# Arm TrustZone for Armv8-Mを利用したマルチタスク対応CFIの検討

河田 智明[1]　本田 晋也[1]　松原 豊[1]　高田 広章[1]

**概要**：Return-Oriented Programming 等の制御フローに対する攻撃への対策として，制御フローを実行時に検査する CFI (Control-Flow Integrity) が知られている．本研究では計算機資源が制限された組込み向けプロセッサ上でセキュリティ機能の実装を支援するハードウェア機構の一つである TrustZone for Armv8-M を利用した，RTOS ベースのアプリケーションを対象とした軽量な CFI 手法 TZmCFI を提案する．

**キーワード**：組込みシステム, 制御フロー攻撃, リアルタイム OS, TrustZone

# Study On Multitasking-aware Control-Flow Integrity Based On TrustZone For Armv8-M

Tomoaki Kawada[1]　Shinya Honda[1]　Yutaka Matsubara[1]　Hiroaki Takada[1]

***Abstract:*** CFI (Control-Flow Integrity) is a class of techniques that allow the detection of control-flow attacks such as Return-Oriented Programming by employing run-time checks of the control flow. We propose a light-weight CFI scheme for RTOS-based applications, TZmCFI, which utilizes TrustZone for Armv8-M, a hardware-assisted security feature for embedded systems with tight resource constraints.

***Keywords:*** Embedded systems, control-flow attacks, real-time operating systems, TrustZone

## 1. はじめに

任意コード実行は攻撃対象のシステムが持つ全権限を攻撃者が獲得でき，さらにそれが権限昇格などの他の種類の攻撃を実行する手段となることから，ソフトウェアに対する攻撃の中では最も重大なものとして分類される．現在のソフトウェアシステムの多くは W⊕X（write XOR execute; 書き込み可能なメモリを実行不可にするメモリアクセス権の設定）等の対抗策を実装している．

しかし，W⊕X の存在下においても任意コード実行と同程度の効果を持つ Return-Oriented Programming (ROP) という攻撃手法が知られている [1]．この攻撃手法はコールスタックを破壊し既存のコードに制御フローを誘導することで不正な動作を行わせるものである．

ROP 等の制御フローに対する攻撃への対策として，制御フローを実行時に検査する CFI (Control Flow Integrity) が知られている．汎用システム向けの CFI 手法の研究は盛んに行われており [2]，幅広く使用されているコンパイラツールチェーン向けの実用的な実装 [3] も存在するが，組込みシステムではプロセッサアーキテクチャの機能的な差異や計算機資源の制約からこうした手法の適用が困難である．

近年の組込み向けプロセッサはセキュリティ機能の実装を支援するハードウェア機構を追加する傾向にある．本研究ではそうした機構の一つである TrustZone for Armv8-M を利用した，RTOS ベースのマルチタスクアプリケーションを対象とした軽量な CFI 手法 TZmCFI を提案する．

[1] 名古屋大学大学院情報学研究科
Graduate School of Infomatics, Nagoya University

## 2. Control Flow Integrity

CFI 手法の目的は正常な状態から逸脱した制御フローを実行時に検出することである．CFI 手法は正常な制御フローをモデル化する手法と実行時に制御フローを監視する手法の組合せによって実現される．

制御フローのモデル化は，基本的には実行対象のコードに対して静的解析を行い，何らかの基準（たとえば C++の言語仕様で未定義動作となる間接呼び出しを禁止する [3]）に基づいて制御フローの妥当性を定義することによって行われる．CFI 手法が異常な制御フローを排除する，すなわちモデル化された正常な制御フローが実際の正常な制御フローをどの程度近似するか，という性質は精度 (precision) と呼ばれる．たとえば，[2] で示した手法はプログラムのバイナリコードに対して静的解析を行うことで得られる制御フローグラフに基づいているが，間接呼び出しのためのポインタ解析は限界があるため保守的なモデルとなり，精度が制限されてしまう．精度を改善する方法の一つは制御フローのモデルに動的な要素を持たせることである．同論文では shadow stack（2.1 節で述べる）という動的な状態を追加することで精度を改善している．また他の動的な状態に基づいた手法として，プログラムの最近の分岐履歴に基づいて実行パスの妥当性を決定する手法が提案されている [4]．

実行時に制御フローを監視する手法は，プログラム中の間接呼び出しの直前に実行時検査用のコードを埋め込む，インライン検査が主流である．インライン検査の埋込はプログラムの機械語を書き換えや，コンパイラツールチェーンへの修正によって実現される．他の監視手法として，独立したタスクから本体のプログラムのコールスタックを定期的に読み取り検査する手法も提案されている [5]．この手法の利点としては検査を行うタスクが本体と独立してスケジューリングされており，開発者がこのタスクのスケジューリングを自由に制御できるためハードリアルタイムシステムでの実用性が高い点であるが，一方で検査の頻度が不十分であれば不正な遷移を見落とす可能性がある．

