互換層を信頼しない互換レイヤー ―Zpolineを用いたtask内でのシステムコール捕捉とcapabilityによる権限の強制― （2026年10月30日版）

大野 清隆[1,a)]

概要：本論文では，Zpolineを活用した，無編集バイナリを実行可能なシステムコールレベルのLinux互換レイヤーの設計と実装を述べる．さらに，互換層を信頼せずにcapabilityで権限を強制できる仕組み，意味論の埋め合わせを設計手法として提案する．Fiasco.OC/L4Re上のプロトタイプによる評価と，Debianの上位100パッケージを用いた調査から，本手法で動作可能なバイナリの範囲および限界を明らかにする．実装と構造の解説を重点に置き，他者からの理解と再現が容易に行える知見の提供を目指す．

1. はじめに

従来のLinux互換レイヤーでは，無編集のLinuxバイナリの実行，カーネルを経由しないシステムコールの捕捉，capabilityによる権限の強制の3つを両立させることは困難であり，権限の強制はそもそも目的とされないことも多かった．これは，Linuxバイナリが発行するシステムコール命令を捕捉するために多くの場合カーネルのトラップを用い，Linux互換サーバーとの安全な通信にもカーネルを経由する必要があったためである．典型的なこの構造の例として，L4Linux[1]が挙げられる．L4LinuxはLinuxをL4ファミリー（Fiasco.OCやPistachio等）のユーザー空間の1つのプロセスとして移植し，汎用的なLinuxと堅牢なL4カーネルの特性の両立を目標に開発された．Linuxをそのまま移植したという性質上，高い互換性を持つが，動かすクライアントの求める権限とカーネル機能を動かすために必要な権限の和集合をL4Linuxプロセスが持つ．そのため，L4Linux上の1つのクライアントに必要な権限が小さくても，L4Linux自体が侵害されれば和集合のすべての権限が攻撃者の手に渡る．capabilityによって権限を細かく分けられるというマイクロカーネルの利点は，Linuxのクライアント単位では生かされない．

マイクロカーネル上のLinux互換レイヤーに対して，本論文は次の3つを同時に満たすことを目標とする．（1） 無編集のLinuxバイナリを実行できること．（2） システムコールの捕捉にカーネルを経由しないこと．（3） 個々のLinuxプロセスに，そのプロセスへ与えたcapabilityの範囲で権限を強制できること．（1）は再ビルドできない既存のソフトウェア資産をそのまま持ち込むために，（2）は捕捉を特定のカーネルの機構に依存させず，システムコールのたびのカーネル遷移を避けるために，（3）は脆弱性を含み得るLinuxバイナリと互換層が侵害されたときの影響をマイクロカーネルの隔離によって限定するために必要である．そして，この3つを同時に満たそうとすると，システムコール命令をプロセス内で書き換え，互換層を信頼せず，権限の強制をtaskの外に置くという本手法の構成が導かれる（3章）．既存の方式は，この3つのいずれかを満たさない．L4Linuxのように互換層を別のサーバーに置く構成は，Linuxプロセスのシステムコールを Fiasco.OCのvCPU[9]によってカーネルを経由して受け取るため（2）を満たさず，前述のとおり（3）も満たさない．Starnix[5]はLinuxプロセスと互換層を同じプロセスに置くが，捕捉はZirconカーネルのrestricted modeに依存しており，（2）を満たさない．Bascule[3]やGraphene[4]などのライブラリOSは，互換層を非信頼側に置き，隔離をその外側で強制する点で本論文に近い．しかし，いずれもlibcを改変してシステムコールを置き換えるため，Basculeは互換のlibcを動的リンクするアプリケーションに限られて（1）を満たさず，Grapheneはlibcを経由しないシステムコールをカーネルのseccompで捕捉するため，その経路では（2）

