修士論文

セキュアプロセッサを用いたマルチプロセッサシステムの構成

Multi-Processor System Design with Secure Processors

平成 29 年 02 月 03 日提出

指導教員

坂井 修一 教授

東京大学大学院 情報理工学系研究科
電子情報学専攻

48-146409　梶原　拓也

概要

近年, 様々な機密情報がコンピュータ上で処理されているが, これらの機密情報を不正に取得するための手段は年々多様化している. ソフトウェア自体の脆弱性をつくものに関してはソフトウェア作成者が注意することにより防ぐことができるが, サイドチャネル攻撃や OS やハイパーバイザなど高い権限を有するシステムソフトウェアによるメモリへの不正アクセスなど, プロセスより低いレイヤーを対象にした攻撃はソフトウェア作成者側で防ぐことができない.

これらの攻撃を防ぐには,OS やハイパーバイザ (HV) より低い層に位置するハードウェアによる対策をする必要がある. セキュアプロセッサは, 暗号化機構や命令のダミー発行機能などセキュリティ性を高めるための実装が付け加えられており, 上記の攻撃を防ぐことができる.

このような強力なセキュリティ機構を持っているが, セキュアプロセッサの普及は進んでいない. 普及が進まない原因の一つとして, 汎用プロセッサを用いたシステムに比べたときの, セキュアプロセッサを用いたシステムの実行速度の遅さが挙げられる. セキュアプロセッサはデータの暗号化や完全性検証をシングルプロセッサシステムにおいて行うことを想定しており, マルチプロセッサシステムでは正常に動作させることができない. よって, セキュアプロセッサを用いたシステムの性能向上を実現するには, シングルチップの性能向上に頼らざるを得ない.

本論文では, セキュアプロセッサを用いたマルチプロセッサシステムを構成するための手法として, セキュアプロセッサの相互認証手順と機密情報の暗号化通信手順, そして監視対象のメモリ範囲を分割して各セキュアプロセッサに割り当てて管理させることによる完全性検証手順を提案, 評価した. その結果, 記号論理的に本論文の提案手法が正しいことが確認された.

目 次

第 1 章　はじめに **5**

第 2 章　セキュアプロセッサの概要 **7**

2.1　機能要件 7

2.2　各機能詳細 8

2.2.1　暗号化機構 8

2.2.2　アクセス制御 8

2.2.3　完全性検証 9

2.2.4　ソフトウェア検証機構 10

2.2.5　タンパ検知 11

2.2.6　Oblivious RAM(ORAM) 12

2.3　普及への問題点 12

第 3 章　提案手法 **14**

3.1　セキュアプロセッサを用いたマルチプロセッサ方式 14

3.2　マルチプロセッサ方式実現への課題 14

3.3　プロセッサ間の認証 15

3.4　機密情報の通信 16

3.5　完全性の検証 17

第 4 章　評価 **24**

4.1　評価環境 24

4.1.1　ProVerif の概要 24

4.1.2　セキュアプロセッサ間の認証 25

4.1.3　データ読み込み時のハッシュ受け渡し手順 26

4.1.4　データ書き込み時のハッシュ受け渡し 27

第 5 章　考察 **30**

5.1　考察 30

第 6 章　関連研究 **31**
6.1　IntelSGX 31

第 7 章　おわりに **33**
7.1　本論文のまとめ 33
7.2　今後の課題 33

図 目 次

2.1 セキュアプロセッサの機能 7
2.2 PUF の動作 (from [15]) 8
2.3 ハッシュツリーアルゴリズム (from [15]) 10
2.4 AEGIS 完全性検証機構の設計図 (from [15]) 11
2.5 Ascend における相対的な性能低下量 (from [8]) 12

3.1 セキュアプロセッサ間の認証手順 17
3.2 マルチプロセッサシステムにおけるデータ完全性検証の失敗 18
3.3 キャッシュコヒーレンシプロトコルの導入による完全性検証のループ . . . 20
3.4 提案手法におけるメモリ読み込み手順 22

6.1 Intel SGX 実行時 (from [4]) 32

第1章　はじめに

近年, 医療画像や顧客情報など様々な機密情報がコンピュータ上で処理されているが, それに伴いこれらの機密情報を不正に取得するための手段は年々多様化している. ソフトウェア自体の脆弱性をつく攻撃手段に関してはソフトウェア作成者がそのような脆弱性を生み出さないよう注意することにより防ぐことができるが, プロセスより低いレイヤーを対象にした攻撃に対してソフトウェア作成者が対策を講じることはできない.

このようなソフトウェアより低いレイヤーに対する攻撃の例として, ハードウェアの物理的特性を利用したサイドチャネル攻撃が挙げられる. Yarom ら [17] が提唱したキャッシュへのアクセスタイミングを利用するサイドチャネル攻撃では, キャッシュ内のデータを全て追い出してから再度監視対象のコードを呼び出したときにかかる時間を計測することで他のプログラムが該当箇所の命令を実行したかを判別する. これにより, 暗号鍵を盗まれてしまう可能性がある. また,Amazon Web Service(AWS) のようなクラウドサービス上においては, 物理, 論理リソースの管理権限を悪意のある管理者が保持するという環境も考えられる. そのような環境下においては,OS やハイパーバイザ (HV) の権限を利用したメモリへの不正アクセス, プローブによるメモリ内の値の直接的な読み出しなどの攻撃が考えられる. これらの攻撃を防ぐには,OS や HV より低い層に位置する, ハードウェアによる対策をする必要がある.

セキュアプロセッサは, 暗号化機構や, 命令のダミー発行機能などのセキュリティ性を高めるための実装が付け加えられたプロセッサであり, 上記の攻撃を防ぐことができる. しかし, このような強力なセキュリティ機構を持っているにもかかわらず, セキュアプロセッサの普及は現状進んでいない. 普及が進まない原因の一つとして, 汎用プロセッサを用いたシステムに比べて, セキュアプロセッサを用いたシステムの実行速度が遅いことが挙げられる. セキュアプロセッサはデータの暗号化や完全性検証をシングルプロセッサシステムにおいて行うことを想定しており, マルチプロセッサシステムでは正常に動作させることができない. 例えば, あるセキュアプロセッサがすでに他のセキュアプロセッサが暗号化したデータを復号しなければならない場合に復号鍵をチップ内に持っていないため復号できない. また, 完全性検証機構に関しては, 他方のセキュアプロセッサが行ったメモリへの操作を改竄とみなしてしまう. これらの問題によりセキュアプロセッサを用いたマルチプロセッサシステムを実現することができず, セキュアプロセッサを用いたシステムの性能向

上を実現するには, シングルチップの性能向上に頼らざるを得ない.

本論文では, セキュアプロセッサ間におけるプロセスごとの暗号鍵のセキュアな交換手順および完全性検証対象のアドレスを分割する手法によって, セキュアプロセッサを用いたマルチプロセッサシステムを構成するための手法を提案する.

本論文では第二章において, セキュアプロセッサの機能要件と普及するにあたっての問題点を, これまでに提唱されてきている研究や製品を例として挙げつつ概観する. 第三章では、この問題点を解決するための手法としてセキュアプロセッサを, マルチプロセッサ方式として構成することを提案し, その実現方法を示す. 第四章において, プロトコル安全性検証ツールを用いた評価、また評価に対する考察を述べ, 第五章では関連研究を示す. 最後に第六章において本論文のまとめ, および今後の展望を述べる.

第2章　セキュアプロセッサの概要

2.1　機能要件

現在, セキュアプロセッサとして提唱されているものはそれぞれ異なる機能を有するが, その中でも一般的にセキュアプロセッサが備えているべき機能について紹介する.