制御フローグラフの辺は，forward edge（関数呼び出し），backward edge（関数復帰）に分類できる．この 2 種類の辺は静的解析や攻撃シナリオなどの特性がまったく異なることから，CFI 手法においては区別して扱うことが多い．Forward/backware edge を対象とした CFI をそれぞれ forward/backward-edge CFI と呼ぶ．

### 2.1 Shadow stack

Shadow stack は，コールスタックのコピーを隔離された環境で管理し，この情報に基づいて関数の復帰先を動的に制限することにより backward-edge CFI の精度を改善する機構である．Shadow stack を実装するためには，関数のコールサイトを修正し，想定される復帰先の番地が shadow stack に push されるようにする．さらに復帰サイトを修正し，ここでは shadow stack から要素を pop し，復帰先番地が想定されるものと一致していることを検証するようにする．

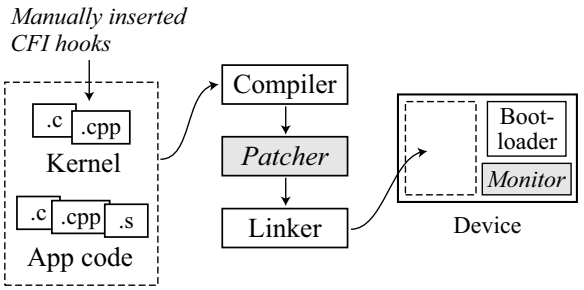


図 1　CFI 有効化のワークフロー

**Fig. 1**　The workflow for enabling CFI enforcement on an application.

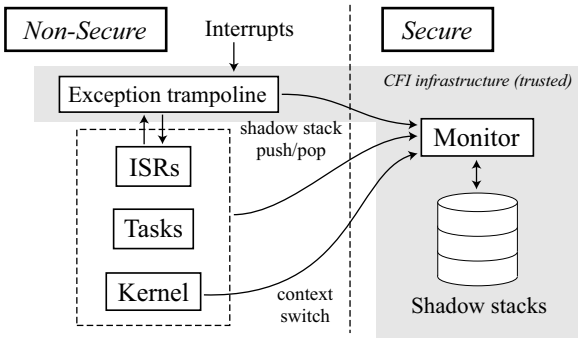


図 2　提案手法の実行時の構成図

**Fig. 2**　The run-time architecture of the proposed CFI scheme.

Shadow stack を実装するうえで，shadow stack を不正なアクセスから保護することはセキュリティ上重要である．保護を実現するためには，shadow stack に不正にアクセス可能なコードが到達不可能であることを保証する必要がある．CFI 機構の存在により実行可能なコードがコードセクションの範囲に制限されており，なおかつコードセクションが読み取り専用である場合，コードセクションに対する静的検査はそうした命令の存在を検出するのに十分である．

この方法が実用的であるためには，shadow stack のメモリ空間が通常のデータから完全に分離されていることが必要である．よく用いられるアプローチはプロセッサの特権レベルやそれに相当する機構を利用し，ユーザコードから shadow stack への任意のアクセスを禁止することである．たとえば，[6] では Cortex-R4F の Supervisor モードを利用した実装を示している．また，[7] は本研究と同様に Armv8-M プロセッサアーキテクチャ（3.1 節）の機能を利用して shadow stack を保護する方法を示している．

### 2.2 System-level CFI

CFI の手法の多くはユーザスペースで動作するコードを対象としている [2]．本研究が対象とするのはそれとは異なり，カーネル等を含むソフトウェアシステムのより広い範囲を CFI の保護対象とする．こうした CFI は [7][8] 等の例があり，本研究では system-level CFI と呼ぶことでユーザスペースのみと対象とした CFI（user-space CFI）と区別する．

[6] は組込みシステム向けの CFI であるが，shadow stack をカーネルからアクセス可能なメモリ領域に配置している，すなわちカーネルは信頼できるものと仮定しており，本研究の system-level CFI の定義に該当しない．[7] は system-level CFI であるが，マルチタスクは未対応である．

System-level CFI は trusted computing base を縮小できる利点があるが，割込みのような外的要因やコンテクスト切替えによる制御フローの変化に対応する必要がある．

