Vol.2023-DPS-194 No.60
Vol.2023-CSEC-100 No.60
2023/3/7

安全な機能拡張性を持つTEEシェルの実装

斎藤 文弥[1,a)]　高野 祐輝[3,b)]　宮地 充子[1,2,c)]

概要：Trusted Execution Environment(TEE) はファームウェアや OS といった基盤システム内の機微情報を保護することを目的とした隔離環境技術である．先行研究では，TEE アーキテクチャの一つである Arm TrustZone をベースとしてメモリ安全性と効果系という概念を主軸に設計した Baremetalisp TEE，およびその TEE の API 定義用言語である BLisp を構築した．さらに作成した BLisp のコードを Coq にトランスパイルし形式的検証も可能な手法を提案した．TEE は他の隔離環境技術である Trusted Platform Module(TPM) 等とは異なりユーザーが自由にセキュリティ仕様を構築できることが特徴として挙げられるが，Baremetalisp では独自の言語 BLisp を用いているため拡張できる機能に制限が存在していた．そこで本研究では Baremetalisp を構成している Rust から関数を BLisp へ組み込み可能にすることによって，柔軟な機能の拡張性を実現した．組み込み関数にも効果系が適用することができ，メモリ安全性と形式的な正しさを保証しつつ安全な機能アップデートが可能な TEE Shell を実現した．

キーワード：TEE，Coq，形式検証，型安全性，効果系

Implementation of TEE shell with safe extensibility of functionality

Abstract: Trusted Execution Environment(TEE) is an isolated environment technology to protect sensitive data in infrastructure systems such as firmware and operating systems. In our previous work, with the concepts of memory-safety and effect system, we developed Baremetalisp TEE based on Arm TrustZone and BLisp, an API definition language for TEE. Furthermore, we proposed a method to enable formal verification by transpiling BLisp code to Coq. Unlike other isolation technologies such as Trusted Platform Module (TPM), TEE enables users to freely build security specifications. However, due to usage of its own language BLisp in Baremetalisp, this system has limitations of extending the functionality. In this study, we realized flexible extensibility of the functionality by making it possible to embed functions into BLisp from Rust, which constitutes Baremetalisp. By allowing effect system to be applied to these functions, we created a TEE shell that guarantees memory-safety, formal correctness and secure function updates.

Keywords: TEE, Coq, Formal Verification, Type-safety, Effect System

1. はじめに

1.1 研究背景

Trusted Execution Environment(TEE) は汎用 OS とは独立して暗号化などのクリティカルな処理を実行できる隔離環境技術である．TEE では汎用 OS とその OS 上で動作する一般的なアプリケーションの動作環境とは別に動作環境を提供し，基本的には TEE 上で構築される専用の OS とアプリケーションを実装する．既存の TEE アーキテクチャは Arm 社の TrustZone，Intel 社の SGX，AMD 社の SEV などがあり，またオープンソースソフトウェア (OSS) として RISC-V Keystone が開発されており，それぞれの構造は異なる．

本研究でベースとして使用した Arm TrustZone は主に IoT デバイスなどのモバイル機器に利用されており，スマートフォンにも広く搭載されている．しかし，TEE アーキテクチャにも様々な脆弱性が報告されている [1]．例えばブーメランバグ [2] は TEE 構造の脆弱性を突いたバグであ

[1] 大阪大学
Osaka University
[2] 北陸先端科学技術大学院大学
Japan Advanced Institute of Science and Technology
[3] 株式会社ティアフォー
[a)] fsaito@cy2sec.comm.eng.osaka-u.ac.jp
[b)] ytakano@wide.ad.jp
[c)] miyaji@comm.eng.osaka-u.ac.jp

り，汎用 OS 側のシステム情報などのクリティカルデータが TEE の環境を介して盗み出されるおそれがある．バススヌーピング攻撃 [3] はハードウェアの側面における脆弱性であり，マルチプロセッサのシステムにおけるプロセッサ間の共有キャッシュ上のデータが盗み出される攻撃である．データの保護を目的とした TEE でもこのような脆弱性が存在しているため，構築した TEE ソフトウェアにも実装コードの検証や動作の検証が必要となる．

