一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report
CPSY2023-15,DC2023-15(2023-08)

RISC-V Keystone における Enclave アプリケーションキャッシュ機能の拡張

梅澤　拓夢†　齊木　昭大†　木村　啓二†

† 早稲田大学 情報理工・情報通信専攻

あらまし　IoT 社会の発展やクラウドコンピューティングの普及によってセキュリティ面の対策が不十分な端末の増加や, 脆弱性の放置されたホスト OS といった問題が生じている. そのため OS からも隔離された安全な実行環境（TEE）でアプリケーションを実行する技術が重要なものとなっている. しかし TEE で実行されるアプリケーションは実行前にその完全性を保証するため，メモリ内に展開されたバイナリイメージのハッシュ計算が必要となり，そのためアプリケーション起動時のオーバーヘッドが問題となる．本論文では，RISC-V の TEE である Keystone に対して繰り返し同じアプリケーションが呼び出された際にハッシュ計算を繰り返す必要無く TEE によるアプリケーション実行を行うことの出来るキャッシュ機構について提案し, 実装と評価をした結果について報告する. 評価の結果，提案するキャッシュ機構により TEE によるアプリケーションの起動が 40〜50 倍高速化可能なことが確認できた.

キーワード　TEE, アプリケーション検証高速化, Keystone

Extension of Enclave Application Cache to RISC-V Keystone

Takumu UMEZAWA†, Akihiro SAIKI†, and Keiji KIMURA†

† Department of Computer Science and Communication Engineering, Waseda University

Abstract　The development of the IoT society and the spread of cloud computing have also led to an increase in devices with inadequate security measures and vulnerable host OSs. Therefore, the importance of trusted execution environment (TEE) technologies, where programs can be executed in an isolated manner even from the host OS, have increased. However, application execution in a TEE requires the expensive hash calculation of its executable binary image extracted in the main memory to guarantee its integrity before execution. To mitigate this overhead, this paper will propose a cache mechanism that can bypass the hash calculation by storing the previously executed binary image in a secure memory space. The evaluation shows the proposed cache mechanism can give us 40 to 50 times speedup in TEE applications startup.

Key words　TEE, acceleration of verification, Keystone

1. はじめに

IoT 社会の発展やクラウドコンピューティングの普及によって，現在では大小様々なコンピュータシステムが社会のあらゆるところで稼働している．しかしながら，セキュリティ対策が不十分な端末と接続するリスクや，システムに内在する脆弱性によりホスト OS 自体が信頼できないといった状況が生じている. そこで末端端末内部における情報保護技術が注目されている．このような技術の一つに，OS を含む他のソフトウェアから隔離された安全な実行環境でアプリケーションを実行する TEE(Trusted Execution Environment) がある [1].

各 CPU ベンダから様々な方式の TEE が提案・利用されているが，これらのベンダ固有の TEE はトレードオフが固定されていて, カスタマイズの余地がほとんどない. センサなどの末端機器においては端末ごとに消費電力やコストの制約, 脅威モデルが様々に存在するため，RISC-V ではこれらに柔軟に対応可能となるよう，オープンソースフレームワークによって構築される TEE である Keytsone が提案されている [2].

TEE の方式として，Intel SGX や RISC-V Keystone などの，システム内にプログラムの隔離実行環境（Enclave）を導入するものがある [2], [3]．これらの TEE の問題点として，Enclave 内部で実行するアプリケーションの起動に時間がかかることが挙げられる. この要因としてはハッシュ計算とメモリマップの検証のオーバーヘッドが大きいことが挙げられる [4], [5]. ハッシュ計算は実行前にその完全性を確認し，安全なアプリケーションであることを保証する (アテステーションする) ために，Enclave の持つバイナリファイルについて実施される. また，メモリマップ検証は Enclave の持つメモリマップ (PTE) が意図しない領域

を指すことによって不正な命令を実行することを防ぐために行われる．これらの処理では，計算コストの大きいハッシュ計算をメモリマップ検証やアプリケーションのバイナリイメージ全体に対して行う必要がある．

この問題に対してこれまでにハッシュ計算用ハードウェアアクセラレータを持つ RISC-V システムの提案や [5], あらかじめユーザが特殊な Enclave を作成しこれに事前にアプリケーションをロードしておき（Shadow Enclave）, 実行時にはその Enclave を呼び出すことで Enclave インスタンスを高速に起動する手法 [4] が考案されている．

