Computer Security Symposium 2020
26 - 29 October 2020

Make TrustZone Great Again

高野 祐輝[1,a)]　金谷 延幸[2]　津田 侑[2]

概要：Trusted Execution Environment（TEE）の一つである Arm TrustZone は、OS やハイパーバイザが実行されるノーマルワールドから TEE で実現されるセキュアワールドを分離する技術であり、たとえノーマルワールドが侵害されたとしても TEE の健全性が確保されなければならない。そこで我々は、型安全性の考えを中核において TrustZone 向けの新たな TEE 実現基盤である Baremetalisp を設計・実装したため本論文で報告する。Baremetalisp は Rust 言語で実装され、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 から構成される。Baremetalisp TEE はメモリ安全性を最重要視して設計・実装した。また、BLisp は効果系を適用しており、副作用のある関数と純粋な関数を完全に分離して扱う。これにより、従来難しかった、セキュアワールドへの動的なプログラムの注入と実行、すなわち Edge Computing 的な任意の計算を TEE 内で安全に行うことが出来るようになる。

キーワード：オペレーティングシステム、Trusted Execution Environment、プログラミング言語、型システム、効果系

Make TrustZone Great Again

Yuuki Takano[1,a)]　Nobuyuki Kanaya[2]　Yu Tsuda[2]

Abstract: Arm TrustZone is one of the technologies of Trusted Execution Environment (TEE) and it separates a secure world from a normal world, in which OS and hypervisor live. Even if the normal world is compromised, the secure world must be clean, but many TEE's bugs are reported. In this paper, we present Baremetalisp, which is a new TEE for TrustZone, and its design and implementation adopt the type safety as a core design principle. Baremetalisp is implemented in Rust language, and it consists of a firmware and OS, called Baremetalisp TEE, and a domain specific language, called BLisp, for defining API. Baremetalisp TEE is carefully designed and implemented from a memory safety perspective. BLisp adopts the effect system and distinguishes functions having side effect and others. This makes it possible to inject program into the secure world. In other words, edge computing like computations can be performed in TEE securely.

Keywords: operating system, trusted execution environment, programming language, type system, effect system

1. はじめに

Trusted Execution Environment (TEE) は隔離技術の一種であり、一般的な OS、ハイパーバイザ、アプリケーションなどが動作する環境と、機微情報等を扱うための環境をハードウェアレベルで分離することができる。TEE は様々な CPU に実装されており、Arm には TrustZone、Intel には Secure Guard Extensions (SGX)、RISC-V には Keystone がある。なお、本論文では、一般的なソフトウェアが動作する環境をノーマルワールド、TEE 技術により隔離された環境のことをセキュアワールドと呼称する。

セキュアワールド内では機微情報を扱うため、セキュアワールドで実行されるソフトウェアには高い安全性が要求される。すなわち、たとえノーマルワールド内のソフト

[1] 大阪大学
Osaka University
[2] 情報通信研究機構
National Institute of Information and Communications Technology
[a)] ytakano@wide.ad.jp

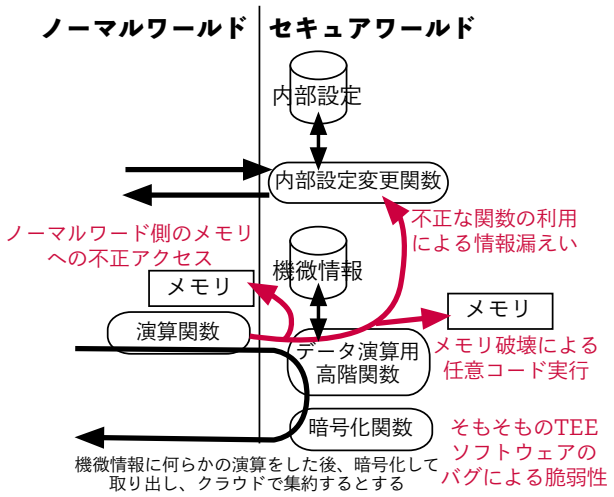

図 1 TEE 内データに対する演算

ウェアに脆弱性が発見されたとしても、セキュアワールド内のソフトウェアは侵害されてはならない。しかし、実際にはノーマルワールドが侵害されるどころか、セキュアワールド内のソフトウェアに様々な脆弱性が発見されている [1]。例えば、ブーメランバグ [2] は、全ての TEE 実装に共通するバグであり、このバグを利用するとノーマルワールド内のメモリを不正に利用することができる。その他には、セキュアワールド上のアプリケーションをのっとり、DRM 情報にアクセスするような脆弱性も報告されている [3]。

このように、セキュアワールド内の脆弱性は大きな問題となる。ここで、これら TEE の利用想定についてとその脆弱性について例を用いて説明する。図 1 は、Edge Computing 環境において、エッジ側の計算ノードで TEE が利用されていると想定する。Edge Computing の場合、何らかの機微情報がセキュアワールド内に格納され、そのデータに演算が施された後に暗号化されてクラウド等に集約されると想定される。また、セキュアワールドに格納するデータ数などを設定するための API も用意されることが想定される。

