Vol.2022-CSEC-97 No.8
Vol.2022-IOT-57 No.8
2022/5/19

Coqで検証可能なTEEシェル基盤の実装

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）は電子計算機のセキュアデータを保護する技術の一つであり、広く利用されているシステムである。一方で、C 言語で実装されているものが多いため、メモリ管理に対する脆弱性が依然として存在する。そこで我々は、Barematalisp TEE の API 定義用言語 Blisp から Coq へトランスパイルするプログラムを Rust 言語を用いて構築した。Coq は定理証明支援系システムであり、関数や式を形式的に証明する。このプログラムはメモリ安全性と意味的正しさを兼ね備えた Trusted Shell 実装の一助となり、一般的なシェルと同様の動的なプログラムの注入と実行を可能にする。

キーワード：TEE，型安全性，Coq，プログラム検証

Implementation of TEE shell infrastructure verifiable by Coq

Abstract: Trusted Execution Environment (TEE) is a widely used software system that is one of the technologies to protect secure data in computers. However, since many TEEs are implemented in C, vulnerabilities to memory management still exist. Therefore, we constructed a program to transpile Blisp, the API definition language of Baremetalisp TEE, to Coq using Rust. Coq is a theorem proving support system that formally proves functions and expressions. This program helps to implement a Trusted Shell implementation that is both memory-safe and semantically correct, and allows dynamic program injection and execution similar to a general shell.

Keywords: TEE, Type-safety, Coq, Program Verification

1. はじめに

1.1 研究背景

IoT 化が加速している社会において電子機器のセキュリティの重要性は日々高まっている。特に、人々が持ち歩くモバイル機器は紛失してしまう可能性が高く、悪意のある攻撃者の手元に渡ってしまうことも少なくない。その一方で、スマートフォンを筆頭にウェアラブル端末などが扱うデータはますます多くの機微情報を扱うようになっている。これらのデータは例えば、秘匿通信に用いられる暗号鍵や指紋認証に利用される生体情報に該当する。そこで、重要なデータをハードウェアとソフトウェア両方の側面から保護するシステム構築が課題となっている。

Trusted Execution Environment（TEE）は隔離環境技術の一種であり、一般的な OS やアプリケーション等が動作する環境と、暗号鍵や生体情報などのクリティカルデータを扱うための環境をハードウェア上で分けることができる。なお、本論文では、一般的なソフトウェアの動作環境をノーマルワールド、TEE により隔離された環境をセキュアワールドと呼ぶ。TEE は隔離環境技術の一つであるが、普及している電子端末には既にいくつかの隔離実行機能が備わっている。それらの技術が Hardware Isolated Execution Environment（HIEE）としてまとめられており、TEE はそれらの一つという位置づけである [1]。TEE 以外の HIEE においてシステムのセキュリティ仕様は、ハードコーディングされており内容を更新することができないことや、CPU や端末のプロバイダーは更新できても一般ユーザーが更新することはできないという特徴がある。しかし、TEE は一般ユーザーがプログラム可能であり各セキュリティ仕様を自由に設計することができるという利点

[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

がある。

一方で、ノーマルワールドにおける侵害を防ぐどころか、セキュアワールド上で様々な脆弱性が報告されている [2]。例としてブーメランバグ [3] は、全てのソフトウェアに存在する脆弱性であり、セキュアワールドを介してノーマルワールドのメモリを不正に利用することが可能なバグである。また、選択的シンボリック実行を利用した Trusted Application（TA）におけるヒープオーバーフローの脆弱性も報告されている [4]。

1.2 本研究の目的

1.1 節で述べたように多機能なモバイル機器の需要が高まっている。ビッグデータを利用した AI のエッジコンピューティング等では、センシングなどで得られたデータがセキュアワールド上に格納され、何らかの演算を施された後に暗号化してクラウドに送信すると想定される。エッジコンピューティングを行う環境では、データに対する演算をエッジ側で実行すると思われるが、そのためには TEE のセキュアワールド上に必要となる計算コードをその都度アップロードしなければならない。既存の TEE システムではこの動的なコードの注入が難しいとされてきた。原因として、任意コードの注入そのものが脆弱性に繋がるという点がある。したがって、この拡張性を実現するためには注入されるコードの動作を検証しなければならない。