本稿では RISC-V Keystone に対して Enclave キャッシュ機能を設計し, これの実装及び性能評価を行った．提案するキャッシュ機構では実行されるアプリケーションのバイナリイメージとハッシュ値をメモリ上の安全な領域に保持することで, 同じアプリケーションが実行される際に改めてハッシュ計算やメモリマップ検証を行わずに新たな Enclave を起動する．これらの処理は全て Enclave の管理などを行う Security Monitor 内部で行われ, Shadow Enclave とは異なりユーザーサイドやアプリケーション側で特別な対応は必要としない．

以下，第 2 節で Enclave 起動の高速化に関する既存の研究を紹介し，第 3 節で Keystone のアーキテクチャについて説明する．第 4 節では Enclave 起動におけるハッシュ値のオーバーヘッドを計測し，第 5 節で Enclave キャッシュの実装方法について説明する．第 6 節では Enclave キャッシュによってセキュリティリスクに変化が無いことを確認し, 第 7 節で実際に RISC-V Keystone のソースコードを書き換えて Enclave キャッシュ機能を実装したことによる評価を行う．そして第 8 節でまとめる．

2. 関 連 研 究

本節では Enclave 起動の高速化に関する既存の研究について紹介する．

文献 [5] では, ハッシュ計算, デジタル署名アルゴリズムに対応するハードウェアアクセラレータを搭載した, Keystone と互換性のある RISC-V システムを提案している．多くのセキュリティアルゴリズムはソフトウェアによって実装されているが, このようにハードウェア実装することで計算速度を向上させることが出来る．本稿の提案方式はソフトウェアによるものであり，ハードウェアアクセラレータとの併用も可能である．

Penglai Enclave では Shadow Enclave と呼ばれる特別な Enclave を作成する機能がある．Shadow Enclave はメモリマップ検証やアテステーション用のハッシュ計算をあらかじめ行った, 実行不可能な Enclave である．この Shadow Enclave を作成するとユーザーには Enclave の ID が返される．ユーザーはこの ID を指定して Enclave の作成を行うことで Shadow Enclave を基に Enclave インスタンスの起動を行うことが出来る．この際ハッシュ計算とメモリマップ検証は Shadow Enclave を作成する際に行われているので Enclave インスタンス起動の際にはこれらの計算を行わなくてよく, 高速な Enclave 起動を行うことが出来る [4]. Shadow Enclave では，これの作成や実行を行うためのプログラムを作成する必要があるが，本稿で提案するキャッシュはそのような必要は無い．

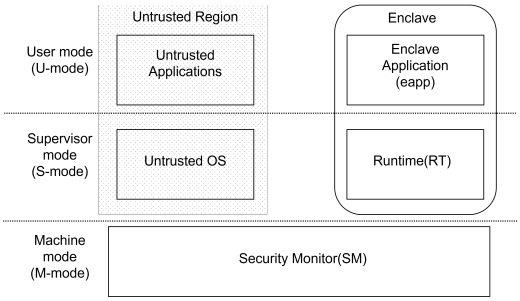

図 1　Keystone システムの外観 (参考文献 [2], Page 2, Figure 1 を参考に作成)

3. RISC-V Keystone のアーキテクチャ

本節では Keystone のアーキテクチャおよびセキュリティの根拠となる RISC-V の権限レベルやメモリ保護機能 (PMP), Enclave について説明する．

3.1. RISC-V の権限レベル

RISC-V は 3 つの権限レベルを持つ．高い権限レベルから順に Machine mode (M-mode)，Supervisor mode (S-mode)， User mode (U-mode) となっている．M-mode は最も高い権限レベルを持ち，全ての物理リソースや割り込みの制御を行う．S-mode はデバイスドライバやカーネルモジュールを含むカーネルに使用される．U-mode は一般のユーザープロセスに使用されるモードである [6].