Arm Trustzone を利用した TEE ソフトウェアも様々に存在し，各種特性が異なる．OP-TEE は Linaro が開発した C 言語で実装されている OSS である．C 言語は昔からシステムプログラミングに用いられており，その動作速度の速さとライブラリの豊富さから様々な基盤システム実装に利用されている．しかし，C 言語はメモリに関する脆弱性を常に考慮する必要がある．ガベージコレクションなどの動的メモリ割り当てを自動的に管理する機能が標準で搭載されていないため，プログラマーが必要なくなったメモリをその都度解放しなければならない．したがって，コードにおけるミスが実行前のコンパイルで発見できずに実行時に露呈してしまうことがある．システムの運用中エラーを回避するためには，C 言語の問題を解決する別の方針を採用しなければならない．以上のように，言語由来の脆弱性や TEE そのものの構造による欠陥によって，情報の保護を目的とした技術にも関わらず情報が流出するおそれがある．

1.2 本研究の目的

先行研究の Baremetalisp [4] は Arm TrustZone をベースに Rust 言語で実装された TEE ソフトウェアである．Rust は独自の概念である所有権とライフタイムという概念を用いて変数の値の処理における競合とプログラムの自動的なメモリ解放を実現している型安全 [5] かつメモリ安全な言語である．またコンパイラ言語であり，かつ実行速度も速いため，高速な処理が必要とされるシステムプログラミングにも適している．しかし Baremetalisp はファームウェアと TEE OS の機能を提供している一方，そのメモリ管理において Rust の提供するメモリ安全性がカバーできない領域が存在する．

先に我々が発表した研究 [6] では BLisp のコードに対して形式的検証を実行可能な TEE シェルを提案した．形式的検証は，証明論理などの形式手法に基づいてプログラムの動作が仕様に沿って正しいかを証明する検証である．この形式的検証を行うことによって Rust では保証しきれない関数の動作を保証することが可能になる．そこでは定理証明支援系言語である Coq を用いて形式証明を実行する．Coq は，プログラミング言語理論と証明論を等価な関係とするカリー＝ハワード同型対応を利用しており，プログラムにおける真偽値や関数適用などの式や演算を証明論理に置き換えることによって，それらの動作の正しさを証明することができる．

一方，BLisp は独自の API 定義用言語であり，C 言語や Rust のような汎用言語レベルでの高度な処理ができないという課題があった．すなわち，Baremetalisp の仕様に関して自由に機能を拡張することができなかった．そこで，Baremetalisp を実装している Rust のマクロを用いて，Rust の関数を BLisp に組み込む機能を追加した．この追加機能によって BLisp 単体では実現しえなかった処理を可能にした．

また，本論文で提案するシステムと既存の TEE ソフトウェアに対する機能比較を行っている．そこでは，前述した OP-TEE に加え，.NET ランタイムモバイルアプリケーションの完全性と機密性を保護する TLR [7], [8]，車載情報システムにおける制御系 OS と情報系 OS を管理するデュアル OS モニタの SafeG [9]，そして先行研究である Baremetalisp を対象に比較考察を記述する．

さらに，様々なソフトウェアに対する検証手法を静的解析と動的解析に大別してその特性比較を行う．静的解析の手法としては本研究の Coq を用いた形式的検証に加え，Network Address Translation(NAT) 機能について仕様を検証する VigNAT [10] や，ソフトウェアデータプレーンを用いた検証ツール [11]，マルウェアコードを検知する SAFE [12] を列挙した．動的解析の手法としては検証済みストレージシステムである Key-Value データベースシステム VeriBetrKV [13] や，NAT やファイアウォールなどのネットワーク機能を実行中に検査する Aragog [14] を対象に考察する。

それぞれの既存研究における検証手法の比較を行うことによって，各種検証がもたらす恩恵と検証プログラムの実装に要するコストを考慮しつつ，TEE ソフトウェアに実装するべき検証手法について考察した．

1.3 本論文の構成

2 章では既存研究について記載する．3 章では TEE が満たすべき要件や脅威の分析と，BLisp の設計について記載する．4 章ではシステムの設計原理および設計内容を記載する．5 章では既存研究との比較評価を記載する．6 章では本研究における長所と短所，および該当分野の展望について記載する．7 章では本研究のまとめを記載する．

2. 関連研究

2.1 既存の TrustZone ソフトウェア

Arm TrustZone は広く利用されている TEE 構築基盤であり，スマートフォンやゲーム機などに搭載されている．しかし，TrustZone を含む TEE ソフトウェアに対して複数の脆弱性が報告されている [1]．その一例としてブーメランバグ [2] が存在し，TEE が提供する隔離環境 Trusted

World と信頼されない Untrusted World 間の構造的な違いに起因する．この脆弱性によって Untrusted なユーザーレベルのアプリケーションが，TEE の特権レベルを利用して，システムの根幹であるカーネルメモリを含んだ Untrusted World 内のあらゆるメモリの読み取りや書き込みが可能になるおそれがある．このようなメモリ関連の脆弱性が TEE には依然として存在し，全ての不正メモリアクセスに対して防御策を講じなければならない．