[1] 開智所沢中等教育学校
[a)] kamezukidaga@gmail.com

ⓒ2026 Information Processing Society of Japan

を満たさない．これらとの詳しい比較は 8 章で述べる．

本論文では，Zpoline[8] によってこの 3 つを両立させる．Zpoline はロード時にバイナリ中のシステムコール命令を `call *%rax` へ書き換え，0 番地に置いたトランポリンを経由してプロセス内のハンドラーへ制御を移す．本手法はこのハンドラーを互換レイヤー（以下，shim）とし，Linux のシステムコールをマイクロカーネルの capability 呼び出しへ変換する．shim は対象バイナリと同じ非信頼の task に置き，信頼の対象には含めない．権限の上限はその task に与えた capability で決まり，その検査は task の外にあるカーネルとサーバーが行う．したがって，shim が侵害されたとしても，影響は task に与えた capability の範囲に閉じ込められる．shim の役割は権限の判定ではなく，capability の有無や上限による結果を Linux の意味論（エラー番号など）へ変換することである．

本論文の貢献は次のとおりである．

- 互換レイヤーを対象バイナリと同じ非信頼ドメインに置きながら，権限の上限を capability で強制する互換レイヤーの構成を示す．
- Linux の意味論と capability 操作の差を，プロセス内の shim とネイティブ IPC に分けて埋める設計手法を示す．
- Linux 上のプロトタイプを用いて Debian の上位 100 パッケージと言語ランタイムで書き換え方式の互換性を調べ，非互換の多くは方式ではなく Linux での注入の手段と観測に由来することを示し，方式に固有の非互換として red zone の破壊，システムコールの実行位置への依存，0 番地の使用による NULL 参照の検出の失敗を示す．
- Fiasco.OC/L4Re 上にプロトタイプを実装し，`write`，`openat`，`read`，`mmap` などの結果が，task に与えた capability の有無と上限によって決まることを確認する．

また，実装と構造の解説に重点を置き，他者が理解し再現できる知見の提供を目指す．

以降，2 章で背景として capability ベースのマイクロカーネルと Zpoline を説明し，3 章で設計目標と脅威モデルを述べる．4 章で設計を，5 章で Fiasco.OC/L4Re 上の実装を述べる．6 章では Linux 上のプロトタイプを用いて書き換え方式の互換性を調べ，7 章で Fiasco.OC/L4Re 上のプロトタイプを評価する．8 章で関連研究を述べ，9 章でまとめる．

2. 背景

本章では，本手法が前提とする capability ベースのマイクロカーネルと，その 1 つである Fiasco.OC および L4Re，そしてシステムコールの捕捉に用いる Zpoline について述べる．

2.1 capability ベースのマイクロカーネル

capability ベースのマイクロカーネルでは，カーネルオブジェクトへの操作は，そのオブジェクトを指す capability を持つ場合にのみ許される．capability はカーネルが管理する表に格納され，ユーザー空間のプログラムは表の添字によってのみそれを参照するため，capability を偽造することはできない．新たな capability を得る方法は，生成したオブジェクトの capability を受け取るか，すでにそれを持つ者から IPC で委譲されるかに限られる．したがって，あるプログラムが行使できる権限は，そのプログラムが持つ capability の集合によって上から抑えられる．seL4[6]，Fiasco.OC，Zircon などがこの方式を採る．

この性質により，プログラムごとに必要最小限の capability だけを与えることで，侵害されたプログラムの影響をその範囲に閉じ込められる．ただし，この保証が有効なのは capability を与える単位，すなわち後述する task の単位である．1 章で述べた L4Linux では，各 Linux プロセスは別の task で動作するが，資源へのアクセスは L4Linux の task が持つ capability を通じて行われるため，権限は Linux プロセスごとではなく L4Linux 全体に与えられる．