3.2. Keystone のアーキテクチャ

Keystone のアーキテクチャの外観は図 1 のようになっている．システム全体としては M-mode の機能を利用して S-mode および U-mode において隔離された環境を作成し，その内部でアプリケーションを実行するようになっている．S-mode では信頼できない OS の代わりに独自の Runtime (RT) を用意し，アプリケーション実行時に必要な S-mode の機能を利用できるようにする．ユーザーが Enclave 内で動作させたいアプリケーションは Enclave user-code application (eapp) と呼ばれ，U-mode で実行される．また，Enclave 内外の情報のやり取りには shared memory と呼ばれる共有メモリ領域を利用する．M-mode には security monitor (SM) をおき，Enclave の生成，実行，破棄を全て管理している [2]. SM は Enclave の meta data を持っていて, ここには Enclave のアドレスやサイズなどの情報や, アテステーションに利用されるハッシュ値の情報が含まれる．

3.3. メモリ保護機能 (PMP)

Keystone における隔離環境を作成するため，RISC-V の ISA 仕様が定める Physical Memory Protection (PMP) が利用される．PMP では PMP エントリを作成し，これによってメモリの特定領域に対する U-mode および S-mode からのアクセス権限を管理する．

PMP エントリは configuration レジスタとアドレスレジスタから構成される．これらのレジスタは通常 M-mode からのみアクセス可能である．Configuration レジスタは図 2 のようなフィー

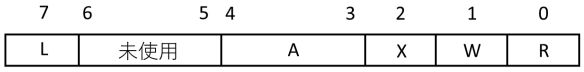

図 2　PMP configuraton レジスタ (参考文献 [7], Page46, Figure 3.27 より)

ルドを持つ. X,W,R ビットはビットがセットされると PMP がそれぞれ実行, 読み取り, 書き込みを許可するようになる. A フィールドは configuration レジスタと対応するアドレスレジスタのアドレスマッチングモードを指定する. L ビットにビットが立つと PMP レジスタはロックされ, configuration レジスタとアドレスレジスタへの書き込みは無視される. また, L ビットにビットが立っている場合 M-mode からのアクセスも X,W,R ビットによって制限されることになる. L ビットにビットが立っていない場合は U-mode および S-mode からのアクセスを X,W,R ビットによって制限することになる [7].

PMP エントリは静的に優先順位がつけられていて, アクセスされたバイトのどれかに合致する最も優先度の高い PMP エントリがそのアクセスの可否を決定する. このとき, アクセスしようとしている全バイトの範囲を PMP がカバーしていない場合はアクセスは拒否される. アクセスしようとしている全バイトを PMP がカバーしている場合は L,X,W,R ビットを参照してアクセスの可否を判断する [7].

Keystone ではまず SM の起動時に最も優先度の高い PMP エントリを SM で使用するメモリ領域に対して設定する. そして優先度が最低の PMP エントリを全てのメモリ領域をカバーし, フルパーミッションとなるように設定する (OS PMP エントリ). これによって OS は他の PMP エントリによって保護されていない全ての領域に対して権限を持ち, Enclave のメモリが要求されると連続する適切なメモリ領域を見つけ, 割り当てることが出来る [2].

各コアはそれぞれに PMP エントリを持っていて, Keystone では Enclave を作成する際に PMP によって全てのコアからの Enclave 領域へのアクセスを禁止する. この PMP の設定はプロセッサ間割り込みによって全てのコアに伝播される. そして Enclave の実行時にコンテキストが Enclave に変わると, その実行されるコアでのみ Enclave の PMP エントリのパーミッションビットを許可に切り替える. 一方でそのコアでは OS PMP エントリのパーミッションビットを不許可に切り替える. コンテキストが非 Enclave に変わると Enclave の PMP エントリのパーミッションビットを不許可とし, OS PMP エントリのパーミッションビットを許可に切り替える. そして Enclave が破棄される時には再びプロセッサ間割り込みによって全てのコアで PMP エントリを無効化する. このようにして隔離実行環境を実現する [2].

3.4. Keystone アプリケーションの構成

Keystone は M-mode で動作する SM, ユーザが実行する Keystone 実行ファイル, 及び SM と実行ファイルの間でデータのやり取りを行う Keystone ドライバで構成される.

SM は Enclave を Meta data を使って管理する. Meta data には Enclave のアドレスやサイズなどの Enclave の実行や破棄に必要な情報, 及びアテステーションに利用されるハッシュ値を持つ.