この図のように、セキュアワールド内のソフトウェアに脆弱性があった場合、機微情報が盗まれる、セキュアワールド内で任意コードが実行されてしまう、ノーマルワールドの不正なメモリの利用されるといった問題が起きる。

さらに、TEE 用ソフトウェア自体の柔軟性の問題もある。Edge Computing 環境ではデータに対する演算を Edge 側で行うことが想定されるが、状況に応じた演算を行うためには、計算コードをセキュアワールドに注入する必要があるが、これは一般的には難しい。なぜなら、通常、セキュアワールドで動作するソフトウェアは頻繁または動的な変更等は想定されていないのと、そもそも任意コードの注入自体がセキュリティホールとなりうるために制限されるべきだからである。注入するコードの検証が正しく行われない場合、不正な関数利用による情報漏洩などが発生する可能性がある。

そこで、本研究では型安全性の考えを中核において設計・実装された、新たな TEE 実現基盤である Baremetalisp の提案を行う。Baremetalisp は、TEE 実装の一つである Armv8 を対象としており、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 から構成される。

型安全性とは、型に関して進行と保存が成り立つことを示す性質であり [4]、すなわち、null 安全、メモリ安全等が型システムによって保証されている事を示す。Baremetalisp TEE は型安全なプログラミング言語である Rust 言語を用いて設計・実装しており、従来のセキュアワールド用ソフトウェアで問題となっていた、バッファオーバランなどのメモリに関する致命的なバグを軽減させることができる。

また、BLisp も同様に型安全な言語であるが、さらに効果系 [5] と呼ばれる概念を取り入れ、副作用のある関数と純粋な関数を完全に分離して扱う。これにより、従来は難しかった、ノーマルワールドからセキュアワールドへの IO の無い任意コードの注入を安全に行う事を可能とする。これにより、セキュアワールド内での Edge Computing 的な計算やソフトウェアアップデートが安全かつ容易に行えるようになる。

2. 関連研究

TrustZone 以外の TEE 実装としては、Intel SGX や、RISC-V の Keystone などもあるが、本節では Baremetalisp の対象とする TrustZone 関連の研究を主に紹介する。

2.1 TrustZone

Arm TrustZone のアーキテクチャは CPU バージョン等によって異なるが、本節では Baremataliasp で対応している Armv8-A [6] の TrustZone について簡単にのみ説明する。また、これ以降、Armv8-A を Armv8 と略記する。Armv8 ではノーマルワールドとセキュアワールドと呼ばれる 2 つの領域が存在し、ノーマルワールドにはハイパーバイザや、Linux などの一般的な OS、Linux アプリケーションなどがおかれる。一方、セキュアワールドにはファームウェアと最小限の OS とアプリケーションのみが配置され、セキュアワールドで動作する OS は Trusted OS、アプリケーションは Trusted Application (Trusted App) と呼ばれることが多い。TrustZone のメモリ保護機能により、セキュアワールド側の特定メモリを、ノーマルワールドから隔離することができる。

Armv8 では例外レベルと呼ばれる概念で OS やハイパーバイザを階層化しており、Exception Level (EL) 3 にはファームウェア、EL2 にはハイパーバイザ、EL1 に OS、EL0 にユーザランドアプリケーションが配置される。EL1

と EL0 はノーマルワールドとセキュアワールドの両方に存在するが、EL3 はセキュアワールドのみ、EL2 はノーマルワールドのみに存在する。

2.2 既存 TEE 用ソフトウェア

TrustZone 用の TEE 用ソフトウェアはいくつか存在し、Qualcomm の QSEE、Samsung の Trustonic TEE、Huawei の TEE、NVIDIA の TEE、Linaro の OP-TEE などがある。D. Cerdeira らの報告によると、これら TEE のいずれにも多くの脆弱性が報告されており、2013 年から 2018 年までの間に、脅威度が中程度より高い脆弱性は 80 もあった [1]。脆弱性の種類は様々であり、ノーマルワールドの任意メモリが読まれるケース、TEE 内のデータが漏洩するケース、TEE 内で任意コードが実行されるケースなどがある。また、TEE ソフトウェアのみではなく、Trusted App に脆弱性があった場合にも、これらの問題が起きると報告している。

TEE ソフトウェアの脆弱性対策は様々な提案がされているが、Rust 言語で Trusted App を実装する RustZone [7] が本研究と近い。しかし、RustZone は OP-TEE 上でのアプリケーションを作成するのみであり、ファームウェアと OS 及び、API 用のプログラミング言語実装までは対象としていない。N. Santos らは TrustZone 内で、.NET アプリケーションを実行させるランタイムを提案している [8] が、これもやはりファームウェアと OS は対象としていない。一方、.NET ランタイムをベースとしているため、実行時エラーが発生したとしてもそのエラーの影響が OS やファームウェアまでに及ぶことはない。