2.2 Fiasco.OC と L4Re

Fiasco.OC は L4 ファミリーの capability ベースのマイクロカーネルであり，L4Re[7] はその上で動作するユーザー空間の実行環境である．Fiasco.OC における保護の単位は task であり，task は仮想アドレス空間と capability の表を持つ．同じ task の中のスレッドはアドレス空間と capability の表を共有するため，task 内のコード同士は互いに隔離されない．スレッド，IPC ゲート，task などのカーネルオブジェクトは，それぞれ capability を通じて操作され，サーバーへの要求は IPC ゲートの capability を用いた IPC として行われる．

L4Re では，メモリは dataspace として抽象化され，task ごとの region manager が dataspace を仮想アドレス空間に配置する．ルートタスクの Moe は，物理メモリの割り当て，起動時に与えられたファイルを公開する ROM の名前空間，ログ出力などのサービスを提供する．Ned はスクリプトに従って task を起動するローダーであり，起動する task ごとに，名前を付けた capability の集合と，メモリやカーネルオブジェクトの割り当て量の上限を与えられる．本論文で「task に与えた capability」と呼ぶものは，この Ned の設定によって起動時に task の表へ置かれる capability を指す．

x86-64 版の Fiasco.OC では，`syscall` 命令がネイティブのシステムコールの入口として使われる．そのため，Linux バイナリをそのまま実行すると，Linux のシステムコール番号と引数が Fiasco.OC のシステムコールとして解釈され，Linux としては意味のない動作になる．一方で Fiasco.OC

ⓒ2026 Information Processing Society of Japan

は，この命令をカーネルで捕捉してユーザー空間へ渡す機構も備える．vCPU がユーザーモードで別の task を実行している間に発行された `syscall` 命令は，カーネルのシステムコールとしては扱われず，例外として vCPU の入口へ届けられる．また，カーネルの構成で有効にできる alien スレッドでは，スレッドが発行する `syscall` 命令ごとに，例外 IPC が例外ハンドラーへ送られる．本手法はこれらの機構を用いず，カーネルを経由せずに捕捉を task の中で完結させる（3.2 節）．

2.3 Zpoline

Zpoline[8] は, x86-64 においてバイナリの書き換えによりシステムコールを捕捉する手法である．x86-64 の `syscall` 命令（`0f 05`）と `call *%rax` 命令（`ff d0`）はどちらも 2 バイトであるため，命令列の長さや後続の命令の位置を変えずに置き換えられる．Zpoline はロード時に実行可能領域を逆アセンブルして `syscall` 命令を探し，`call *%rax` へ書き換える．

`syscall` 命令の実行時，`rax` にはシステムコール番号が入っている．したがって書き換え後の `call *%rax` は，システムコール番号と同じ番地，すなわち 0 番地付近へ分岐する．Zpoline は 0 番地からシステムコール番号の最大値までを `nop` 命令で埋め，その直後にフック関数への分岐を置く．どの番号で分岐しても `nop` の列を滑り降りてフック関数に到達し，フック関数は `rax` からシステムコール番号を得る．処理後は `call` が積んだ戻り番地へ戻るため，元の `syscall` 命令の直後から実行が再開される．

この過程はすべて同じプロセスの中で完結し，システムコールの捕捉のためにカーネルへ遷移しない．一方で，0 番地にトランポリンを置く必要があり，Linux では `vm.mmap_min_addr` による制限の緩和が必要になる．これに対して Fiasco.OC/L4Re では，task が持つ capability の範囲内の操作だけで 0 番地へマップでき，追加の特権を必要としない（5 章）．また，ロード時に存在しないコードや，逆アセンブルで正しく識別できない命令は書き換えの対象から漏れる．このような，書き換えによる方式そのものに起因する制約は 6 章で整理する．

3. 設計目標

本章では，1 章で挙げた 3 つの要件を満たすべき理由を述べ，3 つを同時に満たすことから本手法の構成が導かれることを示す．

3.1 無編集の Linux バイナリの実行

本論文における無編集とは，対象バイナリと，それが動的リンクする共有ライブラリおよび動的リンカのファイルを変更も差し替えもせず，再ビルドや再リンクも行わないことを指す．ロード時にメモリ上で行う命令の書き換えはファイルを変更しないため，これに含まれる．