そこで我々は、TEE 上で動作する Trusted Shell を提案する。図 1 に簡単にその特性を示す。TEE はシステムソフトウェアであることから動作速度が重要視されるため C 言語をベースとして開発している既存研究が多い [5] [6]。C 言語はぶらさがりポインタなどのメモリに関する脆弱性があり、プログラマーが注意してコーディングをしないと簡単に意図しない動作を起こしてしまう。Rust はメモリ安全性を実現しつつ、動作速度も高速なため C 言語に代替するシステムソフトウェア言語として注目されている。この Rust を利用することによってメモリ安全かつ高速に動作する Shell を実現している。Coq は定理証明支援システムの一つであり、高階型システムに基づいている。検証したいプログラムを Coq に書き下し、逐次実行することによって形式的な証明を行うことができ、各々の関数の動作検証が可能となる。この Shell はコードの注入および TA のインストールやアップデートを可能にし、TEE ソフトウェアの拡張性に貢献する。

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 Trusted Shell の設計原理およびそれぞれの使用言語における設計について記載する。4 章では実装および実行結果を記載する。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。

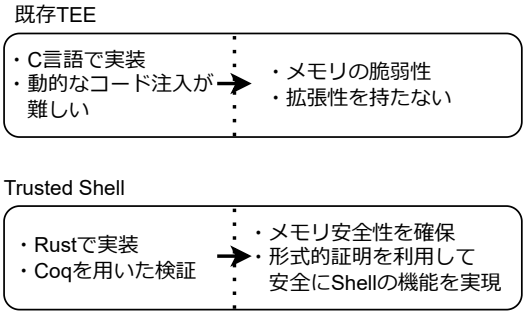

図 1 既存研究との比較

2. 関連研究

2.1 Baremetalisp

Baremetalisp [7] 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。この型安全性は、正しく型付けされた式はその後の演算も正しく進められることと、正しく型付けされた式が評価可能であるならば、評価後の式も正しく型付けされるという性質である [8]。すなわち、“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。ここで不正な動作とは、プログラマーが想定していない処理を含むコードがコンパイルをすり抜けて、実行時に未定義な動作を起こしてしまうことを指す。当論文の著者は TEE OS にこそ型安全性が必要であると主張し、設計・実装している。

図 2 に Baremetalisp の簡単な概要を記載する。Baremetalisp TEE のファームウェアは最も特権レベルが高い層で動作してメモリ保護領域の管理、セキュアワールドとノーマルワールド間の切り替えを管理するセキュアモニタコール（SMC）、コンテキストスイッチ、OS の起動プロセスを担う。Baremetalisp OS はノーマル OS と同じ特権レベルの層で動作して、後述する BLisp ランタイムの起動プロセス、動的メモリ確保、両ワールド間の共有メモリの読み書き、BLisp の評価関数呼び出しを担う。BLisp は本研究でも扱うため 3.2.1 で詳しく書くが、共有メモリに格納されたコードを BLisp ランタイムが評価し型付けチェックを行う。

しかし、Baremetalisp には検証すべき部分がある。Rust で実装することによってメモリ安全性を保証しているものの、直接メモリ操作を行う MMU などではメモリ安全性を担保することができない。これを補うための検証が必要になる。