OP-TEE は OSS として公開されている TrustZone をベースとした TEE であり，C 言語で実装されている．TLR [7], [8] は OS のセキュリティ侵害から.NET モバイルアプリケーションの完全性と機密性を保護し，.NET Micro Framework 実装をベースにしたセキュア構造のランタイムソフトウェアを構築している．SafeG [9] は車載情報システムにおいて，情報を管理する OS と制御を担う OS の二つの OS を単一の CPU で管理するためのソフトウェアモジュールである．情報系 OS より制御系 OS の処理を優先するように設計されており，運転駆動系が安全に管理される車載情報システムの設計が可能になる．

Baremetalisp [4] は型安全性を軸に Rust で実装された TEE 基盤であり，ファームウェアと OS からなる Baremetalisp TEE と API 定義用プログラミング言語の BLisp で構成される．Baremetalisp は汎用 OS に依存せず実装することができ，リソースの小さいデバイスにも搭載することが可能である．

2.2 システムの解析手法

静的解析はソフトウェアを実行する前にプログラムの動作を検証する手法である．一般的にソフトウェアを構成するコードは人が書くものの，それらの全てにミスが無いかを人が網羅的に精査することは困難である．そこで，対象となるプログラムのコードをソフトウェアが機械的に走査する静的コード解析がよく用いられている．

SAFE [12] はウィルスに感染したコードを検知するツールである．SAFE ではマルウェアと検知されないように難読化された悪意のあるコードを，特定のコードパターンを選別するオートマトンと制御フローグラフを構築することによってマルウェアコードの検知を可能にしている．

VigNAT [10] は NAT の仕様に対し，コンパイラやシンボリック実行エンジンなどのツールチェーンと lazy proofs という証明プロセスを適用して動作の事前検証を行った NAT である．このシステムは低レベルなプロパティ (RFC3022 に則って意味的に正しいことと，クラッシュせずメモリーを大量に消費しない) を満たすことを目的として設計されている．

ソフトウェアデータプレーンを用いた検証ツール [11] はルーターやスイッチングハブなどのハードウェアに代替するものとして，ソフトウェアデータプレーンでネットワーク機能を実装し検証する．従来の手法と異なり，検証可能性と性能という一見相反する特性をソフトウェアデータプレーンを用いることによって，検証可能かつ性能を維持することを可能にしている．

動的解析はソフトウェアを実際に実行させ，その挙動を解析することによってプログラムの正しさを検証する手法である．静的解析ではソースコードを対象としてデータの依存関係や制御構文などを検査する一方，動的解析ではコンパイル後の実行ファイルやそのテストケースを対象として，命令の実行時間や実行回数，変数の値などを検査する．

VeriBetrKV [13] は分散システムに対する検証の IronFleet [15] を一般化し，ストレージシステムの検証に適用した検証済み Key-Value データベースである．分散システムの簡単なインスタンスがストレージシステムと捉え，分散システムの検証として洗練されている IronFleet をストレージシステムに応用している．

Aragog [14] は NAT を含めたネットワーク機能を対象とする実行時検証システムである．Aragog は違反を検知するための仕様を記述する専用の言語と，実行時の環境を状態遷移モデルによって管理する状態管理マシンで構成される．これにより，ネットワーク機能に特化した仕様を簡単に定義することができるほか，規模の大きいシステムに対しても実用的に使用することが可能な検証を実装することができる．

3. 設計

3.1 脅威分析

ここでは TEE に対する脅威分析について記載する．2 章で紹介したように，TEE はハードウェアとソフトウェア (ファームウェアや OS) を対象としたセキュリティ保護技術である．そして，一般的に用いられる汎用 OS とアプリケーションの動作環境を Untrusted World，TEE が提供する Trusted OS と Trusted App の動作環境を Secure World としてそれぞれの実行環境を隔離することにより安全性を確保している．

一方で TEE のセキュリティ構造の欠陥を突いたブーメランバグ [2] を利用した攻撃が存在する．TEE では，それぞれ汎用 OS とアプリケーションである Untrusted OS と Untrusted App を Untrusted World に位置づけ，Trusted OS と Trusted App を Secure World に位置づけることによって管理する．しかし，Untrustred OS と Trusted OS をつなぐドライバに関してベンダーが合意した標準化が存在していないため，TEE の安全性が保証しきれていない Untrusted App と Trusted App が直接データ共有を行うチャネルが用意されている．こういった TEE の構造特性を利用してブーメランバグは Untrusted World への不正メモリアクセス可能にしてしまう．