この要件の第 1 の理由は，再ビルドできないソフトウェアが現実に多く存在することである．配布形式のバイナリやディストリビューションのパッケージを再ビルドせずに実行できれば，既存のソフトウェア資産をパッケージ管理ごと持ち込める．第 2 の理由は，libc の協力を前提とすると，互換性の範囲が libc を経由するコードに限られることである．Go のランタイムや，Rust の rustix，静的リンクされた musl，インラインアセンブリなどは，libc を経由せずにシステムコール命令を直接発行する．改変した libc を入口とする方式では，これらのシステムコールは互換層を通らない．第 3 に，libc の改変版を上流の変更に追従させ続ける保守の負担を避けられる．

3.2 カーネルを経由しないシステムコールの捕捉

この要件の理由は，捕捉を特定のカーネルの機構に依存させず，システムコールのたびのカーネル遷移を避けることである．2.2 節で述べたように，Fiasco.OC は vCPU のユーザーモードや alien スレッドによって，Linux のシステムコールをカーネルで捕捉する機構をすでに備える．L4Linux は vCPU のユーザーモードで対象を別の task として実行するため，互換層の task は対象のアドレス空間やスレッドを操作する capability を持ち，1 章で述べた権限の和集合の問題が生じる．alien スレッドでは例外ハンドラーを同じ task のスレッドにでき，互換層を対象と同じ task に置ける．しかしこの場合も，システムコールのたびにカーネルに入り，例外 IPC を介してハンドラーとの間で往復する．また，これらは特定のカーネルに固有の機構であり，alien スレッドはカーネルの構成で明示的に有効にしなければ使えない．Starnix も，Zircon カーネルに restricted mode を追加することで Linux のシステムコールを捕捉している．捕捉をカーネルを経由せずに task の中で完結させれば，こうしたカーネルの機構に依存せず，カーネルを最小に保つというマイクロカーネルの設計原則と両立し，他の capability ベースのマイクロカーネルにも移植しやすい．

加えて，Linux の意味論の多くはプロセス内で処理を完結できる．ファイル記述子の表の操作やアドレス空間の管理情報の更新などは shim だけで処理でき，外部のサーバーとの IPC が必要になるのは，task 外の資源に触れる場合に限られる．トラップによる方式では，すべてのシステムコールについて，トラップ，ハンドラーへの例外 IPC，その応答という経路をたどり，少なくとも 2 回カーネルを経由する．カーネル遷移のコストは 7.5 節で示す．

3.3 capability による権限の強制

Linux バイナリは規模が大きく，その多くはメモリ安全でない言語で書かれているため，脆弱性を含む前提で扱う

ⓒ2026 Information Processing Society of Japan

必要がある．マイクロカーネル上で実行しても，侵害時の影響がプロセス単位で限定されなければ，マイクロカーネルの隔離の利点は失われる．1 章で述べた L4Linux の権限の和集合の問題は，この利点が失われる典型例である．

互換層自体も同様である．Linux の意味論は巨大であり，それを再現する互換層もまた大きく，不具合を含む前提で扱う必要がある．互換層を信頼の対象に含めると，信頼しなければならないコードの量が増える．したがって，権限の強制は互換層に依存せず，互換層の外で行われなければならない．

3.4 3 つの要件から導かれる構成

3 つの要件を 2 つずつ組み合わせると，本手法の構成が順に決まる．

- （1）と（2）:libc の協力を前提とせず，かつカーネルを経由せずにシステムコールを捕捉するには，システムコール命令そのものをプロセス内で書き換えるほかない．本手法はこのために Zpoline を用いる．
- （2）と（3）: 互換層をプロセス内に置く以上，互換層は対象バイナリと同様に信頼できない．そのため，権限の強制は互換層の外にあるカーネルとサーバーが，capability に基づいて行う．
- （1）と（3）: 対象バイナリの協力も前提にできないため，権限の境界は，バイナリがどのような命令を実行しても有効な仕組みでなければならない．本手法では task とその capability の表がこの境界となる．