#### 2.2.1 割込み

割込みは割込みが無効化された区間を除く任意の命令を実行中に発生する可能性がある．従って，割込みの復帰先として有効な命令の集合には実行可能なコードのほぼすべての命令が含まれることになるため，静的な制御フローグラフのみを用いた CFI 手法は有効ではない．2.1 節で述べた shadow stack を拡張し，割込みを一種の関数呼び出しのように扱う方法 [7] が有効である．

#### 2.2.2 マルチタスク対応

CFI をマルチタスクに拡張した素朴なモデルは，個々のタスク毎に独立した CFI である．静的な CFI 手法はマルチタスクアプリケーションでも無修正で適用可能である．一方 shadow stack など動的な状態を含む場合，そうした動的な状態をタスク毎に保持し，コンテクストスイッチ発生時には操作対象の状態を切り替える機構が必要となる [8]．

## 3. TZmCFI

本研究は Arm TrustZone for Armv8-M を利用した軽量な system-level CFI 手法の開発を目的としている．

最初に 3.1 節では Armv8-M の概要について説明する．その後，3.2 節で本手法が前提とする仮定について述べ，最後に 3.3 節で具体的な設計を述べる．

### 3.1 Armv8-M

Armv8-M は 2015 年に Arm 社が発表したマイクロコントローラ向けプロセッサアーキテクチャである．Armv8-M の主な特徴は，オプション機能として TrustZone for Armv8-M（以下 TZ-M と略す）に対応したことである．TZ-M はメモリ空間を Non-Secure/Secure 領域に分離して不正なメモリアクセスを防ぎ，高水準のセキュリティを確保する機能である．

これに対応するプロセッサアーキテクチャへの拡張は Armv8-M Security Extensions あるいは Cortex-M Security Extensions（以下 CMSE と略す）と呼ばれる．CMSE では新たに Non-Secure/Secure mode という実行モードが導入される．モード間の遷移はハードウェアにより管理されており，Non-Secure/Secure 間の不正な遷移を防ぎ確実な分離を実現する一方で，関数呼び出しを経由した低オーバヘッドな相互動作が可能である．

Armv8-M に関する詳細は [9] にある．

### 3.2 前提

本研究の CFI 手法の設計目標は CMSE を利用することで 2.2 節で説明した system-level CFI を低オーバヘッドで実現することである．提案手法は以下を前提としている．

(1) Secure mode で動作するコードは信頼する．
(2) CMSE による分離を無効化するソフトウェア・ハードウェア攻撃の存在は考慮しない．
(3) 適用対象のプログラムのコードは Non-Secure mode からは読み取り専用である．
(4) コンパイラは標準的な ABI に従う．
(5) 適用対象のプログラムのコードのソースコードが利用可能で，binary rewriting 手法に依存せずとも変更が可能である．
(6) 割込みベクタテーブルのベースアドレスは Non-Secure mode からは変更できない．

5 はソースコードが利用できないレガシーなアプリケーションやライブラリでの利用を困難にする．しかし，従来の CFI 手法が主に対象としてきた汎用システムと比較して規模の小さい組込みソフトウェアでは，ソースコードが利用できないコードが占める割合は小さいものと思われる．また，メモリレイアウトを保持しなければならない binary instrumentation ベースの手法 [2][7] よりも実装の自由度が増すため，たとえばインライン検査を効率的な方法（e.g., ソフトウェアトラップ命令の代わりに分岐命令で shadow stack の操作ルーチンを呼び出し）で実装でき，オーバヘッドの軽減が期待できる．

6 は Armv8-M の実装である Cortex-M23/M33 コアでは対応しており，ハードウェア構成次第で設定可能である．

### 3.3 設計

提案手法はインライン検査をベースとしており，間接呼び出し命令の直前に実行時検査を挿入することで制御フローの検査を行う．このインライン検査の挿入は適用対象のプログラムのビルドパイプラインの中間段階として行われる．

CFI の動的な要素として shadow stack を導入することで，高精度な backward-edge CFI を実現する．Forward-edge に対しては [3] 等の既存手法を用いる．

提案する CFI 手法は次の要素から成る（図 1）．

**Patcher** 適用対象のコードを書き換えインライン検査の挿入等を行うことで，CFI を有効化するツール．

**Monitor** CFI 手法の機能を支援する API を提供するランタイムライブラリであり，shadow stack の管理を行う．

Patcher では適用対象のコードに対し以下の変更を行う．

- 直接・間接関数呼び出しの直前に Monitor への呼び出しを挿入し，shadow stack への push が行われるようにする．
- 関数復帰命令の直前に同様に Monitor への呼び出しを挿入し，shadow stack の pop および復帰先の実行時チェックを追加する．
- 例外処理ハンドラの前後で shadow stack への push/pop が行われるよう，例外トランポリンを生成して既存の例外ハンドラをラップし，例外ベクタテーブルを置換する．