また TEE に対する脅威として考慮しなければならない

```
$T  :=                                  型(Type)
       Int                              整数型
     | Bool                             真偽値型
     | '( $T )                          リスト型
     | [ $T+ ]                          タプル型
     | $TFUN                            関数型
     | $TDT                             代数的データ型
     | $TID                             型変数

$TFUN  :=  ( $EF ( → ( $T* ) $T ) )    関数型
$EF    :=  Pure | IO                    効果系
$TDT   :=                               代数的データ型
       $TID                             型引数なし
     | ( $TID $T+ )                     型引数あり

$ADT   :=  ( data $DDEF $MEM* )         代数的データ型定義
$DDEF  :=  $TID | ( $TID $ID* )         型名
$MEM   :=  $TID | ( $TID $T* )          メンバとなる型(型引数)

$DEFUN :=  ( $FEX $ID ( $ID* ) $TFUN $E )  関数定義
$FEX   :=  export | defun | extern      属性
```

図 1 BLisp の構文

事項に，セキュリティ仕様の設計ミスやプログラマーのコーディングミスといった人為的ミスが存在する．この問題はいかに成熟した開発チームやプログラマーが手掛けても発生する可能性があり，時にはシステムの脆弱性の要因となって重大な損失を生むこともある．これを防ぐためには可能なかぎりシステム側において人為的ミスを防ぐ機構が必要であり，さらにはミスを検知し開発者に報告してくれるような機能を備えていれば理想的である．

3.2 設計

3.2.1 BLisp の効果系

本研究で提案する BLisp に実装される効果系 [16] とは，プログラムにおける計算処理の効果を記述する形式的なシステムである．コンパイル時にこの効果系に沿って書かれたコードは，プログラムの起こり得る効果とその副作用を含めてチェックすることが可能である．副作用とはプログラムや関数の定義されている動作領域から外れて処理が進められた際に発生する影響を示す．

効果系は，純粋に計算を行う論理とそれ以外の副作用を持つ論理を明示的に区別することを目的とする．一例として，ゼロ除算やメモリのオーバーフローなどのエラー処理を除き，既定のメモリ範囲に収まる四則演算は純粋な計算論理として認識し，関数の外部からのデータを入出力するような処理が含まれる場合などは純粋な関数ではないと認識する．すなわち，関数やそのプログラムが行う処理の効果を区分けすることによって，対象となる処理がメモリ割り当てやサイドチャネルなどにおける脆弱性となりうるかどうかを記述することができる．

図 1 では BLisp の型に関する BNF を記載する．型は図の上から順に整数型，真偽値型，リスト型，タプル型，関数型，代数的データ型，型変数で構成される．シンプルな型である整数型や真偽値型はそれぞれ “Int” と “String” として管理され型引数は持たない．一方，リスト型やタプル型では内包する要素の型を示す型引数が必要になる．次に関数型では該当する関数が純粋関数か IO 関数を示す識別子が最初に必要となる．ある関数が純粋関数ならば演算処理を行うだけの関数だと判断することができ，脆弱性等を考慮する必要はない．一方，IO 関数ならば外部メモリへのアクセスなどの対外的な処理を含む関数であり，正しく実装されていなければ脆弱性となりうるリスクを抱えた関数だと捉えることができる．この Pure/IO の明示化は開発者のための概念であり，各関数の持つ役割を曖昧に管理しないように，安全な関数とリスクを持つ関数の両方に属性を与えている．関数型において最も深く括弧で覆われている $T*は関数の入力値として与えられる型を表しており，その次の一つ括弧の外にある$T は返り値の型を表している．

続いて少し図の順番と前後するが，関数定義では異なる三つの属性を用意している．export は記述されているコードとは別の BLisp コードからの利用も可能であることを示しており，一般的なアクセス修飾子の public と同等である．また，defun は反対にコード外部からの利用ができないことを示しており，アクセス修飾子の private と同等である．extern も export と同じく外部からの参照を示すアクセス修飾子であるが，特徴として異なることは他の BLisp コー

ソースコード 1 代数的データ型の表現と型定義

```
# 型引数を持たない場合
(data Color
    Red
    Green
    Blue)

# 型引数を持つ場合
(data Fibon5
    (Fibon5 Int Int Int Int Int))
# インスタンス化
(Fibon5 1 1 2 3 5)
```

ドから関数を参照するのではなく，Baremetalisp TEE を構築している Rust 環境において用意されているライブラリから関数を参照するという点にある．BLisp は効果系やジェネリック関数などの概念を取り込んだ新たな言語である一方，Rust といった汎用言語の持つ様々な演算処理や機能は兼ね備えていない．したがって，本提案システムでは BLisp における機能的ボトルネックという問題に対し，Rust による組み込みを可能にすることによって解決している．