2.2 Arm TrustZone

Arm TrustZone [2], [9] は最も広く利用されている TEE

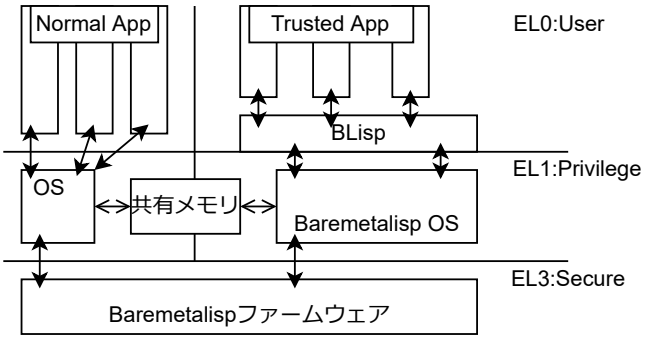

図 2 Baremetalisp の概要

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。TrustZone ではノーマルワールドとセキュアワールドの二つの領域が存在し、ノーマルワールドでは一般的な OS やハイパーバイザ、アプリケーション等が動作する。一方、セキュアワールドにはファームウェアと必要最小限の OS とアプリケーションが置かれており、セキュアワールド上で動作する OS を Trusted OS、アプリケーションを Trusted Application（Trusted App）と呼ぶことが多い。TrustZone のメモリ保護機能により、ノーマルワールドに不正に漏洩することのないメモリ領域をセキュアワールド上に構築することができる。

Arm アーキテクチャの一つである Cortex-A では特権レベル（Exception Level）が 4 つ用意されている。ノーマルワールドとセキュアワールドでは同じように全ての特権レベルがあり、OS を動かしている。また、ノーマルワールドとセキュアワールドを行き来するためには Secure Monitor Call（SMC）命令を用い、セキュアモニタを介することによって切り替えを行う。また、セキュアワールドに向けて確保するメモリ領域やペリフェラルのリソースは起動時に予め設定される。各ワールドのペリフェラルの割り込みは別々の割り込みリクエストが割り当てられている。

TrustZone 向けの TEE ソフトウェアも存在しており、オープンソースとしては Aalto University の Open-TEE、Linaro の OP-TEE、Google の Trusty などがある。商用としては Qualcomm の QTEE、Samsung の Knox、Huawei の TrustedCore などがある。しかし、これらの TEE のいずれにも脆弱性が指摘されている [2]。脆弱性としては、Trusted OS 上で任意コードの実行、セキュアワールドを介したノーマルワールドメモリへの不正アクセスなど、メモリやコードへのインジェクションに対する脆弱性が数多く報告されている。

3. 設計

3.1 設計原理

本研究では、TEE 上で動作する安全なシェル基盤を提案する。そこで我々は、本設計の中核に型安全性を据えて設計および実装した。型安全性を備えた Trusted Shell は未定義の動作や意図していない挙動をすることがないため、スタックオーバーフローやぶら下がりポインタなどのメモリに関する脆弱性を防ぐことができる。さらに、Trusted Shell は動的なコードの注入についての検証を正しく実行できることから、低いリソースで実現されるエッジコンピューティングにおいても有用性を十分に発揮すると推測する。また、BLisp では効果系 [10] という概念を利用している。効果系によりプログラムが起こしうる効果や影響をコンパイル時にチェックすることが可能となる。したがって、BLisp における意図していない IO を防ぐことができる。本プログラムでは BLisp で動作仕様を記述されたコードを、Rust を利用して Coq のコードにトランスパイルする。Coq を用いて形式的に動作を証明することにより、安全な TEE Shell 基盤を実現している。

3.2 設計

3.2.1 BLisp

BLisp は高階 API を実現する関数型プログラミング言語である。BLisp の型と式の種類について図 3 に記載する。型は新しく定義するための型定義 TVar と既存の型を表す TCon の列挙型で表す。TVar は新しく定義した型ごとに u64 の値が割り当てられる。TCon は型の名前を保持する String 型の変数と型引数を保持する二つのフィールドを持つ。例えば真偽値の場合には型名 “Bool”、型引数は None となる。Int 型や String 型、Char 型も同様に扱われる。タプル型やリスト型は同様に型名を付けた後、型引数をとる。タプルは各要素ごとに異なる型を持つことができるため、それぞれの要素の型を格納した可変長リストを Option 型である Some でカプセル化したものを型引数として持つ。一方リスト型は、同じ型の値を複数格納するため、単一の型を保持する可変長リストを Some でカプセル化したものを型引数として持つ。最後に関数型は、型から型への変換を表す演算子として “−＞” を型名とする。型引数としてはそれぞれ効果系である Pure か IO を保持する変数と、関数の引数の型を保持する変数、そして返り値の型を保持する変数がある。