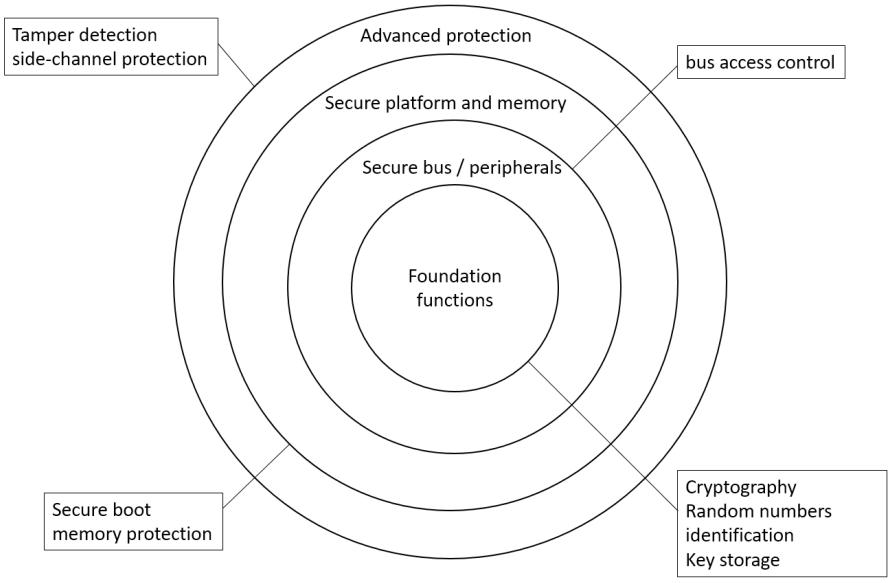


図 2.1: セキュアプロセッサの機能

ハードウェアが備えるべきセキュリティ機能は図 2.1 で示されているように多岐にわたる. 一般にセキュアプロセッサとして提唱・開発されているものは大きく分けてこれらのうちいくつかの機能だけを含むものであることが多い. それぞれの機能の詳細については後述するが, 大きく分けて 5 つの機能がセキュアプロセッサの提案においてよく見られる. プログラム・データをメモリ上においても暗号化するための暗号化機構, 他プロセスによりメモリ内容に不正にアクセスされることを防ぐためのアクセス制御機構, メモリ内容を改ざんされていないことを検証するためのメモリ完全性検証機構, ソフトウェアがユーザの想定しているものと同一のものか検証するためのソフトウェア検証機構, そして物理的な攻撃が行わたことの検知, またその対策を行うためのタンパ検知機構の 5 つである. 以下では, そのそれぞれの詳細について代表的な研究または製品を取り上げつつ, 概観する.

2.2 各機能詳細

2.2.1 暗号化機構

セキュアプロセッサは, プロセスの持つ機密情報を暗号化/復号する. セキュアプロセッサは OS や HV, 高い権限を与えられた他プロセスによる不正なアクセスからそれぞれのプロセスを守るという目的を果たすため, データをメモリ上にロードするときは暗号化した状態でロードする. この機構はほぼすべてのセキュアプロセッサに実装されている. これにより, バスにプローブをあてて信号を読み取るようなハードウェアタンパを防ぐことができる. プログラムの実行をする上でデータ・コードの暗号化・復号をするときはプロセスごとの暗号・復号鍵を用いる.AEGIS [16], [15] ではこの鍵をシリコン製造時の素子遅延特性のばらつきを利用した乱数生成器 PUF [12] を用いて, 乱数的に生成する.

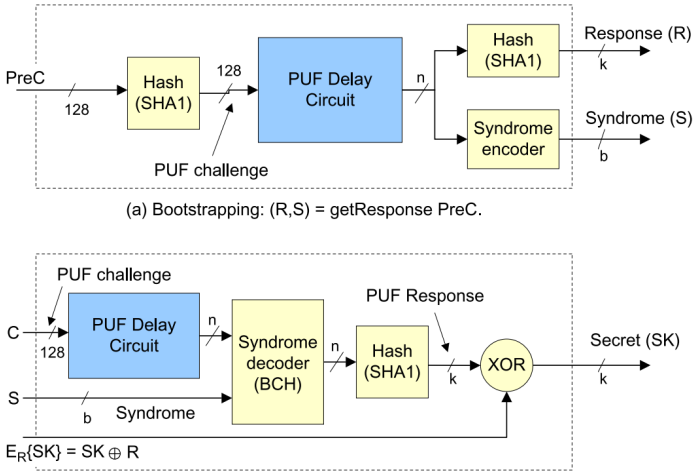


図 2.2: PUF の動作 (from [15])

2.2.2 アクセス制御

上述の暗号化機構によって, メモリ上への不正アクセスによる値の取得は防ぐことができるが, キャッシュ上には平文の状態でデータがのってしまう. これでは, 権限のあるプロセスがこのキャッシュ上のデータに対してアクセスすることでデータを取得できてしまう. このようなキャッシュへの不正アクセスを防ぐために一部のセキュアプロセッサではアクセス制御が取り入れられている.

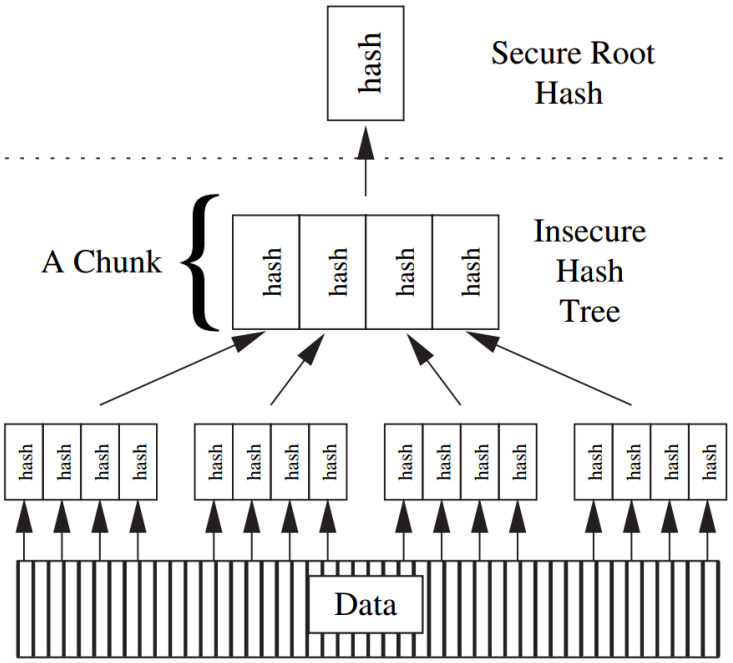
AEGIS [16] は MMU にアクセス権限のチェック機構を設けることでアクセス制御を実現している. また,Lie らの提案する XOM [11] は, それぞれのプロセスをコンテナという単位で保護しており, キャッシュラインに新たにコンテナ ID の識別ビットを追加することでアクセス制御を実現している. ARM の TrustZone [1] では,secure world と normal world の 2 つの実行モードを用意しており, 実行モードによってページテーブルのベースアドレスを保持するレジスタを切り替えることで, あるプロセスが他プロセスの保持するデータへアクセスすることを防ぐ.

2.2.3 完全性検証

セキュアプロセッサはこのように機密性を確保する手段だけではなく, 完全性を検証する手段も用意している. クラウドコンピューティングが普及している今日の情報社会においては, 処理が遠隔地で実行され, 悪意のある管理者がその実行基盤に対して物理的なアクセスを有している可能性があるため, 重要な機能である.

XOM [11] や AEGIS [15] はプログラム, データの完全性を検証するためにハッシュ値の演算機能を持つ. メモリへの書き込み時にあらかじめ計測しておいたデータのハッシュ値とメモリから読みだしたデータのハッシュ値を比較することでデータの完全性を検証する. XOM はプロセスごとのデータが改ざんされていないことを確認するために, データをメモリに書き込むときは書き込むデータにそのデータのハッシュ値を付け加える. しかし, この手法は以前にバス上に流れたデータブロックを盗聴し, 盗聴したデータをそれ以降の該当アドレスへの読み込みリクエストに対してあたかも正しいデータかのように返す再生攻撃に対して脆弱性を持つ. Suh らの提案する AEGIS [15] は, この脆弱性を克服するために Fig.2.3 のようにメモリを 1 つ以上のキャッシュライン分の領域をまとめたブロックごとに分けて, それぞれのハッシュ値に関するハッシュ木をつかって完全性の検証を行っている, AEGIS はハッシュ木の頂点である根の値をルートハッシュとしてプロセッサ内部に保持することで, どの部分に位置しているデータであってもルートハッシュまで該当データのハッシュ値とその親ノードの値の比較を繰り返すことで, データの完全性を検証することができる.

また,Suh らはこのハッシュ木による完全性の検証をできる限りキャッシュ内で行うことで完全性検証に関わるオーバヘッドを削減する手法を提案している [14]. この手法においては完全性の検証機構を図 2.4 のように設計することで完全性検証を遅延的に行う.