3. 設計原理

3.1 OS 設計・実装の型安全性

OS の設計・実装で特に問題となることの 1 つにはメモリの扱いがある。OS 実装に、ぶらさがりポインタへのアクセスやバッファオーバランなどの脆弱性が含まれると、意図しないメモリアクセスが発生し、サービス停止や機密情報の情報漏えいなどといった重大な問題につながる。しかし、これらがおきる主要因は、C 言語といった型安全でないプログラミング言語を用いているからである。

型安全性とは、進行と保存が保証されている性質のことである [4]。進行とは、正しく型付けされた式は、値になるか計算ステップが規則通り進むという性質のことであり、保存とは、正しく型付けられた式が評価可能なとき、評価後の式もまた正しく型付けされるという性質のことである。これら性質を有する OCaml、Haskell、Rust といった型安全なプログラミング言語では、C 言語などでおきる、ぶらさがりポインタやバッファオーバランといった問題はコンパイル時に発見、除去することが可能となる。

型安全なプログラミング言語で実装された OS としては、OCaml 言語で書かれた Mirage OS [9] や、Haskell 言語で書かれた House [10]、Rust 言語で書かれた Redox [11] などがある。これら作者も型安全性はセキュリティ面で重要な役割を担っていると主張している。そこで、本研究では、Trusted OS にこそ型安全性が必要であるとの思想のもと、設計・実装を行う。

3.2 高階 API の型安全性

API はセキュアワールドとノーマルワールドとを結びつける重要な役割を担っており、その設計にも高い安全性が求められる。理想的には、ノーマルワールドからの API 呼び出し時に渡される引数等のチェックは厳格に行われ、実行時に想定外の挙動をしないことを保証しなければならない。しかし、実際には必ずしもそのような設計となっていない場合がある。

例えば、ブーメランバグ [2] は API に渡される引数の使われ方についても検証しないと防ぐことの出来ないバグである。また、TrustZone では Trusted App と呼ばれるアプリケーションが Trusted OS 内で動作するが、QSEE はノーマルワールドから Trusted App を Trusted OS 内に読み込むことができ、これに Trusted App にバグがあった場合に複数の問題が起きることが報告されている [1]。特に、後者のような実行コードを引数に取ることのできる API を、本論文では高階 API と総称する。

そこで我々は、OS の設計・実装のみではなく、Trusted OS の提供する API についても型安全性の考えを取り入れなければならないという哲学のもと、型安全な高階 API を実現する方法を提案する。型安全な高階 API を用いることで、ブーメランバグや Trusted App 中のバグのような問題の多くが排除できるようになる。さらに、eBPF verifier で行われるようなコード検査も型システムの理論に基づいて行うことが出来るようになり、不正なコードを排除できるのに加えて、ある種のアクセスコントロールも可能とする。

4. Baremetalisp の設計

本節では、我々の提案する Baremetalisp の設計を示す。Baremetalisp は高階 API を実現するプログラミング言語の BLisp 言語と、Armv8 のセキュアワールドで動作する Baremetalisp TEE から構成される。そこで、まずはじめに BLisp 言語の設計を示し、その後に Baremetalisp TEE の設計を示す。

4.1 BLisp: 高階 API を実現する型安全な Lisp ライク言語

BLisp の構文は Lisp に近いものとなっているが、静的型付け言語であるため、その意味論は ML や Haskell に近い。Lisp 風の構文を採用した理由は、パーサを簡易に実装できるためである。また、BLisp では効果系 [5] と呼ばれる概

念を取り入れ IO の分離を行っているのが大きな特徴となる。Baremetalisp では、この効果系に基づいて API を定義するため、意図しない IO を防ぐことができる。式の評価は正格に行い、関数適用する際は左の引数から評価する。正格評価を採用した理由は、遅延評価は評価のタイミングや順番を予測するのが難しい場合があり、システムプログラミングには向かないと考えたからである。

4.1.1 構文と意味

図 2 は BLisp の構文と意味を表したものとなる [*1]。このように、構文は基本的に S 式であるが、型に関する意味論は ML や Haskell に近い。ただし、if 式や let 式は一般的な Lisp 言語とほぼ同じように記述できる。

まずはじめに基本的な型の記述方法について説明する。次のソースコード 1 は型指定の例である。BLisp の組み込み型には、64 ビット整数値の Int 型と、真偽値型、リスト型があり、このうちリスト型は高階型となっている。2 行目では Int のリスト型を示している。すなわち、'(10 20 30) というリストの型は 2 行目で表される型となる。5 行目は関数型の例である。これは、Int 型の値を 1 つ受け取り、Bool 型の値を返す関数型となる。また、Pure は効果であり、この関数は純粋、すなわち IO がないと言うことを示している。8 行目も関数型の例であるが、こちらは IO のある関数であることを示す効果が指定されている。