次に式について、真偽値や整数値のリテラルは実際に持つ値を保持するフィールド、コード上にある位置を表すフィールド、その型を Some でカプセル化して保持するフィールドを持つ。また、以下すべての式は位置フィールドと型フィールドを共通して持ち、これらの記載を割愛する。変数では、その自身の変数を表す文字列を格納する。代数的データ型では新しく定義した型を用いる場合に利用する。新しく定義された型を表すフィールドとその中身を格納している可変長リストのフィールドを持つ。let 文では宣言する変数のフィールドと格納される値のフィールドを持つ。if 文では、条件ブロックの式を表すフィールド、then ブロックの式を表すフィールド、else ブロックの式を

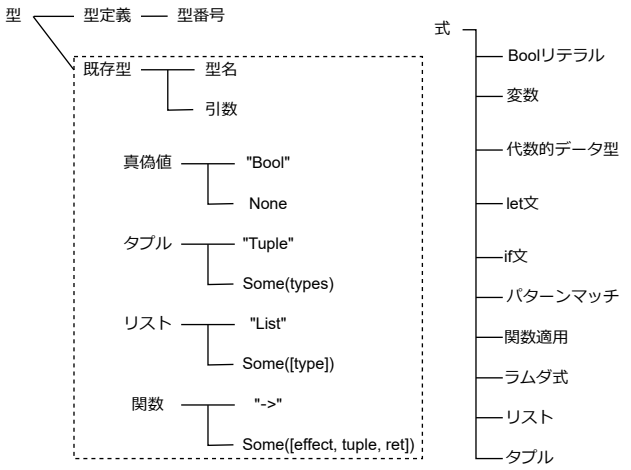

図 3 BLisp の構文

ソースコード 1 BLisp における式表現の具体例

```
変数
:
:"xを文字列として格納する"
int x;代数的データ型

:
:pair (a, b)という型を定義した場合
"pairという型を持つフィールドと"
"aと""bのそれぞれの型や値を持つフィールド"
```

表すフィールドを持つ。パターンマッチは BLisp、Rust、Coq に共通してある記法であり、ある値に対して該当する分岐だけを通り処理を行う制御フローの一つである。パターンマッチでは前述したその値を格納するフィールド、分岐時に判断するための値とその後処理される式を格納するフィールドを持つ。関数適用では複数の式を格納した可変長リストのフィールドを持つ。可変長リストに格納されている型は順に関数、引数の型である。ラムダ式では引数となる変数の文字列とその型を可変長リストで格納するフィールド、演算を決定するフィールド、演算で使用する値を格納するフィールド、ラムダ式を識別するための整数値を格納するフィールドを持つ。最後にリストとタプルのリテラルについて、実際にもつ値を可変長リストとして保持するフィールドを持つ。

3.2.2 Rust

Baremetalisp は Rust により実装されている。Rust は借用権と参照という概念をもとにした記法によりメモリ安全性を実現している。本研究では BLisp のコードを Rust で構文解析し、Coq のコードへとトランスパイルしているが、Rust にはパターンマッチという記法が標準で備わっている。従来、パターンマッチは Haskell といった関数型言語でよく見られた機能だったが、関数型言語以外の言語にも組み込まれ始めている機能である。構文解析プログラムを作成する際に、このパターンマッチは最適だと考え、Rust による実装の一因である。

ソースコード 2 BLisp のコード 1

```
(data (Option t)
    (Some t)
    None)

(data (Result t e)
    (Ok t)
    (Err e))

(export car (x) (Pure (-> ('(t)) (Option t)
    ))
    (match x
        ((Cons n _) (Some n))
        (_ None)))

(export cdr (x) (Pure (-> ('(t)) '(t)))
    (match x
        ((Cons _ l) l)
        (_ '())))

(export map (f x) (Pure (-> ((Pure (-> (a)
    b)) '(a)) '(b)))
    (match x
        ((Cons h l) (Cons (f h) (map f l)))
        (_ '())))

(export fold (f init x) (Pure (-> ((Pure
    (-> (a b) b)) b '(a)) b))
    (match x
        ((Cons h l) (fold f (f h init) l))
        (_ init)))
```