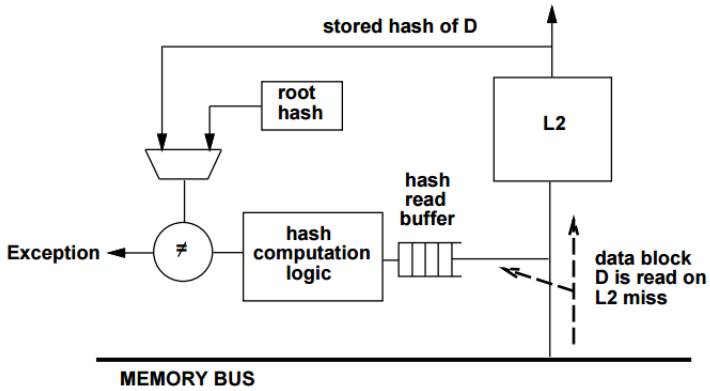
:

図 2.3: ハッシュツリーアルゴリズム (from [15])

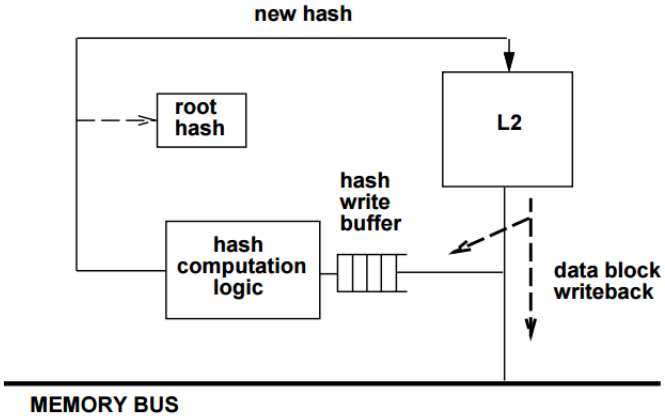
2.2.4 ソフトウェア検証機構

デジタル著作権管理 (DRM) 技術は暗号化などの手段を使うことで, デジタルデータを保護するがこれは OS が信頼できることを前提とした技術である. 悪意のあるエンドユーザが OS に改変を加えてユーザ認証を迂回することが考えられる. ソフトウェア検証機構はシステム上で実行されるソフトウェアが不正な改変を加えられていないことを検証するため, 不正に改変されたプログラムからエンドユーザを守るだけではなく, プログラムを不正に改変しようとするユーザからソフトウェアベンダを守ることも可能にする.

TPM はコンシューマ向けの PC にも組み込むことができるセキュアなコプロセッサである. システムを起動してから実行されるすべてのプログラムが正しいものであることを Trusted Boot と呼ばれるハッシュを利用した手法で検証する [10]. 具体的な手順としては, システム起動直後のブートローダのハッシュ値を計測し, そのハッシュ値と次に実行するプログラムのハッシュ値を合わせたもののハッシュ値をとる, といった手順をすべてのソフトウェアの実行に対して行うことで順次ハッシュ値の検証を行い, この手順をディスク全体のハッシュ値を取得するまで繰り返すことでソフトウェアは自らが実行されている環境が不正に改変されたものでないことを検証することができる.



(a)



:

図 2.4: AEGIS 完全性検証機構の設計図 (from [15])

欠点として, 最初に実行されるブートローダを信頼しなければいけないのにもかかわらず, そのブートローダは一般的に Flash ROM に書かれていることが一般的であり, ユーザによって書き換えが可能である点である.

2.2.5 タンパ検知

高価な機材を保持している攻撃者であれば, プロセッサのパッケージを取り外しチップ内に格納されている機密情報を盗み取ることができる. IBM4765 [7], [13] はこのような物理的な攻撃からシステムを守るためのセキュアなコプロセッサである. 他のセキュアプロセッサと同様に暗号化機構を有するが, ファラデーケージを利用して, 物理的な攻撃の際に生じる電磁波を検知し, データを復号するための秘密鍵を攻撃者によって取得する前に破

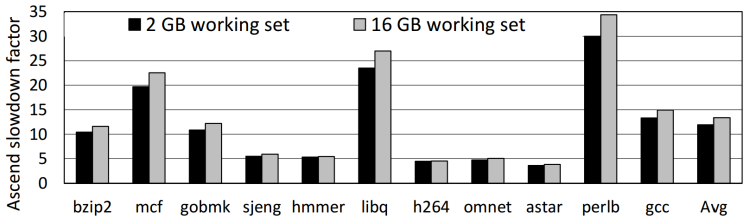


図 2.5: Ascend における相対的な性能低下量 (from [8])

壊することでデータを秘匿する.

2.2.6 Oblivious RAM(ORAM)

Ascend [8] はリモート環境においてセキュアコンピューティングを提供するために提唱されたセキュアプロセッサであり, 唯一サイドチャネル攻撃を防ぐことができる. 命令の実行をする際にダミーの命令も発行することにより, 処理の流れが悪意のある攻撃者に知られてしまうことがない. また, ダミー命令を実際に実行していることにより, 電力解析によるサイドチャネル攻撃も防ぐことができる.

欠点として, 実行速度が極端に低下してしまうことが挙げられる.Fletcher らによると図 2.5 のように平均で 10 倍以上の実行の遅延が発生している [8]. また, この実行結果はシングルチップにおけるベンチマーク実行の結果であるので,OS の上でプログラムを動かせる際はさらに時間がかかることが予想される.

2.3 普及への問題点

セキュアプロセッサはこのようにハードウェアタンパ, ソフトウェアタンパのどちらも防ぐことができるが, 実用上の問題を抱えている.

まず一つは, メモリ, プロセス管理の方法が変わることから該当部分に関わる OS のコードを改変しなければならないことが挙げられる.

次にもう一つは, プロセス切り替えの際に発生する鍵の切り替え処理, メモリへの操作を行う際に発生するデータの暗号化処理と復号処理やデータの完全性検証処理, そして Oblivious ＲＡＭに関わる処理などにおいて発生するオーバヘッドによる実行時間の増加の問題がある.

上記で取り上げたセキュアプロセッサにおいて, 最もオーバヘッドが大きいものは Ascend でベースラインの平均で 10 倍以上の遅延時間の増加を示している. それに対して,AEGIS は完全性を検証するエリアと暗号化・復号処理を行うエリアの分割やキャッシュ上でのハッ

シュツリーの構築をすることにより実行時間の増加を 130%まで抑えている. しかし, これらの計測結果はすべてシングルチップを用いたシステム同士での比較であり, 現状セキュアプロセッサを用いたシステム全体のパフォーマンスを向上させるにはシングルチップのパフォーマンス向上に頼らざるを得ない. セキュアプロセッサを用いたシステムをマルチプロセッサシステムとして構築できるように対応させない限り, 汎用プロセッサを用いたマルチプロセッサシステムとの実行速度の差はさらに広がることが予想される.

第3章　提案手法

3.1　セキュアプロセッサを用いたマルチプロセッサ方式

前項までに述べたように, セキュアプロセッサはセキュリティ性を担保しつつプログラムの実行を行うためにセキュアプロセッサではない汎用プロセッサに比べて実行時間が最大で 10 倍以上にまで増加している. この実行速度の増加は普及を妨げる要因の一つになり得る.

本研究では, 実行速度の向上をはかるために, セキュアプロセッサを用いてマルチプロセッサ構成のシステムを構築することを提案する. これによりセキュリティ性を保ちつつ, 並列計算や独立したタスクのチップごとの実行を可能にする.

3.2　マルチプロセッサ方式実現への課題

現状, セキュアプロセッサの研究は AEGIS [15],Ascend [8] などシングルチップでの構成を想定しているものだけである. セキュアプロセッサをマルチプロセッサシステムに組み込む場合, セキュアプロセッサ単体では実現できた機能のいくつかが意図されている通りに動作しない.

たとえば, 自身が暗号化したデータの復号は同一セキュアプロセッサにしかできない. これはプロセスを生成したプロセッサが, そのプロセスに対する鍵をチップ内部で生成することによる. セキュアプロセッサをマルチプロセッサ方式に組み込む場合, あるセキュアプロセッサがデータを読み込む際, 他セキュアプロセッサがすでに該当データを暗号化していた場合はデータを復号できず正常に処理をすることができない.

この問題を解決するには, 暗号化を施したセキュアプロセッサから復号したいプロセッサへ鍵の受け渡しをする必要があるが, 鍵を平文の状態で受け渡してしまうと攻撃者に鍵を入手される可能性がある. よってまず鍵をリクエストしているプロセッサがセキュアプロセッサであることを認証し, その鍵を暗号化/復号機構を通したうえで受け渡す必要がある.

また,2 つ以上のセキュアプロセッサが同じアドレスのデータへの書き込みをする場合, データ完全性検証機構は正常に動作しない. これもセキュアプロセッサがシングルチップでの実装を想定していることによる. マルチプロセッサシステムに対応していないセキュアプロセッサは自身が動作しているシステム上にはメモリへの操作を行う要素が自身の他

に存在しないという前提でデータの完全性検証を行うため, たとえ他のセキュアプロセッサが同じシステム上のメモリに書き込みを行っている場合であっても, 攻撃者によるメモリの改ざんと判定してしまう. よってデータ完全性検証機構を, マルチプロセッサシステムに対応させるために改変を加える必要がある.

これら改変が必要な機能に対して, アクセス制御はキャッシュ内に平文状態で保持されているデータに対するアクセスをプロセスごとに制御するものであることから, スケジューラによってプロセスを実行するチップが決定される度にキャッシュ内容をフラッシュすればよい.