このように，3 つの要件を同時に満たそうとすると，システムコール命令をプロセス内で書き換え，互換層を信頼せず，権限の強制を task の外に置くという構成が必然的に導かれる．

3.5 脅威モデル

本手法は，対象バイナリが侵害されることを前提とする．攻撃者は対象バイナリの脆弱性などを通じて，その task の中で任意のコードを実行できるものとする．

3.5.1 信頼する対象

Fiasco.OC カーネル，L4Re のルートタスクである Moe，ローダーの Ned，および task に capability を提供するサーバーは正しく動作すると仮定する．

3.5.2 信頼しない対象

対象バイナリと，それと同じ task に置かれるすべてのコードとデータを信頼しない．これには shim，0 番地のトランポリン，動的リンカと共有ライブラリ，および shim が管理するファイル記述子の表やプロセスの状態が含まれる．task の中のコードはアドレス空間と capability の表を共有するため，攻撃者は shim のコードやデータを書き換えられる．また，書き換えから漏れた `syscall` 命令や自ら用意した命令列によって，shim を経由せずに Fiasco.OC のシステムコールを直接発行し，task の表にある capability を任意の引数で呼び出せる．終了コードなど，task が自ら申告する情報も信頼しない．

3.5.3 保証する性質

侵害された task が行使できる権限は，その task に与えた capability と，それらを通じて得られるものに限られる．これは capability の検査を task の外にあるカーネルとサーバーが行うことによる．task が capability を通じて新たに得られるものも，その capability を提供するサーバーの方針と割り当て量の範囲に限られる．たとえば名前空間の capability から得られるのは公開されたファイルの capability だけであり，factory から生成できるオブジェクトの量は factory に設定した上限で抑えられる．fork などで子 task を作る場合も，子 task へ渡せるのは親 task が持つ capability に限られるため，プロセスを増やしても権限の上限は広がらない．

3.5.4 保証しない性質

次の性質は本論文の範囲外とする．第 1 に，task 内の Linux の意味論の完全性である．攻撃者は shim が持つ状態を壊して誤動作させられるが，その影響は原則として task 自身に留まる．例外は，4.3 節で述べる fork によって状態を共有するプロセスの間であり，その範囲ではほかのプロセスの Linux としての動作も乱され得る．shim が行うポインターの検査や，`getpid` の結果のキャッシュなどは，Linux の意味論を再現するための仕組みであり，隔離の境界ではない．第 2 に，与えた割り当て量の範囲内での資源の消費である．task は与えられた優先度で CPU を使い続けたり，割り当て量の上限までメモリを確保したりできる．第 3 に，サイドチャネルによる情報漏洩と，カーネルおよび信頼するサーバー自体の不具合である．

4. 設計

本章では，3 章の要件と脅威モデルに基づく本手法の設計を述べる．4.1 節で全体構成を，4.2 節で Linux の資源と capability の対応による権限の強制を，4.3 節で Linux の意味論の埋め合わせを，4.4 節で shim が持つ状態と信頼の関係を述べる．

4.1 全体構成

本手法の構成を図 1 に示す．Ned は，本手法の実行プログラム（以下，ランタイム）を 1 つの task として起動し，その task に名前を付けた capability の集合を与える．ランタイムは対象の Linux ELF を読み込み，実行可能な領域を走査して `syscall` 命令を `call *%rax` へ書き換えたうえで，0 番地にトランポリンを配置して対象の入口へ制御を移す．以降，対象バイナリ，ランタイム，shim はすべて同じ task の中で動作する．