3.2.3 Coq

Coq は形式的な証明を可能とする定理証明支援系言語の一つである。Coq は数学的な分野における証明だけではなく、プログラム等で定義する関数の動作を Coq が持つ強力な型付けの機能を利用して検証することができる。Coq による形式的な検証は様々な分野で用いることができ、今もなお研究が進められている [11], [12], [13]。本研究では API 定義を行う関数に対し、Coq による形式的証明を利用して意味論的に検証を行う。

4. 実装

本章では、入力として与える BLisp のコードとその出力結果となる Coq のコードを解説する。

4.1 BLisp の検証コード

まずコード 2 について、最初に Option 型と Result 型を定義している。Option 型は型理論に基づいた値のカプセ

ソースコード 3 各関数の実行例

```
1 ; リストの合計
2 (fold + 0 '(20 50 60))
3
4 ; フィルタリング
5 (filter (lambda (x) (= (% x 2) 0)) '(1 2 3
      4 5 6 7 8 9))
6 ; 実行結果'(2 4 6 8)
```

ル化を表す多相性の型である。この型は、関数適用した際の戻り値として意味のある値を返すかどうか分からない場合に有用な型である。もし意味のない値の場合は None を保持し、そうでない場合は元のデータ型 A をカプセル化し Some A という値を保持する。この Option 型により、正当でない値だった場合の例外処理を容易に書くことが可能となる。Result 型は正しい戻り値かエラーが起きた場合のエラーコードを保持するモナド型である。この Result 型を使うことによってプログラマーは成功した場合と失敗した場合の両方の処理を記述することを確約することができる。car と cdr は外部に公開される関数である。car 関数は配列を引数として受け取り、その配列が空でなければ Option 型である Some n として先頭の値を返し、空の場合は None を返す。cdr 関数は同じく配列を引数として受け取り、その配列が空でなければ先頭の値のみを除外した配列を返し、空の場合は空の配列を返す。map と fold も同じく外部に公開される関数である。map 関数は関数と配列を引数として受け取り、その配列が空であれば空の配列を返す。入力した配列が空でない場合は、全ての配列の要素に対して入力値である関数 f を適用した値の配列を返す。fold 関数は入力された配列を展開して関数適用することができる。例として、コード 3 に示す。

次にコード 4 について、外部に公開する純粋関数 filter を定義している。この関数はリストの要素をフィルタリングする。filter 関数では引数として、任意の型 T を受け取り Bool 値を返す純粋関数と型 T のリストを受け取る。そして、外部に公開されない filter' 関数を呼び出す。filter' 関数の内部では関数の内部を再帰的にまわして全ての要素を網羅的にチェックする。filter 関数の実行例をコード 3 に記載する。reverse 関数は配列を引数として受け取り順番を入れ替えた配列を返す、外部に公開されない純粋関数である。その操作の中身は外部に公開されない reverse' 関数で行う。reverse' 関数においても再帰的に処理される。これらの関数は一部であるが、Shell として利用するための各種操作を BLisp に書くことを想定している。

4.2 トランスパイルされた Coq コード

トランスパイル結果の Coq をコード 5 に記載する。

まず、今回のコードではリスト型を利用するため Coq

ソースコード 4 BLisp のコード 2

```
1  (export filter (f x)
2      (Pure (->
3          ((Pure (-> (t) Bool)) '(t))
4          '(t)))
5      (reverse (filter' f x '())))
6
7  (defun filter' (f x l)
8      (Pure (-> (
9          (Pure (-> (t) Bool)) '(t) '(t))
10         '(t)))
11     (match x
12         ((Cons h a) (if (f h) (filter' f a (
               Cons h l)) (filter' f a l) ))
13         (_ l)))
14
15 (export reverse (x) (Pure (-> ('(t)) '(t)))
16     (reverse' x '()))
17
18 (defun reverse' (x l) (Pure (-> ('(t) '(t))
       '(t)))
19     (match x
20         ((Cons h a) (reverse' a (Cons h l)))
21         (_ l)))
```