また、ソフトウェア検証機構に関しては最初の起動時に行われるものであるから, マルチプロセッサシステムにおいても実行途中に問題が生じることはない. もし他のセキュアプロセッサによるプログラムロード時のソフトウェア検証の結果、改ざんが検知されていた場合は該当ソフトウェアから生成されるプロセスは停止させられる. セキュアプロセッサではない他の汎用プロセッサがソフトウェア検証をせずにプログラムをメモリ上にロードした場合においては, スケジューラによって該当プロセスの実行主体がセキュアプロセッサに移った際にメモリへの改竄として検知される.

タンパ検知に関してはそれぞれのパッケージで独立して実装することが可能であるため, マルチプロセッサシステム特有の問題が発生することはない.

以上をまとめると, セキュアプロセッサをマルチプロセッサ方式として構成するには以下の 3 点を解決する必要がある.

- セキュアプロセッサ間における相互認証
- セキュアプロセッサ間における機密情報の通信
- 各セキュアプロセッサによる完全性の検証

次節では, これらの課題に対する解決法を提案していく.

3.3 プロセッサ間の認証

セキュアプロセッサを用いたマルチプロセッサ構成において, あるプロセスが持つデータを 2 つ以上のプロセッサ上において動作させる場合, 該当データは最初に該当プロセスを生成したセキュアプロセッサによって暗号化されているため, 該当データを復号するための鍵（データ鍵）をプロセッサ間で共有する必要がある. データ鍵はデータそのものと同じく機密情報であるから, セキュアプロセッサは自身の通信相手がセキュアプロセッサであることが保証されていなければデータ鍵を渡してはならない. そのため, データ鍵の共有を行う前に通信相手の認証を行う必要がある. 提案手法においては,Ascend [8] のような,

プロセッサごとに固有のプロセッサメーカーによって電子署名を施された公開鍵と秘密鍵のペアを持っているセキュアプロセッサを想定する. プロセッサメーカーによる公開鍵への署名は、それを持っているプロセッサがセキュアプロセッサであることを確認することを可能にする. この鍵のペアを利用してセキュアプロセッサ同士の認証を行う. 認証は図 3.1 の手順で行う.

1. CPU1 が CPU2 に, プロセッサメーカーによって電子署名を付与された CPU1 の公開鍵を送信する.
2. CPU2 は電子署名によって CPU1 の公開鍵が正しいことを確認した後,CPU1 と機密情報を通信するための共通鍵を生成する.
3. CPU2 は生成した共通鍵を, 以下の順番で暗号化する.
   (a) CPU2 の秘密鍵で暗号化する
   (b) CPU1 の公開鍵で暗号化する
4. CPU2 は暗号化した共通鍵とプロセッサメーカによって電子署名を付与された CPU2 の公開鍵を送信する.
5. 暗号化された鍵を受け取った CPU1 は,CPU2 の公開鍵が正しいことを確認した後,CPU1 の秘密鍵,CPU2 の公開鍵によって共通鍵を得る

この手順は SSL プロトコル [9] における鍵の生成と共有を参考に, 互いに相手を認証しなければならないということから 3 の手順において鍵を生成した側も自身がセキュアプロセッサであることを証明する. この手順であれば, システム上のセキュアプロセッサの公開鍵を傍受し再送信することによって実行されるリプレイ攻撃のような攻撃も防ぐことができ, 共有バス上においてもセキュアに通信相手を認証することができる.

3.4 機密情報の通信

セキュアプロセッサはそれぞれのプロセスごとのデータを機密情報と捉えており, これをプロセスごとの暗号鍵で暗号化した状態でメモリ上に展開している. プロセスごとの暗号化されたデータに対応する復号鍵も, それを用いることで平文のデータを取得することができてしまうため機密情報である. 本研究では, この復号鍵の配送を, 認証した際に生成した共通鍵を用いて行うことを提案する. これにより, あらかじめ認証しあった 2 つのセキュアプロセッサ以外には復号鍵を盗聴されることはない. 受け取ったデータ鍵はチップ内部に保持することで, シングルチップでの動作時と同様に鍵管理機構を利用したプロセ

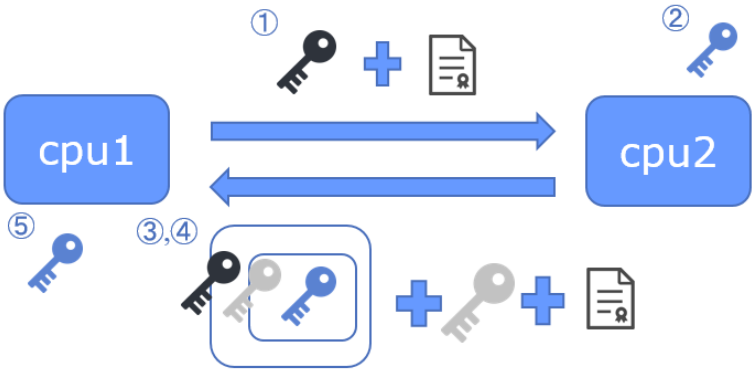


図 3.1: セキュアプロセッサ間の認証手順

スごとの自動的な暗号化・復号が可能になる. また, 後述する完全性検証の手順においてセキュアプロセッサ間で送受信するデータのハッシュもこの共通鍵を用いて行う.

3.5 完全性の検証

第 2 章で述べたように, 一部のセキュアプロセッサはメモリへの改ざん検知をハッシュを用いたデータの完全性検証機構によって実現している. これには,XOM のようにハッシュをデータとともにメモリに書き込むものと,AEGIS のように 1 つ以上のキャッシュラインを束ねたデータチャンクを葉とするハッシュ木を用いるものがある. XOM のようにデータ自体にハッシュを付け加えてデータの完全性を検証する場合, リプレイ攻撃によって以前にメモリに書き込んだ値を最新の値として誤認識させられる可能性があるため本提案におけるデータの完全性検証機構は AEGIS [15] と同じくハッシュツリーをモデルとする.

ハッシュツリーを用いたデータの完全性検証機構はキャッシュ内にハッシュを保持するかしないかの 2 つの実現方法が考えられる. メモリのすべてのデータに対するハッシュ木をメモリ上に構築する場合, マルチプロセッサシステムにおいては検証の度にハッシュ木のノードを取得しなければならないことによる大幅な速度低下だけではなく, 異なるセキュアプロセッサが同じアドレスのデータに対して操作を加えるとき, 図 3.2 のように改竄の誤検出が起こり, 完全性検証処理の失敗が発生する可能性がある. 下図においてはお互いのセキュアプロセッサはすでに初期状態のメモリのハッシュツリーの計測を終えているとする.

1. CPU2 がメモリからデータを読み込む
2. CPU2 がキャッシュ上のデータに変更を加えて, メモリに書き出す

3. CPU1 が書き戻されたデータと同じデータを読み出す
4. CPU1 が保持しているハッシュ木は CPU1 が変更する前のデータを元に計測したものであるため, 改ざんが誤検出される.

しかし,AEGIS の完全性検証機構はできるだけキャッシュ内にハッシュ木のノードを保持することで, 完全性検証処理の高速化をはかっている. この手法の副次的な効果として, マルチプロセッサシステムにセキュアプロセッサを組み込んだ場合においてもキャッシュコヒーレンシプロトコルを介してハッシュ木のノードの値を同期することができ, 図 3.2 のような検証処理の失敗を防ぐことができる.