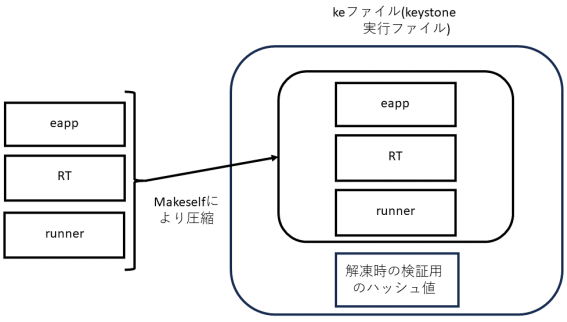

図 3　Keystone 実行ファイルの構成

Keystone 実行ファイルは, makeself によって作られる自己解凍ファイルである. 自己解凍ファイルは, 図 3 に示すように eapp, RT, 及び runnner ファイルという Keystone のドライバや SM と協調して Enclave の起動や実行を制御するファイルを圧縮した TAR アーカイブとシェルスクリプトからなる. 実行時にはこのシェルスクリプトによってアーカイブ中のファイルを一時ディレクトリ上に展開し, 埋め込まれた eapp を実行する. 実行後は一時ディレクトリごと展開したファイルを削除する. またこの自己解凍ファイルは自身の完全性を確認するためのハッシュ値を持ち, ファイル解凍時に自己検証を行う [8].

3.5. Enclave 起動手順

Keystone における Enclave の起動の手順は以下のようになる. まず, runner ファイルが実行され, ここから Keystone ドライバに制御が移る. ドライバでは eapp, RT のサイズなどを基に必要なメモリサイズを計算し Enclave に利用するメモリの範囲を決め, これらの情報に基づき Linux カーネルから必要なメモリの確保を行う. そして Enclave 領域内に eapp, RT の場所を示すページテーブルを作成し, eapp と RT のバイナリイメージをコピーする. Enclave 領域の最下位の 1 ページは root page table となる. 次にプログラム実行に利用される stack を作成し, ここに向かう PTE も作成する. そして Enclave 内外で情報をやり取りするために利用される shared memory 領域を確保し, この領域を指す PTE を作成する.

最後にここまでで作成した Enclave の情報を SM に渡す. SM 上では Enclave 領域と shared memory 領域をそれぞれ PMP によって保護し, Enclave 管理用の meta data を取得する. そしてメモリマップが正当な領域 (Enclave 領域あるいは shared memory 領域) を指すことを確認し, アテステーションを行うためのハッシュ値を計算する.

これらの操作の結果, 最終的には図 4 のようにメモリ上に Keystone 実行ファイルが持っていた eapp, RT がメモリ上に展開され, PTE と shared memory 領域が置かれる. SM は eapp, RT から計算されたハッシュ値を含む Enclave 管理用の meta data を持つこととなる.

4. Enclave 起動時のオーバーヘッドの調査

まずハッシュ計算とメモリマップ検証をバイパスすることでどの程度の性能向上が期待できるのかを調べるために, Enclave

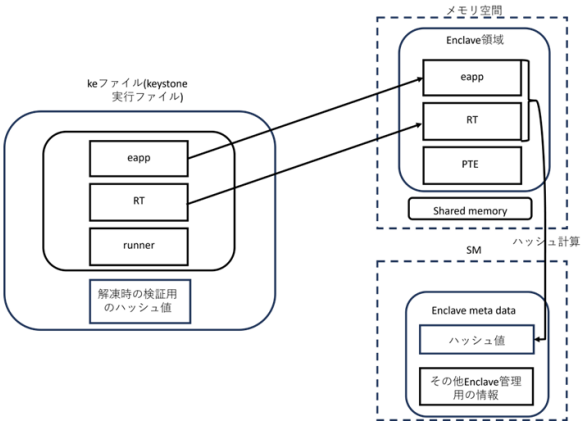

図 4　Enclave の起動の様子

表 1　Hifive Unmatched のマシン性能

	U74	S7
コア数	4	1
L1I キャッシュ	32KiB	16KiB
L1d キャッシュ	32KiB	なし
L2 キャッシュ	2MiB	
動作周波数	1.2GHz	