次に、代数的データ型の定義方法について説明する。BLisp では、Haskell と同様な方法で代数的データ型を定義可能である。ソースコード 2 は代数的データ型の定義例であり、2 行目はリスト型、7 行目は Maybe 型の例を示している。それぞれの型の t は型変数であり、パラメタライズド型を実現するのに用いられる。2 行目のリスト型は BLisp の組み込み型として定義されており、'(Int) 型は (List Int) 型の糖衣構文である。

最後に、BLisp の完全なコードについて解説する。ソースコード 3 は、BLisp の簡単な例を示している。2 行目と 5 行目で代数的データ型を定義しており、9 行目から add 関数を定義している。関数定義は defun か export で行い、export で定義した関数はノーマルワールドから呼び出し可能となる。10 行目は関数の型を指定しており、BLisp では関数の型を必ず指定しなければならない。技術的には、関数の型を指定しなくても型推論は可能ではあるが、API 定義を行うという特性上、型指定は必須とした。ただし、defun では型変数を利用可能であるため、多相関数も定義可能である。11 行目は match 式の例であり、一般的なプログラミング言語と同様にパターンマッチも利用できる。また、13 行目の let 式では分配束縛を行っている。

4.1.2 型システム

本節では、BLisp の型システムを示す。まずはじめに、

[*1] 実際の BLisp はこれより複雑な構文だが、本論文では基礎的な部分に絞って説明する。

表 1 表記

$A \Rightarrow B$	含意 (A ならば B)
e	式
z	整数リテラル（例：10, -34, 112）
x	変数
t	型変数
E	効果
$E_{\mathcal{T}} : \mathcal{T} \to E$	型から効果への関数
$\mathcal{T}$	型
C	型制約
$io : C \to \texttt{Bool}$	C は `IO` 関数を含むか?
Γ	型環境
$\mathcal{X}$	t の集合
$FV_\Gamma : \Gamma \to \mathcal{X}$	Γ 中の自由変数の集合を得る関数
$\Gamma \vdash e : \mathcal{T} \mid_{\mathcal{X}} C$	e の型は Γ より、C の制約のもと $\mathcal{X}$ という変数を利用して $\mathcal{T}$ と推論される

表記について説明する。表 1 は本節で用いる表記となる。$e, E, \mathcal{T}$ は図 2 の$E, $EF, $T に相当する。$\mathcal{C}, \Gamma$ は型制約と型環境であり、図 3 で表される集合となる。型制約とは、右辺と左辺の型が等しいという情報を持つ集合であり、型環境は変数の型を保持した集合である。$E_{\mathcal{T}}, io, FV_\Gamma$ は関数である。例えば、$E_{\mathcal{T}} : \mathcal{T} \to E$ と書かれた場合は、$\mathcal{T}$ を引数にとり E をかえす関数となり、$E_{\mathcal{T}}((\texttt{IO}\ (\to\ (\texttt{Int})\ \texttt{Int})))$ とするとこの値は IO となる。

図 4 は BLisp の制約型付け規則となる。ここでは、一部のみ掲載しているが、if や match 等の式についても形式的に定義している。T-True, T-False, T-Num はリテラルのための型付け規則であり、T-Var は変数の型付け規則である。T-App は関数適用の型付け規則で、$e_1, \cdots, e_n$ の型を得た後、制約規則と型集合を求め、関数適用後の型 t を推論する。ここで、効果を含む関数の型も制約集合 C へと追加していることに注意されたい。

T-Defun は関数定義の型付け規則であり、引数 $x_1, \cdots, x_n$ と式 e の型を推論して、関数の型 $\mathcal{T}$ を導く。ここでは、関数の効果が Pure の場合は、型制約に IO 関数が含まれていないことをチェックしており、含まれている場合は型再構築に失敗する。このようにすることで、Pure 関数の中で IO 関数を呼ぶようなコードを型検査で発見することができる。

T-Lambda は λ 式の型付け規則である。基本的には T-Defun と同じであるが、λ 式は必ず Pure 関数となる。これは、ノーマルワールドで行える関数定義は Pure 関数のみに制限し、IO を含む高階関数呼び出しに制限を設けるためである。

4.2 Barmetalisp TEE: Well-typed TEE

本節では Baremetalisp TEE の設計を示す。Baremetalisp TEE とは、セキュアワールドの EL3 と EL1 で動作するファームウェアと OS の総称である。ファームウェアが主

$T	:=		型
		Int	整数型
	|	Bool	真偽値型
	|	'($T)	リスト型
	|	$TFUN	関数型
	|	$TDT	代数的データ型
	|	$ID	型変数
$TFUN	:=	($EF (→ ($T*) $T))	関数型
$EF	:=	Pure | IO	効果
$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*)	
$MEM	:=	$TID | ($TID $T*)	
$DEFUN	:=	($FEX $ID ($ID*) $TFUN $E)	関数定義
$FEX	:=	export | defun	