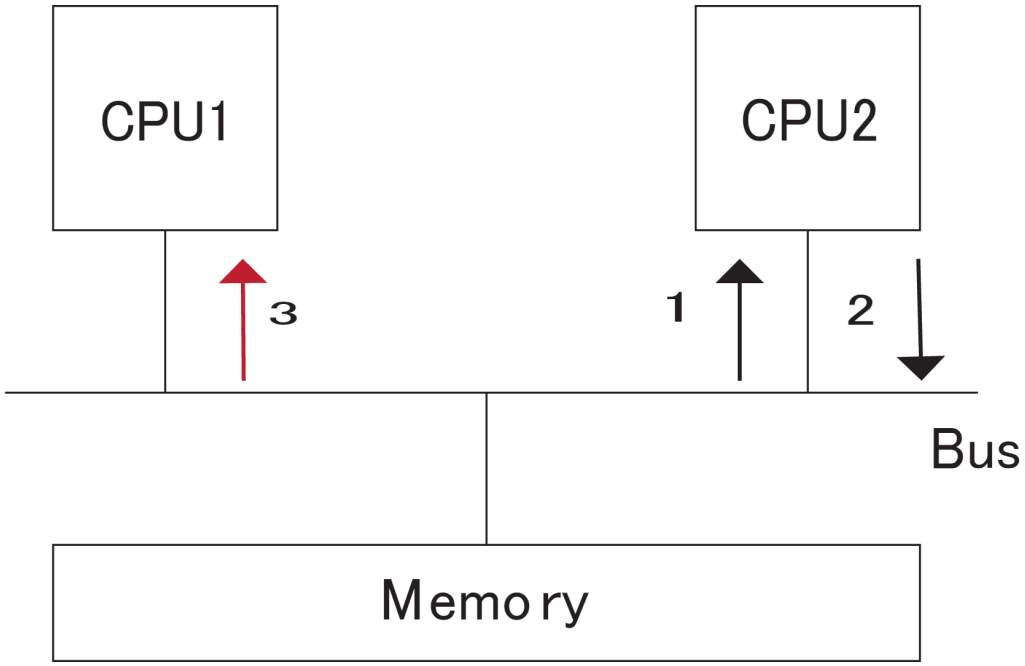


図 3.2: マルチプロセッサシステムにおけるデータ完全性検証の失敗

これは MESI プロトコル [3] や MOESI プロトコル [6] のようなキャッシュコヒーレンシプロトコルによって, 互いのキャッシュの内容に不整合が起きないよう, 自身の持つキャッシュ内のデータの中で他のプロセッサによって改変されている値が無効化されるからである. しかし, キャッシュコヒーレンシプロトコルはキャッシュ内容の不整合を防ぐかわりに, アクセス数の増大を招く. 具体的には, 以下の場合において Suh らが提唱しているハッシュ木アルゴリズム [14] の実装に比べキャッシュからメモリへの書き込み回数がハッシュ木の

高さの分だけ増加する. 以下の例では,CPU1,CPU2 はともに同じアドレスのデータチャンクと該当アドレスに対応するハッシュ値を持っているものとする

1. CPU2 がデータチャンクへの書き込みを終えて, キャッシュからの追い出しによってデータチャンクをメモリに書き込む

2. CPU2 はデータチャンクの親ノードの値に, データチャンクのハッシュ値を計測したものを繰り込み,CPU1 に改変した親ノードを含むキャッシュラインの invalidate 要求を送り,CPU2 は改変した親ノードを含むキャッシュラインを modify 状態に変更する

3. CPU1 がデータチャンクを書き込む際, 親ノードの値をデータチャンクのハッシュ値に改変しなければならないが,CPU1 のキャッシュ内にある親ノードは無効化されているため, CPU2 に親ノードの値を書き戻し要求を送る

4. CPU2 は親ノードの値を書き戻すとともに, さらに 1 階層上の親ノードの値を改変し,CPU1 に改変した親ノードを含むキャッシュラインの invalidate 要求を送る

5. ルートハッシュにたどり着くまで,4 と 5 の手順を繰り返す

図 3.3 のように, 先にメモリの値の改変を行ったセキュアプロセッサ側においては, キャッシュ内ハッシュツリーに対応するキャッシュラインはデータの書き出しとその親ノードの値の改変に伴う他セキュアプロセッサへの invalidate 要求のループが発生し, 後からメモリ内の改変されたデータを読み込むセキュアプロセッサ側においては,modify 状態にある他セキュアプロセッサのキャッシュラインの書き戻し要求と親ノードのキャッシュラインの invalidate のループが発生する. Suh らによるキャッシュ内でのハッシュ木構築手法 [14] においては, 完全性検証を行うためのハッシュ木のノードを取得に伴うメモリアクセスは平均 1 回以下であるが, 上記のようなメモリアクセスのループが発生してしまった場合, CPU2 はハッシュ木の高さの分だけの読み込みと書き込みが発生し,CPU1 ではハッシュ木の高さの分だけの読み込みが発生してしまい, 大幅なアクセス遅延を引き起こす.

そこで, 本論文ではプロセッサ数でメモリ領域を分割し各プロセッサに分割されたメモリ領域を 1 つずつ割り当て, 各プロセッサは, 自らが割り当てられた領域におけるデータの完全性のみを管理することを提案する. それぞれのセキュアプロセッサが自身のアクセスするメモリアドレスの値が改ざんされていないことを確認するときは, メモリアドレスに関わらずまず自身のキャッシュ内にあるハッシュ木を用いて完全性検証を行い, 改竄検出がなされた場合は該当アドレスを管理するセキュアプロセッサにリクエストを送り, アクセスするアドレスのデータの親ノードの値を受け取る. ハッシュ木のキャッシュラインに対してキャッシュコヒーレンシプロトコルの処理を適用してしまうと先述の問題が発生するた

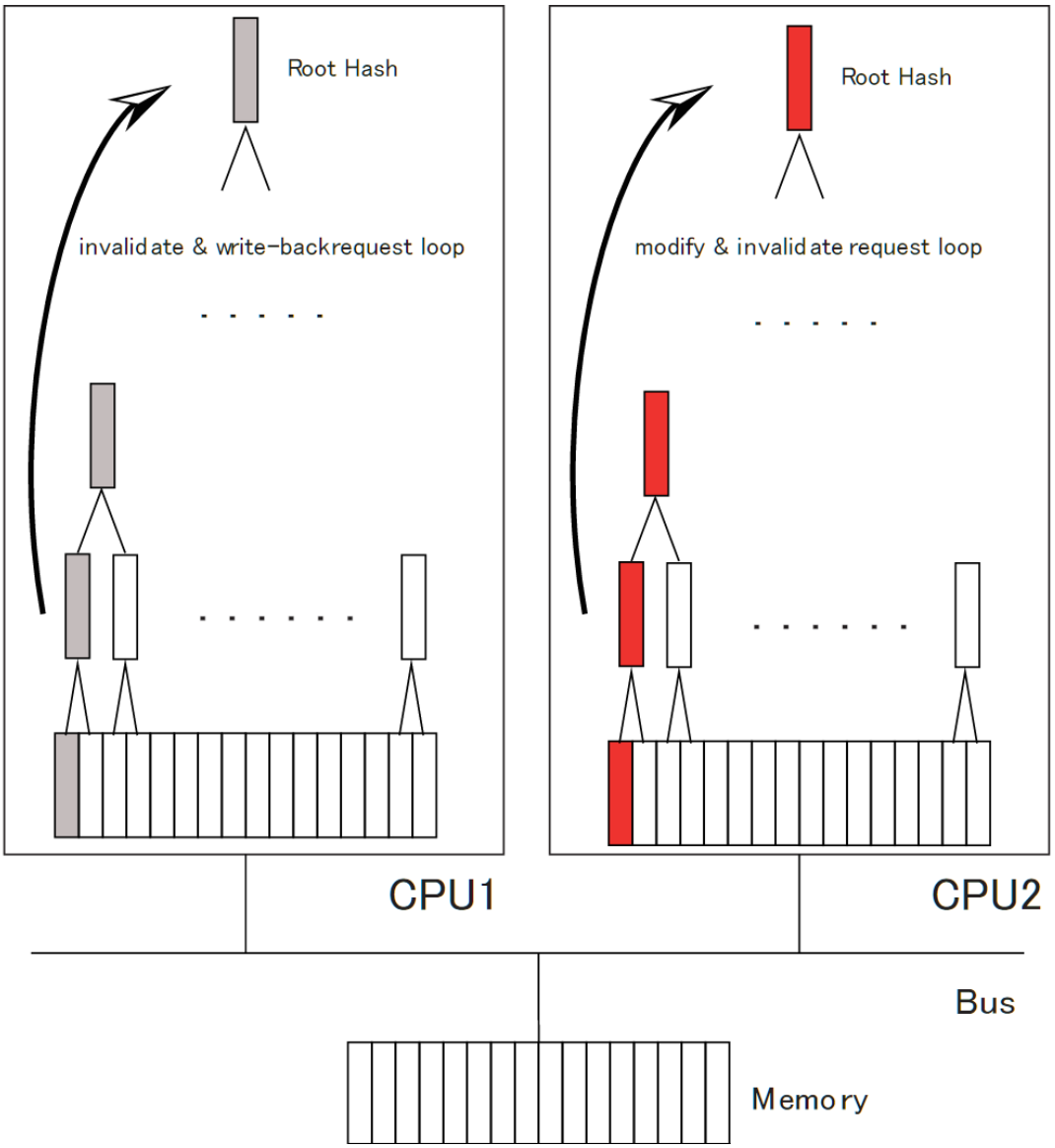


図 3.3: キャッシュコヒーレンシプロトコルの導入による完全性検証のループ

め, この処理はキャッシュコヒーレンシプロトコルによる処理に先立って行われる. 具体的な読み込み, 書き出し命令の際の完全性検証の手順について以下に記す.