最後に代数的データ型について説明する．代数的データ型は列挙型と同様に扱うことができ，その型表現と型定義をコード 1 で示す．この型では型名とその値の最初の文字は大文字でなければならないという制約があり，型引数を取る場合には実際の値を用いてインスタンス化することも可能である．

3.2.2 Rust に基づく型安全性

前述した BLisp とその基盤システムである Baremetalisp TEE は Rust で実装されたソフトウェアである．Rust は独自の所有権とライフタイムという概念をを用いることによってメモリに関する競合や不正な解放を未然に防ぐことが可能となる．また，本研究では型安全性を一貫して保持することを主軸として設計した．型安全性 [5] とは “正しく型付けされたプログラムは不正な動作をしない” ことを保証する指標である．この “不正な動作をしない” という部分については様々な言語のセマンティクスによって捉え方が変わるものの，根本的には不定義な動作を許さないことが重視される．例として，C 言語において { int buffer[10]; buffer[10] = 100; } のようなコードは，バッファの領域外へのアクセスのため定義外の動作が行われてしまう．また，実行時のエラーも不定義な動作と言える．すなわち，ソースコードがコンパイルを通ったのち実行された際に，例外ハンドラによって処理されるのではなくエラーとして異常終了してしまうような言語は型安全性を満たしているとは言えない．一方，Rust では不定義な動作を起こしうるコードに対してコンパイルを通し，検知することができる．Baremetalisp と BLisp はこの Rust を用いていることによって型安全な TEE を実現している．また，BLisp はモジュール化されているため，他の TEE 基盤を実装する際にも用いることが可能である．以上のように，本研究では型安全性を満たした Rust を用いて BLisp および Baremetalisp のさらなる機能拡充を提案する．

ソースコード 2 マクロを用いた実装例

```
// マクロの呼び出し
    use num_bigint::{BigInt, ToBigInt};

    #[embedded]
    fn test_a(
        id: BigInt,
        id_list: Vec<BigInt>
        ) -> Option<String> {
        if id_list.into_iter()
                .any(|x| x == id) {
            let new_pw = get_pw();
            Some(new_pw)
        } else {
            None
        }
    }

// マクロの展開結果
    fn test_a(...) {...} // 省略
    const TEST_A: &str = "(extern test_a (->
          (Int \'(Int)) (Option String)))";
```

4. 実装

ここでは，Rust で実装した BLisp への組み込みマクロ機能を解説する．

今回，用いたマクロは属性マクロと呼ばれる手続き的マクロである．手続き的マクロは，特徴としてコードを入力として受け取り，そのコードに処理を施して，出力としてコードを出力する．属性マクロは宣言的マクロや他の手続き的マクロである derive マクロ，関数マクロよりも柔軟に属性 (機能) を付加することができ，構造体や関数など様々な式に適用することができる．本提案システムでは，入力コードを抽象構文木に落としこんで型や演算などを抽出して BLisp で用いることができるようなコードを生成する．その構文解析では，マクロを使用する際に便利なクレートである syn クレートと quote クレートを利用する．各クレートの主な役割として，syn クレートは入力コードの解析，quote クレートは出力コードの生成を担っている．

ソースコード 2 にマクロの入力および展開結果を記載する．ここでは，ID と ID リストを入力として，もしリストに該当 ID がある場合には新たなパスワードを設定し Some でカプセル化した文字列を返し，なければ Option 型

	OP-TEE	TLR [7], [8]	SafeG [9]	先行研究 (Baremetalisp) [4]	本研究
型安全性	×	△	×	○	○
柔軟な機能拡張性	○	○	△	×	○
機能拡張の安全性	×	△	×	○	○
Normal OS との互換性	△	○	○	△	△
形式的検証の適用	×	×	×	×	○

