．計算対象は，イメージサイズの大きい initramfs と Kernel の 2 種類を選択した．予備評価の環境を**表 8** に示す．また計測値は，全て 5 回平均の値をとった．

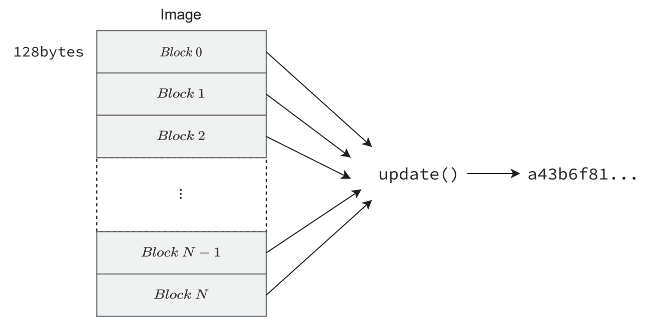

図 4 従来のハッシュ計算の実装

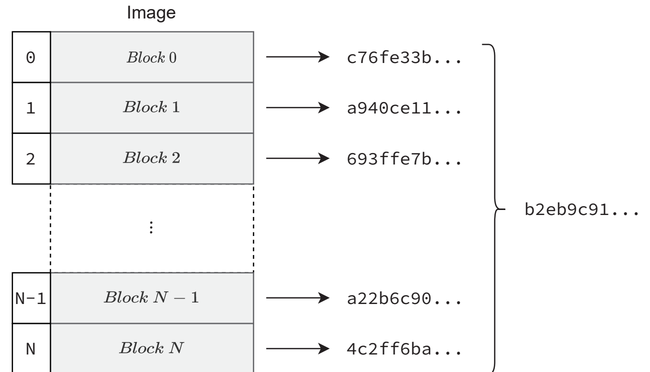

図 5 並列計算可能なブロック分割したハッシュ計算

表 8 評価環境・条件

Board	SiFive HiFive Unmatched A00
SoC	Freedom U740-c000
Core	SiFive U74 × 4
Threads	4
DRAM	DDR4-2400 16GB (On-board)
Storage	SanDisk microSDHC 32GB (QSPI) Samsung SSD 970EVO 250GB (NVMe)
Crypto Lib.	wolfSSL v5.5.0-stable
OS	Freedom U SDK 2022.12.00 Linux 5.19.14
Thread Lib.	libpthread

表 9 従来実装での計算時間

Image	Time [s]
initramfs	3.709
Kernel	1.079

従来実装での計算時間を**表 9** に，並列計算での計算時間を**表 10** にそれぞれ示す．ブロック化した計算においては，ブロックサイズを 1024∼163840bytes の間で変動させ，それに伴う実行時間の変化を確認した．

表 10　ブロック分割した実装の計算時間

Block Size [bytes]	Time [s]			
	initramfs		Kernel	
	Sequential	Parallel	Sequential	Parallel
1024	4.335	1.281	1.262	0.3741
2048	3.935	1.086	1.145	0.3179
4096	3.659	0.9689	1.065	0.2842
8192	3.601	0.9313	1.048	0.2726
16384	3.530	0.9032	1.027	0.2648
32768	3.516	0.8945	1.023	0.2617
65536	3.502	0.8899	1.019	0.2605
81920	3.500	0.8850	1.019	0.2599
98304	3.497	0.8880	1.018	0.2613
114688	3.497	0.8843	1.018	0.2600
131072	3.495	0.8890	1.018	0.2612
147456	3.495	0.8865	1.017	0.2618
163840	3.493	0.8814	1.017	0.2624

表 9 と表 4 の比較により，予備評価環境とブートローダーの環境において実行時間に大きな差異が見られないことが分かる．すなわち，予備評価環境で得られた結果は，ブートローダーの動作するベアメタル環境においても有効性があると考えられる．

表 10 より，ブロック分割した実装では，ブロックサイズの増加につれて計算時間が短縮され，32768∼81920 bytes で initramfs は約 0.88 秒，Kernel は約 0.26 秒に収束している．これ以降も微細な変動は続くが，議論すべき大きな変化はない．また，ブロック分割した実装を逐次計算で行った場合でも，従来実装と差異のない計算時間であることが確認できる．よって，並列化不可能な環境においても，損失を受けずに実装が流用可能であると考えられる．