- 読み込み手順

  1. メモリからデータを読み込む
  2. 読み込み命令を実行しているセキュアプロセッサのハッシュ木を用いて, 読み込んだデータの完全性の検証をする
  3. 手順 2 が成功した場合は, プログラムの実行を続行する
  4. 手順 2 が失敗した, かつデータのアドレスが命令を実行しているセキュアプロセッサの監視しているメモリ領域の範囲内である場合, メモリの改ざんが行われたと判定する
  5. 手順 2 が失敗した, かつデータのアドレスが命令を実行しているセキュアプロセッサの監視しているメモリ領域の範囲内にない場合, 該当アドレスを監視しているセキュアプロセッサにアクセスするメモリアドレスを送る
  6. リクエストを受けたセキュアプロセッサは該当アドレスのハッシュ値をプロセッサ間の共通鍵で暗号化した状態で返す
  7. 読み込み命令を実行しているセキュアプロセッサは受け取ったハッシュを用いて完全性検証を行う
  8. 手順 7 が成功した場合は, キャッシュ内のハッシュ木に受け取ったハッシュの値を反映させるとともにプログラムの実行を続行し, 失敗した場合は改ざんとして検知する

  読み込むデータが自身の監視しているメモリ領域内に格納されていない場合においても, 一度自身のハッシュ内のキャッシュ木を用いて完全性検証を行うことで他のセキュアプロセッサが改変を加えることが少ないメモリへのアクセスの際のオーバヘッドを削減することができる.

- 書き出し手順

  1. データを書き出すアドレスが書き出し命令を実行するセキュアプロセッサの監視しているメモリ領域の範囲内か確認する
  2. 範囲内であった場合は自身のハッシュ木の修正を行った後に他のセキュアプロセッサへ該当アドレスのデータに対応するキャッシュラインの無効通知を送り, データを書き出し, プログラムの実行を続行する

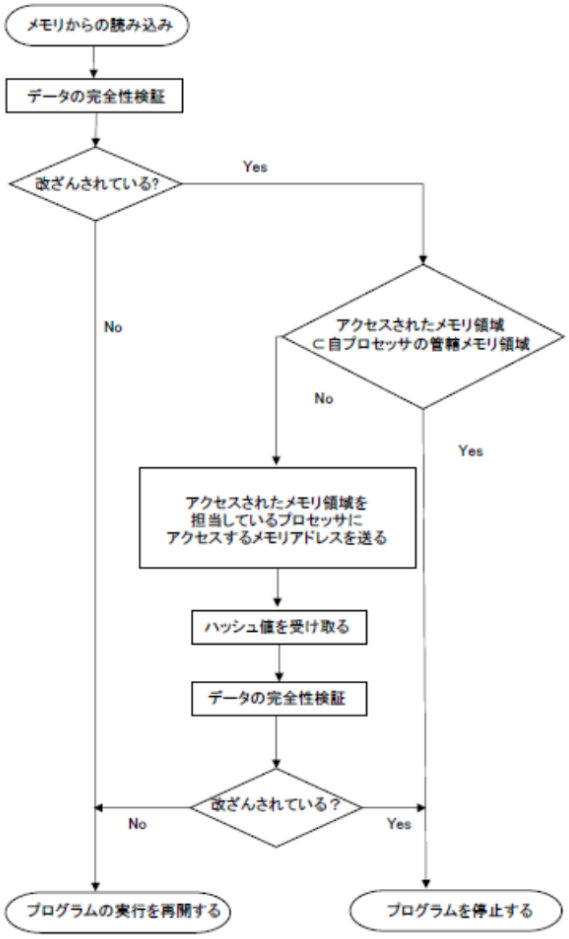


図 3.4: 提案手法におけるメモリ読み込み手順

3. 範囲外であった場合は, そのアドレスを監視しているセキュアプロセッサにハッシュ木修正のリクエストと変更するデータのアドレスとハッシュ値をプロセッサ間通信用の共通鍵で暗号化した状態で送る
4. リクエストを受けたセキュアプロセッサは, 該当アドレスのデータに対応するハッシュを自身のキャッシュに格納した後に, キャッシュ内のハッシュ木の修正をした後に該当ハッシュを共通鍵で暗号化したものをリクエスト元のセキュアプロセッサに返す
5. 結果を受けたセキュアプロセッサはキャッシュ内においてデータの書き出しを行う

上記の処理は, 書き込みタイミングの差異によるデータの不整合を防ぐため, トランザクションとして行う必要がある.

第4章　評価

4.1　評価環境

本研究における提案手法の安全性を評価するために, INRIA が開発したプロトコル安全性自動検証ツールである ProVerif [2] を用いる. 本論文では,Proverif を用いてセキュアプロセッサ間の相互認証とプロセッサ間通信用の共通鍵の生成, 交換, そして完全性検証手法におけるハッシュツリーの受け渡しについて検証を行う。

4.1.1　ProVerif の概要

ProVerif は想定する環境を記号論理としてモデル化し、それぞれのエンティティに対して属性と振る舞いを記述することで、悪意のある攻撃者による指定したデータへの到達可能性、エンティティ間の相互認証、等価観測性を検証することができる. Proverif では Listing 4.1 のように検証するシナリオ中に利用する型の宣言を行い, 関数宣言とリダクション演算の定義により暗号プリミティブの定義を行うことができる. 検証モデルにおけるチャネルの定義, 攻撃者が初期状態で持っている情報の定義, そして, 検証したいそれぞれのイベントについて定義し, 構築した検証モデルにおいて攻撃者が秘匿したい情報に到達し得るか, またあるイベントが発生し得るかを様々な攻撃ベクトルから検証することができる.

Listing 4.1: 検証コードにおける属性設定部分

```
(* type info *)
type sigSK.
type sigPK.
type asynSK.
type asynPK.
type synKey.
type nonce.
type hash.

(* type converters *)
fun synKey2bs(synKey): bitstring [data, typeConverter].
reduc forall c: synKey; bs2synKey(synKey2bs(c)) = c.

fun asynPK2bs(asynPK): bitstring [data, typeConverter].
reduc forall c: asynPK; bs2asynPK(asynPK2bs(c)) = c.

(* symmetric cryptography *)
```

```
fun synEnc(bitstring, synKey): bitstring.
reduc forall m: bitstring, k: synKey; synDec(synEnc(m, k), k) = m.

(* asymmetric cryptography *)
fun getAsynPK(asynSK): asynPK.

fun asynPEnc(bitstring, asynPK): bitstring.
reduc forall m: bitstring, ssk: asynSK; asynSDec(asynPEnc(m, getAsynPK(ssk)), ssk) = m.

fun asynSEnc(bitstring, asynSK): bitstring.
reduc forall m: bitstring, ssk: asynSK; asynPDec(asynSEnc(m, ssk), getAsynPK(ssk)) = m.

(* secrets *)
free sp1AsynSK: asynSK [private].
free sp2AsynSK: asynSK [private].
free vendSpSigSK: sigSK [private].
free secretContent: bitstring.

(* channels *)
free SP1toSP2, SP2toSP1, memToSP1, memToSP2: channel.

(* Secrecy assumptions *)
not attacker(sp1AsynSK).
not attacker(sp2AsynSK).
not attacker(vendSpSigSK).

(* events *)
event verificationBySP1Success().
event verificationBySP2Success().
event contentTamper().
event falseNegative().
event falsePositive().

(* query *)
query event(verificationBySP1Success).
query event(verificationBySP2Success).
query event(falseNegative).
query event(falsePositive).
query event(contentTamper).
query attacker(secretContent).
```

4.1.2 セキュアプロセッサ間の認証

まず, セキュアプロセッサ間における認証手順の安全性を検証した. 以降の検証コード内において, 第 3 章における CPU1 を sp1,CPU2 を sp2 とする. 以下にコードを示す.

Listing 4.2: 認証手順検証コード

```
let sp1(sp1AsynSK: asynSK, vendSpSign: bitstring, vendSpSigPK : sigPK) =
  let sp1AsynPK = getAsynPK(sp1AsynSK) in
  out(SP1toSP2, (sp1AsynPK, vendSpSign));
  in(SP2toSP1, sp2AuthSetPack: bitstring);
```

```
5     let (sp2AsynPK: asynPK, vendSp2Sign : bitstring, n : nonce, authSet: bitstring) =
          sp2AuthSetPack in
6     if verifSign(vendSp2Sign, vendSpSigPK) = asynPK2bs(sp2AsynPK) then
7       event verificationBySP1Success();
8
9   let sp2(sp2AsynSK: asynSK, vendSpSign: bitstring, vendSpSigPK : sigPK) =
10    (* authentication *)
11    in(SP1toSP2, sp1Crt: bitstring);
12    let (sp1AsynPK: asynPK, vendSp1Sign: bitstring, vendSpSigPK: sigPK) = sp1Crt in
13    if verifSign(vendSp1Sign, vendSpSigPK) = asynPK2bs(sp1AsynPK) then
14      event verificationBySP2Success();
15    new n: nonce;
16    let sp2AsynPK = getAsynPK(sp2AsynSK) in
17    new commonKey : synKey;
18    event sndKey(commonKey);
19    out(SP2toSP1, (sp2AsynPK, vendSpSign, n, asynPEnc(asynSEnc((n, commonKey), sp2AsynSK)
          , sp1AsynPK)));
```

以上のコードを検証した結果, 得られた結果の一部を図 4.3 に示す.
RESULT not event(verificationBySP2Success is false 文により片方のセキュアプロセッサ (SP2) によるもう片方のセキュアプロセッサの認証が正しく行われることが記号論理的に証明された. 同様に逆方向の認証も正しく行われていることがわかる.