表 1 TEE ソフトウェアとしての機能比較

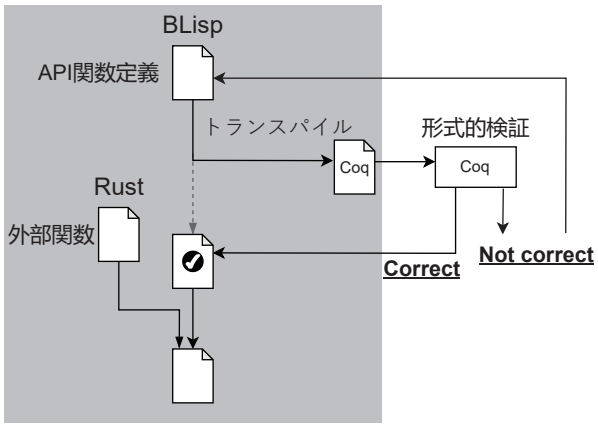

図 2 本研究での検証フロー

の None を返す関数を定義している．先行研究の組み込み関数である call-rust では入力値の型としてリスト型 (Rust ではベクタ型) を用いることや，出力値において Option 型でカプセル化された String 型を利用することはできない．また，マクロの展開結果について 19 行目は 5-16 行目と同じコードが書かれているため省略する．今回，マクロが提供する機能は 16 行目であり，この定数の文字列が Rust と BLisp の組み込みを繋ぐ架け橋となる．マクロで生成された文字列は Rust ではなく BLisp の構文であるため，この文字列を BLisp のコードに追加するように Rust のライブラリを変更して 4 行目のように組み込みたい関数に #[embedded] の属性を付け加えるだけで簡単に Rust 関数の組み込みが可能になる．

次に先行研究 [6] で実装した BLisp から Coq へのトランスパイラと Rust マクロを利用した外部関数の組み込み機能に関するフローを説明する．

図 2 に示すように，Baremetalisp を用いた TEE の設計においてセキュリティ仕様や API の処理に携わる関数を BLisp で作成する．このコードに対して BLisp から Coq へのトランスパイラを適用し，コードの意味論を変えずに Coq のコードに変換する．この Coq コードを，CoqIDE や Proof General を通して検証し，関数の分岐処理や入出力値の型などに曖昧な記述や不正が含まれていないかをチェックする．Coq でエラーが発生した場合はもう一度 BLisp のコードでエラーの原因となった部分を再構成する．検証によって正しいと判断された場合は Baremetalisp 上で実行フェーズに移行することが可能となる．また，実行中に API などの処理について機能の追加を行いたい場合には，Rust のライブラリで関数を作成して Rust-BLisp 間のインターフェースを用意することにより，機能拡張を実現する．

5. 評価

ここでは既存のソフトウェアや先行研究との機能比較と，静的解析や動的解析などの様々な手法を適用したソフトウェアとの検証特性の比較を行う．

最初に既存研究との機能比較について表 1 に記載する．まず OSS(オープンソースソフトウェア) である OP-TEE について，C 言語で実装されているため型安全性は保証されない．また，自由に TEE 向けアプリケーションを開発し実装することができるため，機能の拡張性はある一方，その拡張に関する安全な設計もされていない．また，Trusted OS と対をなす Normal OS との互換性について，各 OS 間のドライバに関する規格の標準化が定まっていないことから，安全なプロトコルが整えられていない．

TLR は.NET ランタイムにより自由に Trusted App の開発および実装が可能で，それに伴い機能の拡張性は高い．しかし，型安全性や機能拡張の安全性は実装に用いた言語に依存する．例えば，C#などの言語では型安全性を保証できないが，OCaml をベースとして型安全であり型推論の機能を持つ F#などを利用すると安全性を保証できる．TLR は隔離環境で保護された Trusted App の開発を目的としており，Normal OS との安全なインタフェースが設計されている．

SafeG について車載情報システムを管理する高信頼デュアル OS モニタであり，運転駆動系の制御を行う制御系 OS とセンサーなどから得られる情報を処理する情報系 OS からなる．設計のコンセプトとして制御系の処理が優先して実行される権限構造や，情報系の環境を常に監視しいつでも停止させられるように構成されていることから各環境間における処理の優先度が厳密に決められている．しかし，SafeG は C 言語で実装されており型安全性は保証されないほか，柔軟にアプリケーションを実装できるようには設計されていない．

先行研究 [4] はスタンドアロン環境で動作する Baremetalisp TEE とその API 定義言語として BLisp を提案して

	動的解析		静的解析			
	VeriBetrKV [13]	Aragog [14]	VigNAT [10]	[11]	SAFE [12]	本研究
汎用性	×	△	×	△	○	△
コンパイル時検証	×	×	○	○	○	○
エラー発生時の修正	○	○	△	△	-	○
スケーラビリティ	○	○	○	○	○	○

表 2 検証手法に関する比較

いる．Rust による型安全性と BLisp の効果系によって機能拡張に関する安全性も兼ね備えている．しかし先行研究における BLisp では，Rust や C 言語のように高度な演算処理を実装することができないため，機能拡張性について難点がある．

本研究では特に BLisp の機能向上に焦点を当て，柔軟な機能拡張と形式的検証の適用を組み込んだ．Rust の手続き的マクロを利用して，BLisp では実装できない高度な演算を Rust で実装しその関数を BLisp で読み込めるようにすることにより，より自由に処理を書き加えることが可能になった．また，BLisp のコードに対して Coq を適用することによって，API 定義に対する形式的な検証も可能にしている．