ブロックサイズの選定について明確な基準は設けていないが，少なくともスレッド数以上のブロック数を保つサイズを選定する必要がある．小さいブロックサイズでは，計算時に保持するハッシュ値の数が多くなるため，メモリ消費量が増加する．大きいブロックサイズでは，収束する点を超えている場合それ以上の高速化の恩恵はなく，また各スレッドへのブロックの配分が均等でない場合，スレッド間の処理時間の差異が大きく開いてしまう可能性がある．それぞれの利点・欠点を考慮したうえで，使用する環境に適したサイズの選択が必要であると考えられる．本評価環境においては，メモリリソースに十分な余裕があるため，収束点付近の 32768∼81920 bytes に設定するのが適切と考える．81920 bytes を選択した場合，initramfs と Kernel でそれぞれ 4.19 倍，4.15 倍の高速化が期待できる．

ここで，高速化倍率がスレッド数以上の値となっているが，これは従来実装と wolfSSL 側の実装が影響している．従来実装では 128 byets 単位で update 関数にデータを渡していたが，wolfSSL の内部実装ではそれを再度 104 bytes 単位に分割している．サイズの値も近いため，単純計算で 2 倍のイテレーションが行われることとなり，これがボトルネックとなっている．

6.1.4　U-Boot への実装・評価

6.1.2 節で述べた並列化手法を U-Boot に対して実装し，性能評価を行った．並列処理の適用対象は，6.1.3 節で予備評価を行った initramfs，Kernel の 2 種類のイメージで，ブロックサイズは 81920 bytes，並列スレッド数は 4 スレッドである．また，評価環境については 5 節で使用した環境と同一である．

並列処理の実装について，実装対象の U-Boot はベアメタルアプリケーションであり，高度なランタイムを持たない．そのため，プロセッサ間割り込みを利用してプロセッサコアを制御し，各ブロックの計算を均等にコアに割り振る形で実装した．また，ブロックサイズの情報を保持するため，イメージヘッダにブロックサイズのフィールドを追加している．

並列処理適用後の各ローダーの処理時間を**表 11** に，並列処理適用後の検証プロセスの処理時間を**表 12** に示す．計測値は，全て 3 回平均の値である．

表 11 より，一部に並列処理を適用した U-Boot の処理時間が 10.50 秒まで短縮されていると分かる．また，並列処理を適用していない U-Boot SPL の処理時間も 3.003 秒に短縮されている．これは，並列処理の実装に伴い，6.1.3 節で述べた重複した分割によるボトルネックを解消したためである．表 12 と表 3 を比較すると，各イメージのハッシュ計算時間が短縮されていることが分かる．特に並列処理を適用した initramfs，Kernel については，それぞれ 4.51 倍，4.46 倍の速度向上となっており，予備評価以上の効果が現れている．各検証プロセスの合計は 2.601 秒と，従来の約 40%まで短縮されており，検証プロセスがブートプロセス全体に占める割合も 12.38%まで減少している．

6.1.3 節で，ブロックサイズの選定時におけるメモリ消費量の考慮について述べたが，本評価の条件で，並列処理のために追加で要するメモリ消費量を**表 13** に示す．

表 11　並列処理適用後の各ブートステージの処理時間

Boot Stage		Time [s]	Freq.
ZSBL		7.149	26MHz
U-Boot SPL		**3.003**	1.2GHz
OpenSBI	_start → sbi_init()	0.1719	1.2GHz
	init_coldboot()	0.1947	1.2GHz
U-Boot		**10.50**	1.2GHz
TOTAL		**21.02**	

7.　まとめ

本稿では，開発ボード HiFive Unmatched を対象として U-Boot に Secure Boot を実装し，その性能評価を行った．評価の結果，Secure Boot の検証プロセスが，ブートプロ

表 12　並列処理適用後の検証プロセスの処理時間

Loader	Target	Process	Time [s]
U-Boot SPL	OpenSBI	Hash	0.02819
		Signature	0.3845
	U-Boot	Hash	0.09785
		Signature	0.3845
	FDT	Hash	2.743e-3
		Signature	0.3845
U-Boot	initramfs	Hash（並列）	**0.8102**
		Signature	0.09009
	Kernel	Hash（並列）	**0.2383**
		Signature	0.08984
	FDT	Hash	3.302e-4
		Signature	0.08990
TOTAL			**2.601**