Listing 4.3: 認証手順検証における検証結果出力

```
1 The event verificationBySP2Success is executed.
2 A trace has been found.
3 RESULT not event(verificationBySP2Success) is false.
4
5 The event verificationBySP1Success is executed.
6 A trace has been found.
7 RESULT not event(verificationBySP1Success) is false.
```

4.1.3 データ読み込み時のハッシュ受け渡し手順

次にデータ読み込み時のハッシュ受け渡し手順の安全性を検証する. 攻撃者からの攻撃を検知できるか, そうでない場合は正常であると判断して処理を続行できるかを評価する. また, ハッシュ受け渡しリクエストを受けたセキュアプロセッサが行う完全性検証はキャッシュ内に格納されているハッシュノードだけで完結するものと仮定する. そうでない場合においても, シングルチップでのセキュアプロセッサの完全検証作業の安全性は担保されており, リクエストを受けたプロセッサがシングルチップでの実行時における完全性検証を行うだけでよいことから, 前述の仮定による安全性への影響はない.

Listing 4.4: データ読み込み時のハッシュ受け渡し手順

```
1 let sp1(sp1AsynSK: asynSK, vendSpSign: bitstring, vendSpSigPK : sigPK) =
2   (* hash tree read request *)
```

```
3   in(memToSP1, test: bitstring);
4   new reqAddr : bitstring;
5   out(SP1toSP2, reqAddr);
6   in(SP2toSP1, integrityInfo: bitstring);
7   let parentHash:bitstring = synDec(integrityInfo, commonKey) in
8
9   if bs2hash(parentHash) <> hashC(test) then
10    event contentTamper;
11  if (bs2hash(parentHash) = hashC(test)) && (secretContent <> test) then
12    event falseNegative;
13  if (bs2hash(parentHash) <> hashC(test)) && (secretContent = test) then
14    event falsePositive.
15
16 let sp2(sp2AsynSK: asynSK, vendSpSign: bitstring, vendSpSigPK : sigPK) =
17  (* hash tree read request *)
18  in(SP1toSP2, reqAddr: bitstring);
19  let (reqAddr: bitstring) = reqAddr in
20  out(SP2toSP1, (synEnc(hash2bs(hashC(secretContent)), commonKey))).
21
22 let mem() =
23  new memData : bitstring;
24  out(memToSP1, secretContent).
```

図 4.4 のコードを検証した結果, 図 4.5 が出力された. これによりコンテンツの改竄 (contentTamper) を検知することができ, その検知には偽陽性 (falsePositive) はなく, 検知漏れ (falseNegative) もないことが証明された.

Listing 4.5: データ読み込み時のハッシュ受け渡し手順の検証結果出力

```
1  The event contentTamper is executed.
2  A trace has been found.
3  RESULT not event(contentTamper) is false.
4  -- Query not event(falsePositive)
5  Completing...
6  ok, secrecy assumption verified: fact unreachable attacker(sp1AsynSK[])
7  ok, secrecy assumption verified: fact unreachable attacker(sp2AsynSK[])
8  ok, secrecy assumption verified: fact unreachable attacker(vendSpSigSK[])
9  Starting query not event(falsePositive)
10 RESULT not event(falsePositive) is true.
11 -- Query not event(falseNegative)
12 Completing...
13 ok, secrecy assumption verified: fact unreachable attacker(sp1AsynSK[])
14 ok, secrecy assumption verified: fact unreachable attacker(sp2AsynSK[])
15 ok, secrecy assumption verified: fact unreachable attacker(vendSpSigSK[])
16 Starting query not event(falseNegative)
17 RESULT not event(falseNegative) is true.
```

4.1.4 データ書き込み時のハッシュ受け渡し

最後に, データ書き込み時におけるハッシュ受け渡し手順の安全性を検証する. 通信路におけるデータの改ざんを検知できるか, また通信相手のセキュアプロセッサからのハッシュ

が改竄されていない場合は誤検知をしないかを評価する. また, ハッシュツリーはキャッシュ内に保持されることを想定しているため, ハッシュを受け取ったセキュアプロセッサが正しくメモリに書き込みを行えるかに関しては考慮しない.

Listing 4.6: データ書き込み時のハッシュ受け渡し

```
let sp1(sp1AsynSK: asynSK, vendSpSign: bitstring, vendSpSigPK : sigPK) =
  (* hash tree write request *)
  new reqWriteAddr : bitstring;
  new writeData : bitstring;
  out(SP1toSP2, (synEnc((reqWriteAddr, hash2bs(hashCal(writeData))), commonKey)));
  in(SP2toSP1, writeStatusInfo : bitstring);
  let (returnedHash : bitstring) = synDec(writeStatusInfo, commonKey) in
  if bs2hash(returnedHash) <> hashCal(writeData) then
    event contentTamper.

let sp2(sp2AsynSK: asynSK, vendSpSign: bitstring, vendSpSigPK : sigPK) =
  (* hash tree write request *)
  in(SP1toSP2, integrityInfo: bitstring);
  let (reqWriteAddr : bitstring, dataHash: hash) = synDec(integrityInfo, commonKey) in
  out(SP2toSP1, synEnc(hash2bs(dataHash), commonKey)).
```

図 4.6 のコードを追記した後においても, 検証結果出力における改竄検出について図 4.7 の出力を得た. このことによりデータ書き込み時におけるハッシュ受け渡し手順においても, 改竄検出が意図された通りに動作することが証明された.

データ書き込み手順においては, データを書き込む際には必ずキャッシュに格納した上で書き込むという性質を利用して, 該当メモリエリアを監視しているセキュアプロセッサが自身のキャッシュ内のハッシュツリーへの修正を行うことで, 書き込み側のセキュアプロセッサは自身のキャッシュ内のデータを書き換えて, 他方のセキュアプロセッサの該当データが乗っているキャッシュラインに対しての invalidate 要求を送るだけでよい. その後の invalidate されたデータの完全性検証はデータ読み込み時のハッシュ受け渡し手順に沿って行われる.

Listing 4.7: データ書き込み時のハッシュ受け渡し手順の検証結果出力

```
The event contentTamper is executed.
A trace has been found.
RESULT not event(contentTamper) is false.
```

以上の結果から, 提案手法におけるセキュアプロセッサ間における相互認証手順, セキュアプロセッサ間における機密情報の通信, 各セキュアプロセッサによる完全性の検証手順に関して,

- データへの改ざんを検知することができること
- 誤検知を起こさないこと

- 攻撃者に機密情報を取得されないこと

が記号論理的に証明された.

第5章　考察

5.1　考察

第4章により, 記号論理を用いて提案手法の安全性が示された. 本論文における提案手法においては, アドレス分割を行うことによりキャッシュコヒーレンシによってデータの整合性をとる手法に比べて, アクセスするメモリ領域を管轄するセキュアプロセッサが, キャッシュ内に完全性検証に必要なハッシュ木のノードを保持している場合, メモリアクセス回数の増加はない. また, 本論文においては単純なアドレス分割を提案したが, 分割手法はこれに制限されるものではなく, 改良の余地がある. 例えば, 管轄アドレス領域を動的に変更することで, できるだけ多くのメモリアクセスを自身の管轄アドレス領域内において行うようにすることも考えられる. 今後実機に実装する場合は, キャッシュコヒーレンシプロトコルとの親和性を保つことや, 実行時特有の問題が発生しないかを検証する必要がある.

第6章　関連研究

6.1　IntelSGX

IntelSGX [3] は米 Intel 社が開発しているセキュリティを高めるための拡張命令セット, そしてその命令セットを実行するための実行機構である.IntelSGX は第 2 章で紹介したそれぞれのセキュアプロセッサが備えている機能のうち, タンパ検知と ObliviousRAM を除くすべての機能を備えている. このうちタンパ検知が実装されていないことの理由として, タンパ検知機構を実装する費用が高額であることが考えられる. そして ObliviousRAM に関してはその実行オーバヘッドによる大幅な実行速度低下が SGX への実装を妨げているのではないかと考えられる.