続いて，検証手法に着目して 3 章で紹介した既存研究との比較を表 2 に記載する．

まず動的解析について VeriBetrKV はキーバリューデータベースシステムの検証に絞っており，またデータ挿入のイベント毎に処理の正しさを検証するため，汎用性は低くコンパイル時検証ではないため，実行時にエラーやバグが露呈するリスクがある．一方，イベント毎に検証する点やデータベースというシステムの性質から，エラー発生時の修正がしやすいことやスケーラビリティに優れている．

Aragog はネットワークを中継する機能 (NAT:Network Address Translation) に対する検証手法を提案しており，実行時において仕様を満たさないイベントを検知する．独自の仕様記述言語と状態遷移を用いたモデル検査によって，仕様と各イベントの整合性を確認するプロセスであるため，他のソフトウェアに容易に適用することはできない．内部的に検証項目をローカルとグローバルに分けることによって，スケーラビリティに優れている．

同様に，VigNAT は NAT に対する手法を提案しており，NAT の仕様に関する正しさの証明を部分化し連鎖的に検証する．そのため実行前に検証するが，証明の内容としては NAT の動作に偏っていることから，他のソフトウェアへの適用は難しい．証明においてはデータ構造やモデルを対象とし，エラーの特定は全体的なシステムの概要に関する知識が必要となる．

Software Dataplane を用いた検証 [11] も NAT を対象としており，プログラムのソースコードもしくはバイナリコードからその動作を検証する．パケット処理において分岐を効率的に構成し，分岐先の異常な出力を検出するというプロセスのため，エラーの原因となるコードを特定し出力することは難しい．

SAFE は悪意のあるコードを検出するためのツールであり，様々なソフトウェアに適用可能である．また，外部からコードが入ってきた際にその動作を検証するため，プログラムが動く前にその安全性を確かめられる．エラーの修正については，悪質なコードの検知を行うツールであり，基本的にコードの修正を目的としたツールではない．さらに採用するシステムの規模に依存していないため，様々なソフトウェアに搭載することが可能である．

本研究が提案する BLisp と Coq を用いた検証では，TEE としてユーザーが希望する仕様を Coq で検証した後に隔離環境を提供する構造であり，Coq による検証を外部ですることによりリソースの小さい端末から大きいマシンまで様々なハードウェアに搭載することが可能である．また Coq では対話型コンパイルを実行するため，正しくないコードを逐一修正することができる．しかし，TEE における Trusted OS と Trusted App に対する実装であるから，ソフトウェア的な観点から汎用性が高いとは言えない．

6. 議論

ここでは評価に対する考察と今後の展望について議論する．

TEE はラップトップ PC やサーバーマシンからスマートフォンに至るまで，様々な電子機器に利用されている技術である．その技術の主たる目的は，機微情報の保護と暗号化の処理の秘匿化であり，C 言語の脆弱性に挙げられるようなメモリ管理のリスクは最も忌避すべき問題である．型安全性は，ソースコードに対してコンパイルがエラー無く通り正しく型付けされていることが確認されたならば，決して不正な動作をしないことを保証する．すなわち，定義領域外へのアクセスや解放済みメモリへのアクセスなどを未然に防ぐことが可能となる．このような安全性は TEE のみならず汎用 OS でも満たされるべきであり，C 言語系統に存在するメモリの脆弱性から脱却する新たな言語やシステム構成が必要とされる．

次に検証手法について静的解析および動的解析はその性

質が大きく異なる．静的解析はソフトウェアプログラムのソースコードおよびバイナリコードを対象にその起こしうる動作を検証する．一方，動的解析はテスト環境などで実際に実行した際の動作を対象に検証を行う．さらに，動的解析は静的解析と両立することができるため，最低限満たすべき仕様を静的解析によって検証し，実際の運用環境に近い実験環境で動的解析を行うことでそのソフトウェアがどういった動作を起こしうるのかをテストすることができる．

以上のように規模の大小や機能性の幅にかかわらず，機微情報を扱ったり暗号化を伴う処理を実行するソフトウェアを実装するような全ての電子機器に対して，TEE といったデータおよび動作環境を保護する機構や各種検証手法を適用することによって，ハードウェア側面でもソフトウェア側面でも安全であることを保証しなければならない．