現時点のプロトタイプ実装では，Patcher はアセンブラコードを処理対象としている．対象のプログラムをコンパイルして得られたアセンブラを入力とし，これをパースして抽象表現に変換する．得られた抽象表現に対して修正操作を行い，結果をアセンブラとして出力することで機能する．アセンブラに対して操作を行うことの利点は，PC 相対アドレッシングが未解決な状態であるため，メモリレイアウトに影響するような加工が容易であることである．

Monitor は CMSE の Non-Secure/Secure mode を利用することにより対象のプログラムから隔離された空間で動作する（図 2）．Non-Secure mode から Secure メモリ領域を操作することは，Secure mode 側のコードが明示的に呼び出しを許可したライブラリ関数（この場合は shadow stack の push/pop など）を通じてのみ可能である．このため，CFI の状態が不意に破壊されることを防げる．

マルチタスクをサポートするために，それぞれがタスクと対応するように複数の shadow stack の作成ができる．適用対象のプログラムの OS は Monitor が提供する API を通じて最初にタスク ID の割り当てを行う．この割り当て処理はタスクの有効なエントリポイントの番地をデータとして渡すため data-oriented attack に対して脆弱である．このため，初期化完了後にはロックダウン状態に遷移し，それ以降のタスクの追加を禁止する必要がある．

システム起動後，OS がコンテクストスイッチを行う際には Monitor の API を呼び出すことで shadow stack の切替えを行う．

## 4. おわりに

コード注入が不可能な状況においても任意コード実行と同等の効力を持つ Return-Oriented Programming は組込みシステム，特に IoT エンドポイントのようなインターネットに接続されたシステムにおいては深刻な脅威である．

そうした攻撃への対策として，組込みシステム向けのセキュリティ支援機構の一つである TrustZone for Armv8-M を利用した CFI 手法を示した．現在プロトタイプ実装を開発中であり，将来的にはオーバヘッドやセキュリティ等の観点から評価を行う予定である．

**謝辞** 本研究の一部は JSPS 科研費 JP17K00075 の助成を受けたものです．

### 参考文献

[1] Shacham, H.: The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86), *Proceedings of the 14th ACM conference on Computer and communications security*, ACM, pp. 552–561 (2007).

[2] Abadi, M., Budiu, M. and Erlingsson, c.: Control-Flow Integrity, *ACM Conference on Computer and Communication Security (CCS)*, Alexandria, VA, pp. 340–353 (online), available from ⟨https://www.microsoft.com/en-us/research/publication/control-flow-integrity/⟩ (2005).

[3] Tice, C., Roeder, T., Collingbourne, P., Checkoway, S., Erlingsson, Ú., Lozano, L. and Pike, G.: Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM, *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, USENIX Association, pp. 941–955 (online), available from ⟨https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/tice⟩ (2014).

[4] van der Veen, V., Andriesse, D., Goktas, E., Gras, B., Sambuc, L., Slowinska, A., Bos, H. and Giuffrida, C.: Practical Context-sensitive CFI, *CCS 2015 - Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, Vol. 2015-October, Association for Computing Machinery (ACM), pp. 927–940 (online), DOI: 10.1145/2810103.2813673 (2015).

[5] Pike, L., Hickey, P., Elliott, T., Mertens, E. and Tomb, A.: TrackOS: a Security-Aware Real-Time Operating System, *Proceedings of the 16th Intl. Conference on Runtime Verification*, LNCS, Springer (2016). Preprint available at `http://www.cs.indiana.edu/~lepike/pub_pages/rv2016.html`.

[6] Brown, N.: Control-flow Integrity for Real-time Embedded Systems (2017).

[7] Nyman, T., Ekberg, J.-E., Davi, L. and Asokan, N.: CFI CaRE: Hardware-Supported Call andReturn Enforcement for Commercial Microcontrollers, *Research in Attacks, Intrusions, and Defenses* (Dacier, M., Bailey, M., Polychronakis, M. and Antonakakis, M., eds.), Cham, Springer International Publishing, pp. 259–284 (2017).

[8] Criswell, J., Dautenhahn, N. and Adve, V.: KCoFI: Complete control-flow integrity for commodity operating system kernels, *Security and Privacy (SP), 2014 IEEE Symposium on*, IEEE, pp. 292–307 (2014).

[9] 河田智明，本田晋也：ARM TrustZone for ARMv8-M を利用した軽量メモリ保護 RTOS，情報処理学会論文誌，Vol. 59, No. 2, pp. 762–774 (2018).