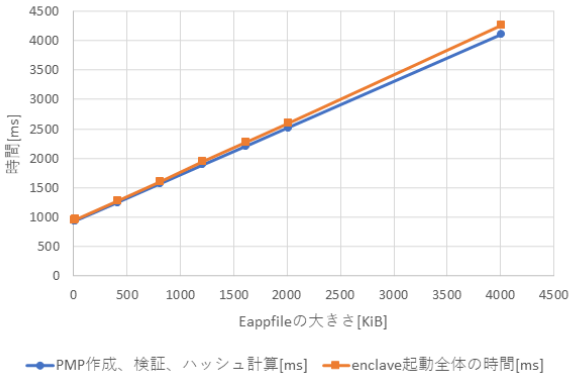

図 5　Enclave 起動のオーバーヘッドの調査

を通常の手順で起動した際の起動時間，及び起動時間中のハッシュ計算やメモリマップ検証に要する時間を測定した．

4.1. 評価環境

本稿における測定には RISC-V マシンである SiFive 社の Hifive Unmatched [9] を利用した．Hifive Unmatched は表 1 のように SiFive U74 コア 4 つと SiFive S7 コア 1 つが搭載されている．U74 コアはそれぞれ 32KiB の L1D キャッシュと 32KiB の L1I キャッシュを持っていて，S7 コアは 16KiB の L1I キャッシュを持っている．L2 キャッシュは 2MiB であり，動作周波数は 1.2GHz である [9]．また，計測は 10 回行いその平均値を使用した．

4.2. 測定結果

測定結果は図 5 のようになった．横軸が eappfile の大きさ，縦軸が時間を表している．上側のグラフが Enclave の起動にかかる全体の時間を示していて，下側のグラフがハッシュ値計算やメモリマップ検証にかかった時間を示している．

この結果を見ると eapp のサイズにかかわらず，Enclave の起動全体にかかる時間のおよそ 96〜97.5% がメモリマップの検証およびハッシュ値計算といった安全性，完全性の確認に必要な手順であることが分かる．ここからこれらの手順をバイパスすることが出来れば約 25〜40 倍の速度向上が期待出来る．

5. Enclave キャッシュの実装

第 4 節における測定の結果，Enclave 起動にかかる時間の 96〜97.5% がメモリマップ検証，ハッシュ計算という完全性，安全性の検証に使われていることが分かった．本節では本稿で提案する，ユーザーから意識することなくメモリマップ検証とハッシュ計算をバイパスして Enclave の起動を高速に行える Enclave キャッシュについて述べる．本キャッシュの実装は Penglai Enclave の Shadow Enclave [4] を参考としている．

5.1. Enclave キャッシュの作成

提案するキャッシュ機構ではメモリ空間上に Enclave キャッシュを作成し，SM 上に Enclave キャッシュをを管理するための meta data を作成する．Enclave キャッシュは eapp, RT のバイナリおよびメモリマップの情報を持ち，meta data は Enclave キャッシュの位置やサイズなどの情報，Enclave キャッシュ作成時に計算したハッシュ値，適切な Enclave キャッシュが存在するかを検索するキーを持つ．Enclave キャッシュのメモリ領域は PMP によるメモリ保護を行う．

Enclave キャッシュは，図 6 に示すように Keystone 実行ファイルから eapp, RT をメモリ上に展開して PTE を作成するという，通常の Enclave の起動と同じ手続きによって作成する．通常の Enclave との違いは stack の作成を行わないことと shared memory 領域の PMP 保護を行わないことである．これらはアプリケーションの実行時に利用される領域であり，ハッシュ値による完全性の保証対象には含まれない．そのためキャッシュに含む必要は無く，実際に動作する Enclave を起動する際に作れば十分である．それ以外に関しては，Enclave を作成し，PMP エントリによる保護，およびメモリマップ検証とハッシュ計算を行い，必要な情報を SM の持つキャッシュ管理のための meta data に渡す，通常の Enclave 起動と同様の処理を行う．

本稿で実装した Enclave キャッシュ機構では，Enclave キャッシュの meta data に，Keystone 実行ファイルが作成されたときに makeself によって計算されたハッシュ値をキャッシュ検索用のキーとして持たせる．このハッシュ値は RT, eapp, runner ファイルが異なると違った値となるため，Keystone 実行ファイルが更新されて Enclave キャッシュの持つイメージが古くなった場合には，キャッシュミスとして古いバイナリを保持したキャッシュの内容を破棄し，新しいバイナリから適切な Enclave を作成することができる．偶然キーが衝突した場合であっても，キャッシュされた Enclave が起動し，ハッシュ計算をバイパスして不正な Enclave が実行されることはないが，意図した Enclvae は実行されない．キャッシュのキーとして適切なものの選択は今

後の課題である.