SGX は XOM のようにプロセスごとのコンテナ (Enclave) を持つことができ, メモリ内のあらかじめ用意されたエリア内に秘匿したいアプリケーションのコードやデータを書き込む. また, アクセス制御により Enclave 外のプロセスからのアクセスを防ぐ.

Costan ら [5] よると,IntelSGX は HyperThreading 機構によりプログラムから見えるプロセッサコアを物理プロセッサコア数より多くすることが可能である. ハイパースレッディング機構による論理的なマルチプロセッサ化とはいえ, 他のセキュアプロセッサがマルチプロセッサシステムへの適用に関して言及していないのに対して, IntelSGX はマルチコア化を最も早く推し進めている. しかし IntelSGX もメッセージ認証コードを用いたデータの完全性検証機構を備えているために既存のセキュアプロセッサと同様にマルチプロセッサシステムへの適用を行うには本論文で取り上げた完全性検証機構, 暗号化・復号機構への改変が不可欠である.

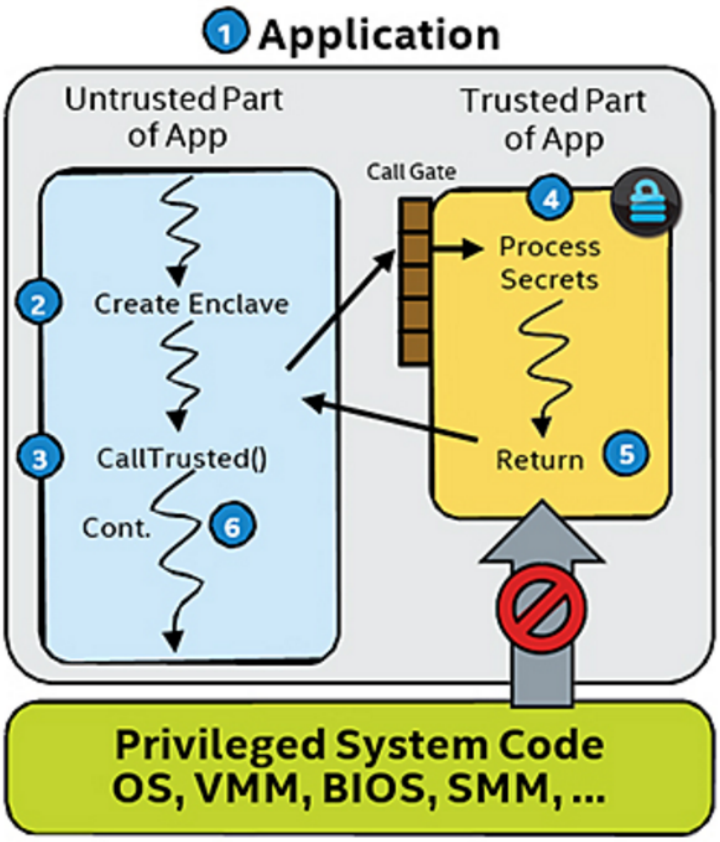


図 6.1: Intel SGX 実行時 (from [4])

第7章　おわりに

7.1　本論文のまとめ

本稿では, セキュアプロセッサが普及していない原因として実行速度の低下を挙げ, シングルチップの性能向上のみに頼らずにこの課題を解決するためにセキュアプロセッサを用いてマルチプロセッサシステムを構成する手法を提案した. セキュアプロセッサをマルチプロセッサシステム上で動作させる上で発生し得る問題を調査し, 「データ鍵を生成したセキュアプロセッサでなければ復号ができない」,「他のセキュアプロセッサによるメモリへの変更を改ざんとして判定してしまう」という課題を発見した. 1 つ目の課題に対しては, セキュアプロセッサ間の相互認証手順と機密情報を暗号化して通信することを提案した. 2 つ目の問題に対しては, 各セキュアプロセッサが完全性検証をするために監視しているメモリ領域を分割し各プロセッサごとに割り当て, 自身の監視しているメモリエリア外に対してアクセスする際は該当メモリエリアを監視しているプロセッサに対してデータの完全性検証を依頼することを提案した. また, セキュアプロセッサ間の相互認証手順とデータ完全性検証手法に関して, プロトコル安全性自動検証ツールである ProVerif を用いてプロトコルの安全性の評価を行った.

7.2　今後の課題

今後の課題の 1 つとして, データの完全性の検証に伴う通信量を削減することが挙げられる. 現段階では, メモリの割り当て方としてアドレスによる単純な 2 分割を考えており, 完全性の検証処理の負担がいずれかのプロセッサに集中してしまう可能性がある. 今後はデータの局所性を意識した監視対象のメモリの割り当てや動的に監視対象のメモリエリアの変更することなどについても考える.

2 つ目の課題として, 実機における実装またはシミュレーションを行いパフォーマンスを計測し, セキュアプロセッサをシングルチップで稼働させたシステムと比較することが挙げられる.

3 つ目の課題としては, 分散システムにおけるセキュアプロセッサの構成方法の考案が挙げられる.AWS のようなクラウド環境のような分散システムでセキュアプロセッサを使うことで, プロセッサの故障など分散システム以外では発生確率が少ないものに関しても考

慮する必要がある. このようにセキュアプロセッサをマルチプロセッサ方式として構成したシステム同士を通信していく上で発生し得る分散システム特有の課題, またその解決法について調査していく必要がある.

参考文献

[1] ARM. *ARM Security Technology Building a Secure System usingTrustZone Technology*, 2009.

[2] Bruno Blanchet, Ben Smyth, and Vincent Cheval. *ProVerif 1.90: Automatic Cryptographic Protocol Verifie , User Manual and Tutorial*, 2015.

[3] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual*, 2016.

[4] Intel Corporation. Intel software guard extensions, https://software.intel.com/en-us/sgx/details, 2016.

[5] Victor Costan and Srinivas Devadas. Intel sgx explained. Technical report.

[6] Advanced Micro Devices. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, 2013.

[7] Joan G Dyer, Mark Lindemann, Ronald Perez, Reiner Sailer, Leendert Van Doorn, and Sean W Smith. Building the ibm 4758 secure coprocessor. Technical Report 10, 2001.

[8] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pp. 3–8. ACM, 2012.

[9] A. Freier, P. Karlton, and P. Kocher. The secure sockets layer (ssl) protocol version 3.0. Technical report, 2011.

[10] Trusted Computing Group. Tpm main specification Technical report, 2013.

[11] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. *ACM SIGPLAN Notices*, Vol. 35, No. 11, pp. 168–177, 2000.

[12] Charles W O'Donnell, G Edward Suh, and Srinivas Devadas. Puf-based random number generation. *In MIT CSAIL CSG Technical Memo*, Vol. 481, , 2004.

[13] Advanced Cryptographic Hardware Development IBM Poughkeepsie, Z¨ urichIBM Research. Ibm 4765 cryptographic coprocessor security module. Technical report, 2012.

[14] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verificatio and encryption for secure processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, p. 339. IEEE Computer Society, 2003.

[15] G Edward Suh, Charles W O'Donnell, and Srinivas Devadas. Aegis: A single-chip secure processor. *IEEE Design & Test of Computers*, Vol. 24, No. 6, 2007.

[16] G Edward Suh, Charles W O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *ACM SIGARCH Computer Architecture News*, Vol. 33, pp. 25–36. IEEE Computer Society, 2005.

[17] Yuval Yarom and Katrina Falkner. Flush+ reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security*, Vol. 2014, pp. 719–732, 2014.

研究業績

ポスター発表（査読なし）

1. 梶原拓也, 宮永瑞紀, 入江英嗣, 坂井修一: セキュアプロセッサを用いたマルチプロセッサシステムの設計, 電子情報通信学会技術研究報告, Vol.116, No. 240, pp. 7-10 (2016)

謝辞

本研究を進めるにあたり, たくさんの方々にお世話になりました. 心より感謝いたします.

指導教官である坂井修一教授には, 相談会でのご指導に加えて, 留学中においても様々な貴重なご意見を頂きました.

入江英嗣准教授には, 原稿の組み立て方から提案手法の詳細な部分に至るまで, 多方面でのご指導をいただきました.

事務補佐員の八木原晴水さん, 赤羽彩子さんには, 研究室設備の整備や各種事務手続きの補助まで幅広いサポートをしていただきました.

また, 坂井・入江研究室のメンバーには研究生活のサポートをしていただきました.

特にセキュリティ班の宮永瑞紀さんには, 坂井研究室に配属されてから現在に至るまで様々な面で親身になって相談に乗っていただき, 大変お世話になりました.

なお, 本論文の研究の一部は, 公益財団法人セコム科学技術振興財団の助成を受けたものです.