$E	:=		式
		$LITERAL	リテラル
	|	$ID	変数
	|	$TID	代数的データ型
	|	($TID $E*)	代数的データ型
	|	(if $E $E $E)	if 式
	|	(lambda ($ID*) $E)	λ 式
	|	($E+)	関数適用
	|	'($E*)	リスト
	|	$L	let 式
	|	$M	パターンマッチ
$L	:=	(let ($V+) $E)	**let 式**
$V	:=	($LP $E)	
$LP	:=	$ID | ($TID $LP+)	
$M	:=	(match $E $C+)	パターンマッチ
$C	:=	($MP $E)	
$MP	:=	$LITERAL | $ID	
	|	$TID | ($TID $MP*)	

図 2　BLips の構文と意味（基礎部分のみ）

ソースコード **1**　型の例

```
; Int のリスト型
'(Int)

; Int を 2 つ受け取り、Bool をかえす純粋関数型
(Pure (-> (Int Int) Bool))

; Int のリストを受け取り、Bool をかえす関数型
(IO (-> ('(Int)) Bool))
```

ソースコード **2**　代数的データ型の定義例

```
; リスト型
(data (List t)
    (Cons t (List t))
    Nil)

; Maybe 型
(data (Maybe t)
    (Just t)
    Nothing)
```

ソースコード **3**　BLisp の例

```
; 2次元のデータ型
(data Dim2 (Dim2 Int Int))

; Maybe 型
(data (Maybe t)
    (Just t)
    Nothing)

(defun add (a)
    (Pure (-> ((Maybe Dim2)) Int))
    (match a
        ((Just val)
            (let (((Dim2 x y) val))
                (+ x y)))
        (Nothing
            0)))
```

$\mathcal{C}$	:=	$\mathcal{T} = \mathcal{T}, \mathcal{C}$	型制約	Γ	:=	$x : \mathcal{T}, \Gamma$	型環境
	|	$\varnothing$			|	$\varnothing$	

図 3　型制約と型環境

に担当するのは、メモリマネジメントユニット（MMU）によるメモリ領域保護設定、セキュアモニタコール（SMC）、コンテキストスイッチ、OS の初期化と起動である。OS が担当するのは BLisp ランタイムの初期化と起動、動的メモリ確保、ノーマルワールドとの共有メモリへの読み書き、BLisp の eval 関数呼び出しである。なお BLisp ランタイムの起動時に、built-in された BLisp の API 定義ファイルも読み込む。eval 関数からは、この定義ファイル中に export された関数を呼び出すことが可能となる。

図 5 は、ノーマルワールドから BLisp の式を評価するまでの流れを示している。1. まずはじめにノーマルワールドの OS は共有メモリに BLisp コードを書き込む。ただし、この共有メモリは、事前に MMU にてセキュアワールドとノーマルワールドからアクセス可能に設定してあるものとする。2. その後 SMC でファームウェアを呼び出し、3. ファームウェアはセキュアワールドの OS に割り込みを発行する。4. 次にセキュアワールドの OS は共有メモリから BLisp コードを読み込み、BLisp ランタイムの eval 関数

$$\Gamma \vdash \texttt{true} : \texttt{Bool} \mid_{\varnothing} \varnothing \quad \text{(T-True)} \qquad \Gamma \vdash \texttt{false} : \texttt{Bool} \mid_{\varnothing} \varnothing \quad \text{(T-False)}$$

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T \mid_{\varnothing} \varnothing} \quad \text{(T-Var)} \qquad \Gamma \vdash z : \texttt{Int} \mid_{\varnothing} \varnothing \quad \text{(T-Num)}$$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : \mathcal{T}_1 \mid_{\mathcal{X}_1} C_1 \quad \Gamma \vdash e_2 : \mathcal{T}_2 \mid_{\mathcal{X}_2} C_2 \wedge \cdots \wedge \Gamma \vdash e_n : \mathcal{T}_n \mid_{\mathcal{X}_n} C_n \{t\} \cap FV_\Gamma(\Gamma) = \varnothing \\ \{t\} \cap \mathcal{X}_1 \cap \cdots \cap \mathcal{X}_n = \varnothing \mathcal{X} = \{t\} \cup \mathcal{X}_1 \cup \cdots \cup \mathcal{X}_n \quad E = E_{\mathcal{T}}(\mathcal{T}_1) \\ C = C_1 \cup \cdots \cup C_n \cup \{\mathcal{T}_1 = (E\ (\rightarrow\ (\mathcal{T}_2\ \cdots\ \mathcal{T}_n)\ t))\}\end{array}}{\Gamma \vdash (e_1\ e_2\ \cdots\ e_n) : t \mid_{\mathcal{X}} C} \quad \text{(T-App)}$$