対象が `syscall` 命令を実行していた箇所に到達すると，

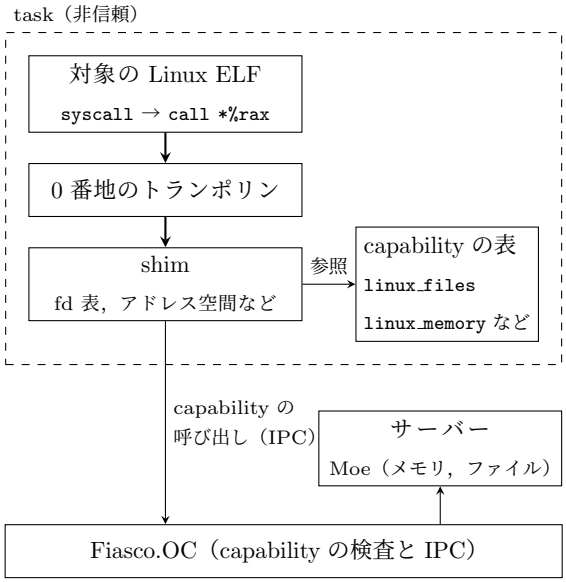

図 1 本手法の全体構成．破線の枠の外にあるカーネルとサーバーが，task に与えた capability の範囲で要求を検査する．

`call *%rax` によって 0 番地のトランポリンへ分岐し，そこから shim の入口へ進む．shim は対象のレジスタと浮動小数点の状態を保存し，ランタイム用の実行文脈に切り替えてから，システムコールを処理する．処理には，shim の中だけで完結するものと，task に与えられた capability を呼び出すものがある．後者では，capability の検査は task の外にあるカーネルとサーバーが行い，shim はその結果を Linux の戻り値やエラー番号に変換して対象へ返す．

4.2 Linux の資源と capability の対応

本手法では，Linux の資源の種類ごとに，それを提供する capability を task に与える．プロトタイプでは，標準出力にはログサーバー，ファイルには公開するファイルだけを登録した名前空間，匿名メモリには割り当て量を制限した Moe のメモリ割り当て元，子プロセスには割り当て量を制限した factory，CPU にはスケジューラーの capability を用いる．Linux のシステムコールが資源に触れる場合，shim は対応する capability を呼び出すことでのみ処理を行う．

この対応において，判断を行うのは shim ではなく，task の外にあるカーネルとサーバーである．たとえば非公開のファイルが `ENOENT` になるのは，名前空間にそのファイルが登録されていないためであり，割り当て量を超えた `mmap` が `ENOMEM` になるのは，Moe が割り当てを拒否するためである．capability が与えられていない場合は shim が `EACCES` を返すが，task の表に capability がない以上，shim を迂回しても資源には到達できない．すなわち shim の役割は，capability の有無や拒否の結果を Linux の意味論に翻訳することであり，権限の判定ではない．

ここで，task が持つ capability には，ランタイム自身の起動に必要なもの（起動用の ROM 名前空間，メモリの割り当て元，カーネルの factory，ログなど）も含まれる．3.5 節の脅威モデルにおける権限の上限は，これらを含む task 全体の capability である．一方，侵害されていない対象から見える Linux としての権限は，Linux の資源に対応する capability だけで決まるようにする．そのため shim は，Linux の資源に対応する capability がない場合に，起動用の ROM 名前空間やランタイム用の割り当て元など，より広い capability で代用しない．代用を許すと，構成した者が Linux に与えたつもりの権限よりも広い権限で対象が動作することになり，設定と動作が一致しなくなる．

fork などで子 task を作る場合，子 task には親 task が持つ capability のうち必要なものを，必要な権利で委譲する．子 task のために新たな capability の種類は追加せず，子孫は家族単位の割り当て量を共有する．したがって，プロセスを増やしても，権限の上限と資源の総量は最初に与えた範囲を超えない．プロトタイプで各 task に与えた capability とその権利は，7 章で示す．