のリストに関するライブラリをインポートしている。次に Option を定義しており、型を引数として取り型を返す。いずれの値もなかった場合には None を、正当な値がある場合には Some にカプセル化して値を保持する型となる。なお、Arguments の部分は Coq 用に追加している文言であり、Some や None の Option 型がコード内で使用される場合には暗黙的に、いずれにも型引数 t が与えられるようになる。これにより本来ならば、Some{t} n と記述しなければ正しく認識されないが、Some n と書いても正しく解釈されるようになる。続いて、Result を定義しており正当な動作を行った場合にその値をカプセル化して保持する Ok とエラーが発生した場合にそのエラーコードをカプセル化して保持する Err の二つの要素を持つ。Result も同じく Arguments を設定しており、Ok と Err に暗黙的に二つの型引数を与えることを事前に明示している。17 行目では car 関数を定義している。Coq では関数の定義として Definition と Fixpoint が用意されている。Fixpoint は再帰させることが可能であるため、再帰が必要な関数では Fixpoint を使って定義する。car 関数は型引数 t とその型 t の値を持つリストを引数として型 t の値をカプセル化した Option を返す。Coq においてもパターンマッチの記法があり、BLisp における書き方と少し異なるが動作は同じである。また BLisp にも Coq にもある cons という文字は Lisp ライクな言語で使用される基本的な関数に由来する。cons はしばしばリストや二分木といった複雑なデータ構造

ソースコード 5 Coq のコード 1

```
Require Import Coq.Lists.List.

Inductive Option (t: Type): Type :=
| Some (x0: t)
| None.

Arguments Some{t}.
Arguments None{t}.

Inductive Result (t e: Type): Type :=
| Ok (x0: t)
| Err (x0: e).

Arguments Ok{t e}.
Arguments Err{t e}.

Definition car {t: Type} (x: list t): Option t :=
match x with
  | (cons n _) => (Some n)
  | _ => (None)
  end.

Definition cdr {t: Type} (x: list t): list t :=
match x with
  | (cons _ l) => l
  | _ => nil
  end.
```

ソースコード 6 データ構造の具体例

```
リスト表現
;
[1, 2, 3] = (cons 1 (cons 2 (cons 3 nil)))

;と carcdrリスト
;の番目の要素を取り出す場合 L3
(car (cdr (cdr L)))
```

ソースコード 7 Coq のコード 2

```
Definition filter {t: Type} (f: t -> bool) (x: list t): list t :=
(reverse (filter' f x nil ) ).

Fixpoint filter' {t: Type} (f: t -> bool) (x l: list t): list t :=
match x with
  | (cons h a) => match (f h ) with
    | true => (filter' f a (cons h l) )
    | false => (filter' f a l )
    end
  | _ => l
  end.

Fixpoint fold {a b: Type} (f: a -> b -> b) (init: b) (x: list a): b :=
match x with
  | (cons h l) => (fold f (f h init ) l )
  | _ => init
  end.

Fixpoint map {a b: Type} (f: a -> b) (x: list a): list b :=
match x with
  | (cons h l) => (cons (f h ) (map f l ))
  | _ => nil
  end.

Definition reverse {t: Type} (x: list t): list t :=
(reverse' x nil ).

Fixpoint reverse' {t: Type} (x l: list t): list t :=
match x with
  | (cons h a) => (reverse' a (cons h l) )
  | _ => l
  end.
```

を表現するために使われ、本研究では用途をリストに絞って使用している。car 関数と cdr 関数は Lisp 言語におけるリストを操作するための最も基本的な関数である。実用例はコード 6 に記載する。

同様にコード 7 について、filter は再帰せず、型引数 t、型 t を受け取り bool 値を返す関数 f、型 t の値を格納するリスト x を引数として、型 t の値のリストを返す関数として定義されている。関数の内部で reverse と filter' を呼び出しているが、Coq では呼び出される関数は事前に定義されている必要があるため、エラーが発生する。今回の実装では、関数や型の定義の順序性を考慮してトランスパイル後のコードに反映させる機能はまだ付けられていない。そのため、手動で関数の順番を入れ替えたのちに Coq で検証していかなければならない。