$$\frac{\begin{array}{l}\Gamma \vdash x_1 : t_1 \mid_{\varnothing} \varnothing \wedge \cdots \wedge \Gamma \vdash x_n : t_n \mid_{\varnothing} \varnothing \quad \neg io(C) \\ \Gamma \vdash e : \mathcal{T}_0 \mid_{\mathcal{X}} C_0 \quad C = \{\mathcal{T} = (\texttt{Pure}\ (\rightarrow\ (t_1\ \cdots\ t_n)\ \mathcal{T}_0))\} \cup C_0\end{array}}{\Gamma \vdash (\texttt{lambda}\ (x_1\ \cdots\ x_n)\ e) : \mathcal{T} \mid_{\mathcal{X}} C} \quad \text{(T-Lambda)}$$

$$\frac{\begin{array}{l}\Gamma \vdash x_1 : \mathcal{T}_1 \mid_{\varnothing} \varnothing \wedge \cdots \wedge \Gamma \vdash x_n : \mathcal{T}_n \mid_{\varnothing} \varnothing \quad \Gamma \vdash e : \mathcal{T}_0 \mid_{\mathcal{X}} C_0 \quad E = E_{\mathcal{T}}(\mathcal{T}) \\ (E = \texttt{Pure}) \Rightarrow \neg io(C) \quad C = C_0 \cup \{\mathcal{T} = (E\ (\rightarrow\ (\mathcal{T}_1\ \cdots\ \mathcal{T}_n)\ \mathcal{T}_0))\}\end{array}}{\Gamma \vdash (\texttt{defun}\ \text{name}\ (x_1\ \cdots\ x_n)\ \mathcal{T}\ e) : \mathcal{T} \mid_{\mathcal{X}} C} \quad \text{(T-Defun)}$$

図 4　BLisp の制約型付け規則（一部のみ）

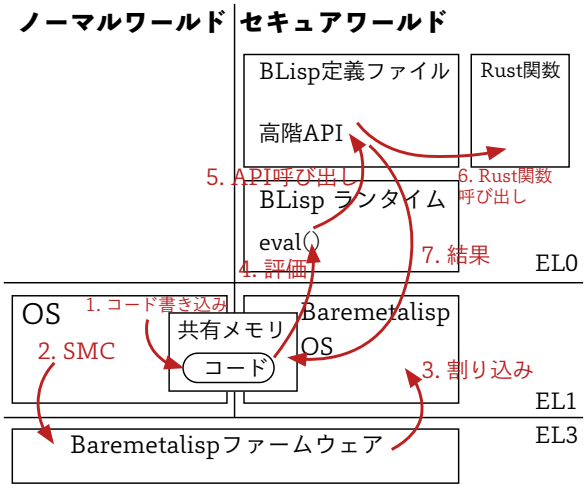

図 5　ノーマルワールドから BLisp 式の評価

ソースコード 4　API 定義の例

```
1  (export callback-test (x)
2      (IO (-> (Int) Int))
3      (call-rust x 0 0))
4
5  (export lambda-test (f)
6      (Pure (-> ((Pure (-> (Int) Int))) Int))
7      (mul2 (f 2)))
8
9  (defun mul2 (x) (Pure (-> (Int) Int))
10     (* 2 x))
```

を呼び出す。5. eval 関数は指定されたコードを実行し、必要な場合 API 定義ファイルで定義された関数を呼び出す。6. さらに、もし必要ならば Rust の関数を呼び出し、7. 最後に結果を共有メモリへ書き込む。

4.3　API 定義

本節では BLisp を用いた API 定義について具体例を示して説明する。ソースコード 4 は API 定義の例である。ここでは、1 と 5 行目で、callback-test と lambda-test と言う外部に公開する関数を定義している。9 行目で定義される mul2 関数は、API 定義ファイルのみで利用可能なローカル関数である。ここで、3 行目の rust-call 関数は Rust 関数の呼び出しを行うための組み込み IO 関数である。

ソースコード 5　eval の例

```
1  ; 成功
2  (lambda-test (lambda (x) (* x x)))
3
4  ; 成功
5  (callback-test 5)
6
7  ; 型付けエラー。IO 関数を渡している。
8  (lambda-test callback-test)
9
10 ; 型付けエラー。IO 関数をλ式の中で呼んでいる。
11 (lambda-test (lambda (x)
12     (let ((_ (callback-test 1)))
13         x)))
14
15 ; 型付けエラー
16 ; export されていない関数を呼んでいる。
17 (mul2 10)
```

ソースコード 5 は、eval 関数に渡す式の例となる。2 と 5 行目の式は、外部公開された関数の lambda-test と callback-test を呼び出しているのみであるため、これら式の評価は成功する。一方、それ以降の式は型付けエラーで失敗する。8 行目の式は、Pure 関数を受け取る高階関数の引数に、IO 関数である callback-test を渡しているため、11 行目の式は、IO 関数をλ式の中で呼び出しているため、型付けエラーとなる。また、17 行目の式は、export されていない式を呼んでいるため型付けエラーとなる。