4.3 意味論の埋め合わせ

Linux のシステムコールの意味論は，capability の操作と一対一には対応しない．本手法では，スレッド，プロセス，アドレス空間，fd 表，ファイルシステムの情報をそれぞれ別の状態として shim の中に持ち，`clone` の指定に応じて共有または複製することで，この差を埋める．

これらの状態に対する処理は，次の 2 つに分けられる．

4.3.1 shim の中で完結する処理

fd 表の操作，ファイル記述の読み取り位置の管理，`mmap` の領域の管理情報の更新，シグナルマスク，umask，`getpid` などは，shim が持つ状態だけで処理できる．同じ task の中のスレッドの間では，futex の待機と起床も shim の中で処理する．

4.3.2 capability の呼び出しが必要な処理

出力，ファイルの参照，メモリの確保と配置，スレッドや task の生成は，task に与えた capability を呼び出して行う．`CLONE_VM` を指定した `clone` は同じ task の中に新たなスレッドを作り，アドレス空間を共有する．`CLONE_VM` を指定しない `clone` や `fork` では，制限した割り当て量の factory から新たな task を作り，親のメモリの内容を子のメモリへ複製する．子 task は親のシステムコールの直後から実行を再開する．

4.3.3 例外としての fork

fork は，複数の task にまたがる Linux の意味論を必要とする点で例外である．継承したファイル記述の読み取り位置，子プロセスの終了の通知と回収，孤児の引き継ぎなどは，1 つの task の状態だけでは再現できない．プロトタイプでは，これらを fork で生じたプロセスの shim の間で，共有メモリと親への通知用の IPC ゲートを用いて協調し

ⓒ2026 Information Processing Society of Japan

て処理する．これは fork を実現するための現時点の手段であり，共有に用いるメモリや IPC ゲートも，それらのプロセスに与えた capability の範囲で作られる．

一方，現時点で埋められていない意味論もある．コピーオンライトによる `fork`，`execve`，`vfork`，非同期なシグナルの配送，`MAP_SHARED`，ファイルの作成と書き込みは未対応である．

4.4 shim の状態と信頼

4.3 節で述べた shim の状態は，Linux の意味論を再現するための協調用の状態であり，隔離の境界ではない．たとえば `getpid` は，高速化のためにスレッドごとの領域に PID を保持し，shim の通常の処理を経ずに返す．shim が行うポインターの検査も，Linux の `EFAULT` を再現するためのものである．これらはいずれも同じ task の中にあるため，侵害された対象は書き換えられる．

例外として，fork で状態を共有するプロセスの間では，侵害されたプロセスが共有メモリ上の状態を壊し，ほかのプロセスの Linux としての動作を乱し得る．しかし，いずれの場合も，攻撃者が行使できる権限は task に与えた capability の範囲を超えない．本手法は，意味論の再現を shim に任せる一方で，権限の強制を shim の外に置くことで，shim が壊れても権限の上限が保たれるように設計されている．

5. 実装

本章では，4 章の設計を Fiasco.OC/L4Re 上に実装したプロトタイプについて述べる．5.1 節で概要を，5.2 節でロードと書き換えを，5.3 節でシステムコールの入口を，5.4 節で資源の提供を，5.5 節でスレッドと fork を，5.6 節で実装上の制約を述べる．

5.1 概要

ランタイムは，L4Re のネイティブなプログラムとして，C++，C，アセンブリで約 4,800 行で実装した．命令の解析には Zydis 4.1 を用いる．対象は，x86-64 向けに musl で静的または動的にリンクした Linux ELF である．実装にあたって，Fiasco.OC，Moe，Ned には変更を加えていない．L4Re 上と Linux 上のプロトタイプのソースコードと試験は，入手先 ⟨`https://github.com/kamedaga/zcomp`⟩ で公開している．ランタイムが用いるのは，region manager，dataspace，factory，IPC ゲートなど，L4Re のプログラムが通常使う API だけである．