7. おわりに

Baremetalisp [4] は Arm TrustZone をベースに設計された TEE システムであり，型安全性を中核において Rust 言語により実装されている．Baremetalisp システムの API 定義用プログラミング言語 BLisp は Lisp ライクな高階言語であり，効果系の概念をサポートしている．効果系を採用することによって意図しない IO を防ぐことを実現している．これにより従来難しかった，TEE ソフトウェアへの任意のコード注入を安全に実行することが可能となる．しかし，BLisp はその構成の性質上，C 言語や Rust と同じレベルの演算や型が用意されていなかったため，柔軟に API の機能を拡張することができなかった．

本研究では，Baremetalisp を実装している Rust のマクロを用いることで Rust による高度な演算処理も兼ね備えた BLisp を実現した．また，BLisp はモジュール化されているため Baremetalisp のみならず他の TEE ソフトウェアを構築する際にも用いることができる．そして BLisp を適用することによって効果系による I/O の安全な管理と Rust による型安全性を，実装するソフトウェアにも提供することが可能になる．

謝辞 本研究の一部は文部科学省の平成 30 年度「Society 5.0 実現化研究拠点支援事業」，JSPS 科研費 JP21H034438，さらにセコム科学技術振興財団 特定領域研究助成の助成を受けています．

参考文献

[1] Cerdeira, D., Santos, N., Fonseca, P. and Pinto, S.: SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems, *2020 IEEE Symposium on Security and Privacy*, pp. 1416–1432 (2020).

[2] Machiry, A., Gustafson, E., Spensky, C., Salls, C., Stephens, N., Wang, R., Bianchi, A., Choe, Y. R., Kruegel, C. and Vigna, G.: BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments, *24th Annual Network and Distributed System Security Symposium* (2017).

[3] Lee, D., Jung, D., Fang, I. T., Tsai, C. and Popa, R. A.: An Off-Chip Attack on Hardware Enclaves via the Memory Bus, *29th USENIX Security Symposium*, pp. 487–504 (2020).

[4] 高野祐輝，金谷延幸， 津田侑：Make TrustZone Great Again, コンピュータセキュリティシンポジウム 2020 論文集，pp. 422–429 (2020).

[5] Milner, R.: A theory of type polymorphism in programming, *Journal of Computer and System Sciences*, Vol. 17, No. 3, pp. 348–375 (1978).

[6] 斎藤文弥，高野祐輝，宮地充子：Coq で検証可能な TEE シェル基盤の実装，技術報告 8，コンピュータセキュリティ研究会（IPSJ-CSEC） (2022).

[7] Santos, N., Raj, H., Saroiu, S. and Wolman, A.: Using ARM trustzone to build a trusted language runtime for mobile applications, *Architectural Support for Programming Languages and Operating Systems*, ACM, pp. 67–80 (2014).

[8] Santos, N., Raj, H., Saroiu, S. and Wolman, A.: Trusted language runtime (TLR): enabling trusted applications on smartphones, *12th Workshop on Mobile Computing Systems and Applications,*, ACM, pp. 21–26 (2011).

[9] Sangorrin, D., Honda, S. and Takada, H.: Dual operating system architecture for real-time embedded systems, *Proceedings of the 6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT)*, pp. 6–15 (2010).

[10] Zaostrovnykh, A., Pirelli, S., Pedrosa, L., Argyraki, K. J. and Candea, G.: A Formally Verified NAT, *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ACM, pp. 141–154 (online), DOI: 10.1145/3098822.3098833 (2017).

[11] Dobrescu, M. and Argyraki, K.: Software Dataplane Verification, *11th USENIX Symposium on Networked Systems Design and Implementation*, pp. 101–114 (2014).

[12] Christodorescu, M. and Jha, S.: Static Analysis of Executables to Detect Malicious Patterns, *Proceedings of the 12th USENIX Security Symposium* (2003).

[13] Hance, T., Lattuada, A., Hawblitzel, C., Howell, J., Johnson, R. and Parno, B.: Storage Systems are Distributed Systems (So Verify Them That Way!), *14th USENIX Symposium on Operating Systems Design and Implementation*, pp. 99–115 (2020).

[14] Yaseen, N., Arzani, B., Beckett, R., Ciraci, S. and Liu, V.: Aragog: Scalable Runtime Verification of Shardable Networked Systems, *14th USENIX Symposium on Operating Systems Design and Implementation*, pp. 701–718 (2020).

[15] Hawblitzel, C., Howell, J., Kapritsos, M., Lorch, J. R., Parno, B., Roberts, M. L., Setty, S. T. V. and Zill, B.: IronFleet: proving safety and liveness of practical distributed systems, *Commun. ACM*, Vol. 60, No. 7, pp. 83–92 (2017).

[16] Pierce, B. C.: *Advanced Topics in Types and Programming Languages*, The MIT Press (2004).