このように Baremetalisp では型システムと効果系によ

表 2　TrustZone 用 TEE 実装との比較

	Trusted OS の型安全性	Trusted App の型安全性	動的計算コード注入の安全性
Baremetalisp	○	○	○
OP-TEE [15]	×	×	×
RustZone [7]	×	○	×
TLR [8]	×	△	×

り関数の利用に制限を設けることが可能となり、ぶらさがりポインタなどのポインタ関連のエラーも排除することができる。これにより、従来は難しかった、ノーマルワールドからセキュアワールドへの IO のない任意の計算コードの安全な注入を可能とする。

5. 実装

Baremetalisp は一部ブート用のコードをアセンブリで実装しているが、それ以外は、ファームウェア、OS、BLisp の全てを Rust 言語で実装している。開発は、まだ初期段階であるが、シングルボードコンピュータの Raspberry Pi 4 と Pine 64+、及び QEMU 上での動作を確認している。ソースコードは GitHub にて MIT ライセンスで公開しており [12], [13]、BLisp のデバッグのため既存の OS 上でスタンドアロン動作する BLisp-repl [14] も用意している。

直接のメモリ操作が必要な MMU、メモリアロケータ、MMIO などの箇所は unsafe となってしまい、型システムによる安全性は低下してしまうが、Rust にはスライスという概念があるためバッファオーバランなどは防ぐことができる。BLisp 部分は、実行時にマークアンドスイープに似たオブジェクト管理を行う箇所のみ unsafe であるが、それ以外の構文解析、意味解析などの大部分は型安全である。

6. 評価

本節では、提案方式と既存 TEE ソフトウェアを比較し、定性的な評価を行う。比較対象としては、OP-TEE と、本研究と類似手法の RustZone、及び Trusted Language Runtime (TLR) の比較を行う。TEE 基盤を実現するソフトウェアは他にもあるが、プロプライエタリな物が多く正確な比較が難しいため、ここではオープンソースである OP-TEE を対象とする。

比較項目は、Trusted OS と Trusted App の型安全性、動的な計算コード注入の安全性についてである。型安全性は、メモリ安全性やエラーハンドリング忘れなどの重大な実行時エラーに直結するため、非常に重要である。動的な計算コード注入の安全性とは、実行時にコードを注入すること自体のサポート及び、注入しても TEE やノーマルワールド内ソフトウェアの健全性及び、想定外のメモリへのアクセスが無いことが保証されていることと定義する。

提案方式の Baremetalisp では、Trusted OS と Trusted App の型安全性及び、動的な計算コード注入の安全性は保証されている。これは、Rust 言語を用いて実装したためと、効果系を適用した言語である BLisp の設計と実装を行ったためである。ただし、メモリアロケータなど、メモリを直接扱う必要がある部分については型安全では無いため、今後検証を行う必要がある。

OP-TEE は全て C 言語で実装されているため、OS、アプリケーションの両方とも型安全では無い。また、動的な計算コード注入も可能ではあるが、その安全性については全く保証されていない。RustZone は OP-TEE 上で動作するアプリケーション実装のための提案であり、Rust 言語を用いて実装するため、Trusted App に関しては型安全となる。ただ、動的な計算コード注入については RustZone の対象外となる。

TLR は.NET ランタイムを TEE 内で動作させるため、重大なメモリエラーなどは防ぐことができる。しかし、.NET で用いられる C#言語は動的型付け言語であり型安全ではなく、null 値などの実行時エラーは発生する。一方、F#言語は静的型付け言語であり型安全であるため、F#で実装した場合は Trusted App の型安全性は保たれる。TLR もあくまで Trusted App 用であるため、OS の型安全性は保たれず、動的な計算コード実行もサポートしていない。一方、TLR ではデータアクセスに関して細かい粒度で権限を設定できると言う特徴がある。

7. 議論

本節では、unsafe なコードの考察とメモリ関連のエラーについての考察を行う。

7.1 unsafe なコードについて

J. V. Bulck らの報告によると Rust 言語を用いた場合でも、unsafe なポインタ演算がオーバーフローしてしまい脆弱性につながると報告されている [16]。これは、Rust 言語により実装された Intel SGX 向けの TEE ソフトウェアの Rust-EDP [17] にて実際に見つかった脆弱性である。このように、unsafe のコード部分は型システムによる保護は弱くなるが、Baremetalisp ではノーマルワールドから渡された値は、ポインタ演算に利用しないと言う保守的な設計としているため、この種のバグは発生しない。

ただし、先にも述べたとおり、メモリアロケータなどのコードは型安全とは言えないため、この部分については将来的にシンボリック実行解析や定理証明などを用いて検証する必要がある。

7.2 メモリ関連のエラー

アドレス空間配置のランダム化 (ASLR) やスタッククッキーは、バッファオーバランなどによって引き起こされる問題を軽減する手法であり、Linux などではカーネルのアドレス空間もランダム化されている。しかし、多くの