このように作られた Enclave キャッシュにおいて, ハッシュ値は eapp, RT バイナリから計算されるため通常の手続きによって Enclave を起動した場合と変わらない値が出される. また, メモリマップに関しても Enclave キャッシュにおいて完全性が確認され, 適切な領域を指すことが確認されているため, これを基に Enclave 内のメモリマップを作成すれば適切な領域を示すメモリマップ情報が作成される.

5.2. Enclave キャッシュの利用

Enclave キャッシュを実装した際の Enclave 起動の流れを図 8 に示す．まず，起動要求された Enclave がキャッシュに存在するかどうか確認する．存在した場合，すなわち Enclave キャッシュがヒットした場合，その Enclave キャッシュを基に Enclave を作成する．ミスした場合は Enclave キャッシュを新たに作成する. そしてその作成した Enclave キャッシュを基に Enclave を作成する.

Enclave キャッシュから Enclave が作成される様子を図 7 に示す. Enclave キャッシュの meta data のキーによるキャッシュのヒットが確認できたら，Enclave キャッシュを基に eapp, RT とメモリマップの情報を Enclave 領域に移す．そして Enclave の meta data を作成し, ハッシュ値を Enclave キャッシュの meta data からコピーする.

以上の手順によりユーザーサイドで意識する必要なく Enclave の高速な起動を実現する.

6. セキュリティ評価

本節では，提案する Enclave キャッシュ機能が，オリジナルの Enclave と同様の外部からの不正なメモリアクセスによる盗聴や改ざんに対する耐性があることを確認する.

6.1. 脅威モデル

本稿で想定する脅威モデルとしては外部の S-mode および U-mode で動作するソフトウェアからの不正なメモリアクセスによるアプリケーションの盗聴および改ざんである. 悪意のあるユーザは S-mode および U-mode から Enclave キャッシュおよび Enclave キャッシュの meta data へのアクセスを試みると想定する.

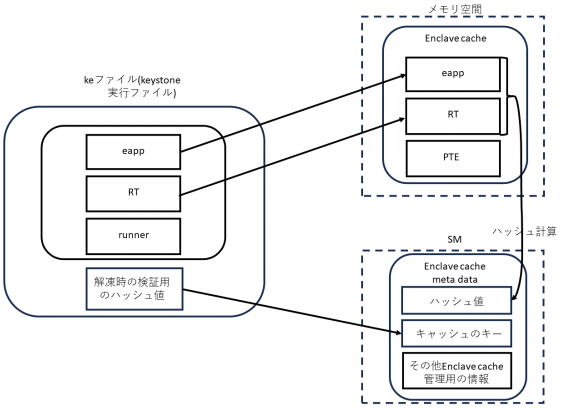

図 6　Enclave キャッシュの作成

Enclave キャッシュのデータの流れ. eapp と RT をメモリ上に展開し, makeself のハッシュ値を Enclave キャッシュ検索用のキーとしている. 本研究では通常起動時の手順を参考にしながら Keystone 実行ファイルから Enclave キャッシュとその meta data の作成をする機能の作成を行った.

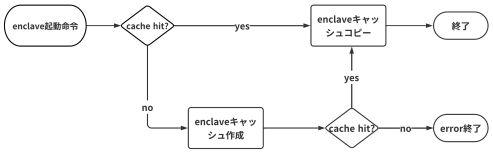

図 7　Enclave キャッシュ実装時の Enclave 起動フロー

Enclave キャッシュのヒット時には Enclave キャッシュから Enclave を起動する. Enclave キャッシュのミス時には Enclave キャッシュを作成し, 作成された Enclave キャッシュを基に Enclave を起動する.