表 13　並列処理に要するメモリ消費量

イメージ	ブロック数	メモリ消費 [bytes]
initramfs	1240	59520
Kernel	361	17328

セス全体の 26.08%を占めることを確認した．

また，その結果を踏まえた上で，オーバーヘッドの大きな要因であったハッシュ計算の並列処理による高速化する手法を考案し，予備評価行った．ハッシュ計算は通常，イメージの先頭から逐次ハッシュ関数に通してハッシュ値を得るため，そのままでは並列処理はできない．そこで，イメージをブロックに分割し，ブロック単位でハッシュ計算を行うことで，並列処理を可能にした．4 つのアプリケーションコアを搭載する HiFive Unmatched で 4 スレッド並列化の評価を行った結果，適切なブロックサイズを選択することで，最大 4.51 倍の高速化が可能なことを確認した．また，ブロックサイズを変動させる計測において，ブロックサイズの増加につれて計算時間が収束することを確認した．しかし，適切なブロックサイズは実装対象のリソース許容量とのバランスを考慮する必要があるため，環境に応じて慎重に検討する必要がある．

謝辞　本研究においてご助言を頂きました早稲田大学基幹理工学部 森達哉教授，並びに同学部 佐古和恵教授に感謝を申し上げます．

参考文献

[1] Lee, I.: Internet of Things (IoT) Cybersecurity: Literature Review and IoT Cyber Risk Management, *Future Internet*, Vol. 12, No. 9 (online), DOI: 10.3390/fi12090157 (2020).

[2] Corallo, A., Lazoi, M., Lezzi, M. and Luperto, A.: Cybersecurity awareness in the context of the Industrial Internet of Things: A systematic literature review, *Computers in Industry*, Vol. 137, p. 103614 (online), DOI: https://doi.org/10.1016/j.compind.2022.103614 (2022).

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

[4] Dave, A., Banerjee, N. and Patel, C.: CARE: Lightweight Attack Resilient Secure Boot Architecture with Onboard Recovery for RISC-V based SOC, *2021 22nd International Symposium on Quality Electronic Design (ISQED)*, pp. 516–521 (online), DOI: 10.1109/ISQED51717.2021.9424322 (2021).

[5] Wang, R. and Yan, Y.: A Survey of Secure Boot Schemes for Embedded Devices, *2022 24th International Conference on Advanced Communication Technology (ICACT)*, pp. 224–227 (online), DOI: 10.23919/ICACT53585.2022.9728840 (2022).

[6] SiFive: SiFive HiFive Unmatched, `https://www.sifive.com/boards/hifive-unmatched` (Accessed 2023/2/10).

[7] wolfSSL: wolfBoot, `https://github.com/wolfSSL/wolfBoot` (Accessed 2023/2/1).

[8] wolfSSL: wolfSSL Embedded SSL/TLS Library, `https://github.com/wolfSSL/wolfssl` (Accessed 2023/2/1).

[9] ARM: Building a Secure System using TrustZone®Technology, `https://documentation-service.arm.com/static/5f212796500e883ab8e74531` (Accessed 2023/1/20).

[10] Ling, Z., Yan, H., Shao, X., Luo, J., Xu, Y., Pearson, B. and Fu, X.: Secure boot, trusted boot and remote attestation for ARM TrustZone-based IoT Nodes, *Journal of Systems Architecture*, Vol. 119, p. 102240 (online), DOI: https://doi.org/10.1016/j.sysarc.2021.102240 (2021).

[11] SiFive: SiFive FU740-C000 Manual v1p6, `https://sifive.cdn.prismic.io/sifive/1a82e600-1f93-4f41-b2d8-86ed8b16acba_fu740-c000-manual-v1p6.pdf` (Accessed 2023/2/10).

[12] SiFive: HiFive Unmatched Datasheet, `https://sifive.cdn.prismic.io/sifive/d0556df9-55c6-47a8-b0f2-4b1521546543_hifive-unmatched-datasheet.pdf` (Accessed 2023/2/10).

[13] Lee, D., Kohlbrenner, D., Shinde, S., Asanovic, K. and Song, D.: Keystone: An Open Framework for Architecting Trusted Execution Environments, *Proceedings of the Fifteenth European Conference on Computer Systems* (2020).

[14] DENX Software Engineering: U-Boot, `https://source.denx.de/u-boot/u-boot` (Accessed 2023/2/1).

[15] DENX Software Engineering: U-Boot Verified Boot, `https://source.denx.de/u-boot/u-boot/-/blob/fe4c21de4fbf5756d354d2473ffc675e7596ccfb/doc/uImage.FIT/verified-boot.txt` (Accessed 2023/1/21).

[16] Muduli, S. K., Subramanyan, P. and Ray, S.: Verification of Authenticated Firmware Loaders, *2019 Formal Methods in Computer Aided Design (FMCAD)*, pp. 110–119 (online), DOI: 10.23919/FMCAD.2019.8894262 (2019).

ⓒ2023 Information Processing Society of Japan