他の関数についても同じように、再帰の有無、型引数の有無、引数とその型、返り値の型をもとに定義しており各関数の動作については 4.1 節で解説したとおりである。

5. 評価

本節では、提案した Trusted Shell と既存 TEE ソフトウェアの定性的な評価を比較する。比較対象としては、OP-TEE と Trusted Language Runtime（TLR）[14] とする。TEE に関するソフトウェアは他にも数多くあるが、

表 1　既存研究と本研究の比較

	本研究	OP-TEE [5]	TLR [14]
型安全性	✓		△
ソフトウェア検証	✓		
動的なコード注入の安全性	✓		

Arm TrustZone をベースに設計されておりオープンソースである OP-TEE と、同じく Arm TrustZone をベースにランタイム検証を行っている TLR を対象とした。

比較項目は、TEE システムにおける型安全性、ソフトウェア検証、および動的なコードの注入の安全性である。型安全性はメモリに関するエラーを検知し重大な実行時エラーを防ぐことができるため、システムソフトウェアの安全性としては非常に重要である。ソフトウェア検証は、対象とするシステムに期待される要求と実際の実装とのミスマッチによる実装バグの発生を防ぐことを目的とする。検証の手法としてシンボリック実行、モデル検査、形式手法があり、想定される入力パターンを網羅的に検証しその動作を保証する。本研究では実装に用いるコードを Coq に過不足なく書き換えることで、強力な型推論による形式的な検証をサポートする。これにより、型安全性に関して Rust ではカバーできなかった部分の安全性も保証する。最後に動的なコード注入の安全性では実行中に計算コードを注入されても、TEE におけるノーマルワールドとセキュアワールドの両方でシステム要件に沿った機密性と完全性を保証する。

本研究の提案方式では、型安全性と形式的検証、動的なコード注入の安全性を全て保証することが出来る。これらは、Rust 言語による実装と効果系を適用した BLisp による設計と実装および Coq を用いた検証を行っているためである。

OP-TEE は C 言語で実装されている TEE であり、型安全性は保証されていない。そのため、メモリの脆弱性や例外処理の不備などを検知することができず、実行時にエラーが起こりうる。TLR は.NET ランタイムにより重大なエラーの発生を防ぐことができる。しかし、.NET で利用される C#言語は型安全性を保証できない。また、動的なコードの注入に関してもサポートしていないため安全性は保証されていない。

6. 議論

本章では提案したソフトウェアの課題点と今後の展望について考察する。

6.1 課題点

本研究で提案したトランスパイルを行うプログラムでは、関数の順序性は考慮されない。一方で、Coq では関数定義が適切な順に行われていなければ、コンパイルが通らないため手作業で関数の順番を入れ替えなければならない。すると、コードの行数が増えるにつれユーザーへの負担が爆発的に増大してしまう。ソフトウェアの利便性や使いやすさを考慮すると改善しなければならない。また、Coq は数学における定理をコンピュータで扱い証明を進められるソフトウェアとして知られているが、システムソフトウェアの検証ツールとしては研究が進められていない。したがって、Coq の特性および記法を理解して用いるには少し時間がかかる。

6.2 今後の展望

従来、TEE が保護していた領域は暗号処理であり秘匿保護していたものは暗号鍵などであった。しかしハードウェア性能の向上により TEE システムで利用できるリソースが増えると、暗号鍵などのデータだけでなくプログラムのコードそのものも保護しようとその領域が広がっている。また、機械学習や分散データ管理においては実行されるコードに加え、その重み付けデータやセキュリティパラメータが重要になるため、それらを秘匿しなければならない。さらに、暗号プロトコルを安全に実行するために TEE 上で実装する提案も存在する [15]。機密性と完全性が必要とされる暗号処理と、メモリ保護およびデータバスインジェクションの防止をサポートする TEE 技術を組み合わせて、それぞれの弱点を補うことでより強固なプロトコルの実現が期待できる。