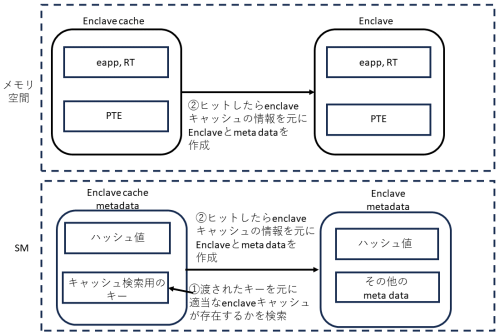

図 8　Enclave キャッシュから Enclave の作成

6.2. 機密性, 完全性

Enclave キャッシュとして使用される領域は PMP によって保護するため，S-mode および U-mode のソフトウェアは Enclave キャッシュ領域内にアクセスしたりメモリの書き換えを行うことは出来ない. また SM の持つメモリは PMP によってあらかじめ保護されているため [2]S-mode 以下の権限のソフトウェアでは meta data にアクセスして Enclave キャッシュが作成時とは別の領域を指すようにしたり，検索用のキーを書き換えたりするといった Enclave イメージのすり替えを行うことは不可能である. 以上よりキャッシュ作成後の機密性や完全性も保たれる. すなわち，外部の S-mode，U-mode で動作するソフトウェアから Enclave キャッシュのメモリを参照する，あるいは meta data の改ざん等により正規の実行者の意図しないプログラムが実行されるということは防げる.

7. 性 能 評 価

本節では実際に RISC-V Keystone に提案する Enclave キャッシュ機能を実装し，性能評価した結果を報告する. 測定環境は第 4 節同様 Hifive Unmatched によって行い, 10 回計測した平均値を取った. また, Enclave キャッシュ未実装時との比較のためハードウェアキャッシュ上に Enclave キャッシュを作成した際のキャッシュが残っていない状態での計測を行った. 測定項

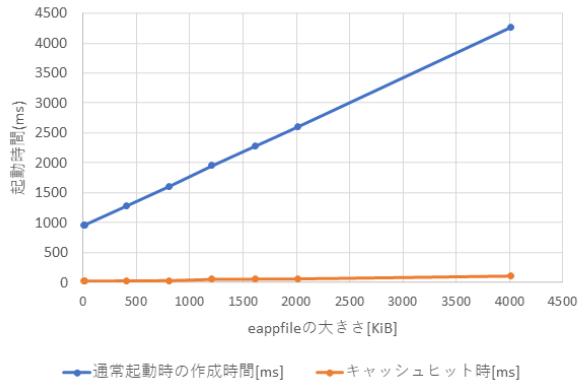

図 9　通常起動時とキャッシュヒット時の Enclave 起動時間比較

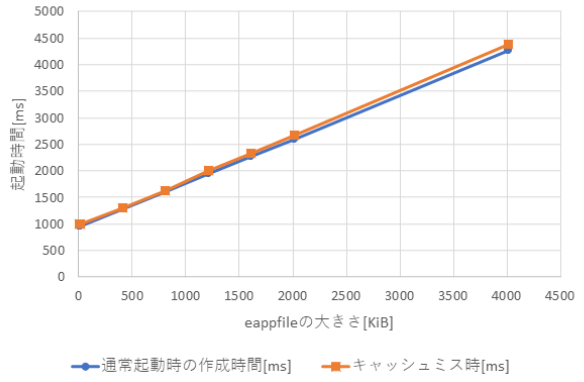

図 10　通常起動時とキャッシュミス時の Enclave 起動時間比較

目は, Enclave キャッシュがヒットしたときとキャッシュ機能未実装時 (通常起動時) の比較による速度向上率，及び Enclave キャッシュがミスしたときと通常起動時の比較によるキャッシュ機能実装におけるオーバーヘッドである.

7.1. 評価結果

評価結果を図 9, 10 にそれぞれ示す．図 9 では Enclave の通常起動時と Enclave キャッシュのヒット時の Enclave 起動時間を比較している. 図より，Enclave キャッシュにヒットしたときには約 40〜50 倍の速度向上率が得られることが分かる. また，表 10 では Enclave の通常起動時と Enclave キャッシュのミス時の Enclave 起動時間を比較している. 評価の結果から，Enclave キャッシュにミスした際にも起動速度は通常起動時と比べて 0.96〜0.97 倍であり，Enclave キャッシュを実装したことによるオーバーヘッドの増加はほとんど無いことが確認された.