TrustZone 向け実装では ASLR などの実装がされていないと指摘されている [1]。我々の設計・実装ではバッファオーバランからの保護は型システムによって行われており、そもそもバッファオーバランは発生しない。したがって、Baremetalisp においては ASLR などの重要度は低いと考えられる。また、ノーマルワールドからのアドレスを信頼しない事で、ブーメランバグも防ぐことができる。

8. まとめと今後の課題

本論文では、Armv8 TrustZone 向けの新たな TEE 実現基盤となる Baremetalisp を提案した。Baremetalisp はファームウェアと OS からなる Baremetalisp TEE と、API 定義用プログラミング言語の BLisp から構成される。従来の TrustZone 向けソフトウェアは C 言語で実装されており、その安全性に問題があり、実際にいくつもの脆弱性が報告されていた [1]。そこで、Baremetalisp では型安全性の考えを中核におき、型安全なプログラミング言語の Rust を用いて設計と実装を行った。

また、BLisp には効果系を適用しているため、副作用のある関数と純粋な関数を完全に分離して扱うことができる。Edge Computing 等の環境では、TEE 内でも任意の計算を行うことが要求されるが、従来の TEE ソフトウェアでは設計上行うことが難しかった。しかし、BLisp では型システムと効果系により、IO の無い任意の計算をノーマルワールドからセキュアワールドに安全に注入可能となる。

しかし、Rust 言語を用いても unsafe なコード部分は型安全ではないため、これら箇所については何らかの検証が必要である。今後、シンボリック実行解析や定理証明などの技術を用いて、これらの検証を行っていく予定である。

現在は基礎部分に絞って実装を行っているため、実用までには足りない機能が多くあり、ノーマルワールドのソフトウェアとの連携などはまだできていない。今後は機能拡充するとともに、ノーマルワールドのソフトウェアと連携を可能にし、システムとしての完成度を上げていく必要がある。

BLisp はモジュール化しており、OS の実装と独立している。そのため、メモリアロケータさえ用意すれば、様々なプラットフォームでも動作する。現在、RISC-V での動作も確認済みだが、今後は様々なプラットフォームへの展開も行う。

謝辞

本研究の一部は文部科学省「Society5.0 に対応した高度技術人材育成事業成長分野を支える情報技術人材の育成拠点の形成 (enPiT)」、文部科学省の令和 2 年度「Society 5.0 実現化研究拠点支援事業」、ソフトバンク株式会社の助成を受けています。

参考文献

[1] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 2020.

[2] Aravind Machiry, Eric Gustafson, Chad Spensky, Christopher Salls, Nick Stephens, Ruoyu Wang, Antonio Bianchi, Yung Ryn Choe, Christopher Kruegel, and Giovanni Vigna. BOOMERANG: Exploiting the Semantic Gap in Trusted Execution Environments. In *24th Annual Network and Distributed System Security Symposium, NDSS 2017, San Diego, California, USA, February 26 - March 1, 2017*. The Internet Society, 2017.

[3] CVE-2016-2431, Android One devices allows attackers to gain privileges via a crafted application. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-2431`.

[4] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 1 edition, February 2002.

[5] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[6] Arm Architecture Reference Manual Armv8, for Armv8-A architecture profile. `https://developer.arm.com/documentation/ddi0487/fb/`.

[7] Eric Evenchick. RustZone: Writing Trusted Applications in Rust. `https://www.evenchick.com/dl/slides/bh-asia-18-rustzone.pdf`, 2018. Blackhat 2018.

[8] Nuno Santos, Himanshu Raj, Stefan Saroiu, and Alec Wolman. Using ARM trustzone to build a trusted language runtime for mobile applications. In Rajeev Balasubramonian, Al Davis, and Sarita V. Adve, editors, *Architectural Support for Programming Languages and Operating Systems, ASPLOS '14, Salt Lake City, UT, USA, March 1-5, 2014*, pp. 67–80. ACM, 2014.

[9] Anil Madhavapeddy and David J. Scott. Unikernels: the rise of the virtual library operating system. *Commun. ACM*, Vol. 57, No. 1, pp. 61–69, 2014.

[10] Thomas Hallgren, Mark P. Jones, Rebekah Leslie, and Andrew P. Tolmach. A principled approach to operating system construction in Haskell. In Olivier Danvy and Benjamin C. Pierce, editors, *Proceedings of the 10th ACM SIGPLAN International Conference on Functional Programming, ICFP 2005, Tallinn, Estonia, September 26-28, 2005*, pp. 116–128. ACM, 2005.

[11] Redox OS. `https://www.redox-os.org/`.

[12] Baremetalisp. `https://github.com/ytakano/baremetalisp`.

[13] BLisp. `https://github.com/ytakano/blisp`.

[14] BLisp's Repl. `https://github.com/ytakano/blisp-repl`.

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

[16] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pp. 1741–1758. ACM, 2019.

[17] ENCLAVE DEVELOPMENT PLATFORM — Rust EDP. `https://edp.fortanix.com/`.