7. おわりに

本研究では、TEE 環境で動作する Trusted Shell 基盤の設計および実装を行った。TEE は他の隔離環境技術と異なり一般ユーザーが仕様を決めることができ、セキュリティ仕様を自由に設計することが出来る。Baremetalisp 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。この技術は、機械学習を用いたクラウドシステムで用いることが可能にし、特に低いリソースでクリティカルデータを扱うモバイル機器などのエッジコンピューティング環境で、演算処理のプロトコルをアップデートすることができる一助になると推測している。

一方で、実装に用いた Rust は型安全かつメモリ安全ではあるものの、ファームウェアや OS における直接的なメモリ操作に関してはそれらの安全性を保障することができ

ない。そこで我々は、Coq を用いた形式的検証を組み込む手法を提案した。Coq における証明は型付きラムダ計算の一種であり、関数内部の演算における型の受け渡しを強力に検証することができる。Coq での検証は、アプリケーションレベルやサーバー等の状態を検証するのみでなく、動作する関数の正当性が非常に重要視されるシステムソフトウェアにも用いられるべきである。

本研究は Trusted Shell の基盤を実装しているのみであり、ノーマルワールドにおけるアプリケーションの相互性などはまだ実装できていない。今後は実用的に利用できる仕様にまでソフトウェアを拡張し、Raspberry Pi 等に実装するなどして動作性およびその仕様をより向上させていきたい。

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」、さらに JSPS 科研費 JP21H034438 の助成を受けています。

参考文献

[1] Zhang, F. and Zhang, H.: SoK: A Study of Using Hardware-assisted Isolated Execution Environments for Security, *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016, HASP@ICSA 2016, Seoul, Republic of Korea, June 18, 2016*, ACM, pp. 3:1–3:8 (online), DOI: 10.1145/2948618.2948621 (2016).

[2] 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, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, IEEE, pp. 1416–1432 (online), DOI: 10.1109/SP40000.2020.00061 (2020).

[3] 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, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*, The Internet Society, (online), available from ⟨https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/boomerang-exploiting-semantic-gap-trusted-execution-environments/⟩ (2017).

[4] Busch, M. and Dirsch, K.: Finding 1-Day Vulnerabilities in Trusted Applications using Selective Symbolic Execution (2020).

[5] : Open Portable Trusted Execution Environment, `https://www.op-tee.org/`.

[6] : Open-TEE, `https://github.com/Open-TEE/Open-Tee.github.io`.

[7] 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).

[8] Pierce, B. C.: *Types and Programming Languages*, MIT Press (2002).

[9] Pinto, S. and Santos, N.: Demystifying Arm TrustZone: A Comprehensive Survey, *ACM Comput. Surv.*, Vol. 51, No. 6, pp. 130:1–130:36 (online), DOI: 10.1145/3291047 (2019).

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

[11] Affeldt, R., Kobayashi, N. and Yonezawa, A.: Verification of Concurrent Programs Using the Coq Proof Assistant: A Case Study, *IPSJ Digital Courier*, Vol. 1, pp. 117–127 (online), DOI: 10.2197/ipsjdc.1.117 (2005).

[12] Dumas, J., Duval, D., Ekici, B. and Pous, D.: Formal verification in Coq of program properties involving the global state effect, *CoRR*, Vol. abs/1310.0794 (online), available from ⟨http://arxiv.org/abs/1310.0794⟩ (2013).

[13] Jain, K., Palmskog, K., Çelik, A., Arias, E. J. G. and Gligoric, M.: mCoq: mutation analysis for Coq verification projects, *ICSE '20: 42nd International Conference on Software Engineering, Companion Volume, Seoul, South Korea, 27 June - 19 July, 2020* (Rothermel, G. and Bae, D., eds.), ACM, pp. 89–92 (online), DOI: 10.1145/3377812.3382156 (2020).

[14] 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, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014* (Balasubramonian, R., Davis, A. and Adve, S. V., eds.), ACM, pp. 67–80 (online), DOI: 10.1145/2541940.2541949 (2014).

[15] 鈴木達也，江村恵太， 面和成，大東俊博：Intel SGX を用いた公開検証可能な関数型暗号の構成と実装評価，暗号と情報セキュリティシンポジウム (SCIS) (2020).