通常起動時の Enclave 起動時間の約 96〜97.5% がハッシュ値計算やメモリマップ検証に使われていたため，Enclave キャッシュのヒット時には最大で 40 倍程度の速度向上率を見込んでいた. しかし実際には前小節に示した通り約 40〜50 倍の速度向上率が得られた. 予想より速くなった要因としてはメモリマップ作成時の関数呼び出し回数の減少や stack を作成しないこと，バイナリファイルをコピーする際に別ファイルを参照することなく, そしてドライバを経由することなくコピーを行えるといったことが挙げられる. これらの要因によって Enclave キャッシュヒット時の Enclave の起動時間は通常起動時の起動時間からハッシュ値，メモリマップ検証の時間を除いたよりも速くなったと考えられる.

また 810kiB から 1.2MiB に eappfile のサイズが変化したときのキャッシュヒット時の速度向上率は約 54.6 倍から約 39.1 倍と低下している. これはドライバによる Enclave 用のメモリ確保は 2 のべき乗ページで行われ，この間で確保されるメモリサイズが変化したためである. メモリ確保によるオーバーヘッドは Enclave の通常起動時には大きな差としてあらわれないが, ハッシュ計算をバイパスして高速化した結果全体に対するメモリ確保の影響が大きくなっている.

8. ま と め

本稿では RISC-V Keystone を対象とし，Enclave 起動の高速化の手法として Enclave キャッシュ機能を提案した. Enclave 起動におけるオーバーヘッドを調査し，ハッシュ計算やメモリマップ検証にかかる時間が全体の 96〜97.5% を占めることを確認した. 提案する Enclave キャッシュ機能を実装し評価した結果, Enclave ヒット時には通常起動時に比べて 40〜50 倍高速化されることが確認できた. また，Enclave キャッシュから Enclave を作成した際にはハッシュ計算やメモリマップ検証をバイパス出来るだけでなく，その他の手続きに関しても高速化されることも確認した. そして Enclave キャッシュのミス時にも Enclave の起動にかかる時間は通常起動時と比べて 0.96〜0.97 倍とほとんど変化せず，キャッシュ機能を搭載したことによるオーバーヘッドの増加はほとんど見られないことも確認できた.

謝辞　本研究の成果の一部は JSPS 科研費 JP23K11040 の助成を受けたものです.

文　献

[1] 飯田正樹，松田俊寛，永見健一，遠藤貴裕，古瀬正浩，“IoT をセキュアにする TEE 応用技術とその実用化への取り組み,” https://www.intec.co.jp/company/itj/itj17/contents/itj17_54-61.pdf. (Accessed on 06/17/2023).

[2] L. Dayeol, K. David, S. Shweta, S. Dawn, and A. Krste, “Keystone: An open framework for architecting tees,” arXiv preprint arXiv:1907.10119, pp.1–12, 2019.

[3] V. Costan and S. Devadas, “Intel SGX Explained,” IACR Cryptol. ePrint Arch., vol.2016, p.86, 2016.

[4] F. Erhu, L. Xu, D. Dong, Y. Bicheng, J. Xueqiang, X. Yubin, Z. Binyu, and C. Haibo, “Scalable Memory Protection in the {PENGLAI} Enclave,” 15th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 21), pp.275–294, 2021.

[5] H. Trong-Thuc, D. Ckristian, N.-H. Duc-Thinh, L. Duc-Hung, T. Akira, S. Kuniyasu, and P. Cong-Kha, “Quick Boot of Trusted Execution Environment with Hardware Accelerators,” IEEE Access, vol.8, pp.74015–74023, 2020.

[6] “Risc-v background,” https://docs.keystone-enclave.org/en/latest/Getting-Started/How-Keystone-Works/RISC-V-Background.html.

[7] A. Waterman, KrsteAsanovi ´ c, SiFive Inc., “The RISC-V Instruction Set Manual, Volume II: Privileged Architecture,” https://riscv.org/wp-content/uploads/2017/05/riscv-privileged-v1.10.pdf.

[8] S. Peter, “makeself - make self-extractable archives on unix,” https://makeself.io/.

[9] SiFive, “HiFive Unmatched Datasheet,” https://sifive.cdn.prismic.io/sifive/d0556df9-55c6-47a8-b0f2-4b1521546543_hifive-unmatched-datasheet.pdf.