5.2 ロードと書き換え

ランタイムは，対象の ELF の各ロード可能なセグメントを新たに確保した dataspace へコピーし，BSS をゼロで埋める．実行可能なセグメントは，まず書き込み可能な別名の番地に配置し，そこで Zydis による走査と `syscall` 命令の書き換えを済ませる．その後，別名を外してから，対象の番地に読み取りと実行の権限だけで配置する．これにより，書き換え途中のコードが対象から見えることも，書き込みと実行が同時に可能な領域が残ることもない．走査で書き換えてよいかを判定できない候補が残った場合は，そのまま実行せずに対象の起動を拒否する．書き換えを見送った `syscall` 命令は，Fiasco.OC のシステムコールとして実行されてしまうためである．ただし，この判定は走査で見つけた候補に対するものであり，すべての書き換え漏れを防ぐものではない（6 章）．

0 番地のトランポリンも同じ手順で用意する．まず 1 ページの dataspace を確保し，書き込み可能な別名の上で，先頭からシステムコール番号の上限である 512 バイトを nop 命令で埋め，その直後に shim の入口への RIP 相対の間接 jmp 命令を置く．ページの残りは，範囲外の番号で実行が進まないよう，未定義命令で埋める．次に，別名を外して dataspace を読み取りと実行の権限で任意の番地に配置し，`L4::Task::map()` によって，自 task のそのページを自 task 内の 0 番地へマップする．この操作は自 task の capability に対する操作であり，ほかの権限を必要としない．マップの成功だけでは既存のマップとの衝突を検出できないため，最後に 0 番地の内容を読み出して確認する．

動的リンクされた ELF では，PIE でない main は ELF が指定する番地に配置し，PIE の main と interpreter は予約した範囲の空きから，アライメントを満たす番地を選んで配置する．初期スタックの補助ベクタには main のプログラムヘッダと interpreter の番地を置き，interpreter の入口へ移る．libc やそのほかの共有ライブラリは，interpreter 自身が `openat` と `mmap` で読み込む．再配置，コンストラクター，TLS の初期化は対象の musl が行うため，ランタイムは再配置の機能を持たない．interpreter を持たない静的 PIE も同じ方法で配置し，自己再配置は対象のスタートアップコードに任せる．ただし，書き換えはロード時に済ませるため，コードの領域への再配置を必要とする ELF は拒否する．実行中に `mmap` や `mprotect` で新たに実行可能になる領域も，別名の上で走査と書き換えを終えてから公開する．一度準備したコードを再び書き込み可能にする要求や，書き込みと実行を同時に求める要求は拒否する．

5.3 システムコールの入口

トランポリンから進んだ shim の入口は，GS からスレッドごとの領域を得て，そのスレッドの実行文脈と shim 用のスタックに切り替える．入口の時点では FS が対象の TLS を指しているため，対象のスタックやメモリに依存せずに参照できる，スレッドごとの領域が必要になる．そこで，ランタイムはスレッドの開始時に，Fiasco.OC の API で GS の基点をそのスレッドの領域に設定する．入口は汎用

ⓒ2026 Information Processing Society of Japan

レジスタ，フラグ，x87 と XMM の状態を保存し，FS をランタイムの TLS に戻してから，C++で書いたディスパッチ処理を呼ぶ．処理の後は FS を対象の値に戻し，保存した状態を復元して対象へ戻る．

Fiasco.OC では FS の基点の変更をカーネルの API で行うため，これをシステムコールごとに行うと，捕捉そのものはカーネルを経由しなくても，入口と出口でカーネル遷移が 2 回生じる．そこで，スレッドごとにランタイムと対象の TLS の基点を Fiasco.OC の GDT の API でセグメント記述子として登録しておき，入口と出口ではセグメントセレクターを FS に読み込むだけで切り替える．カーネルを呼ぶのは，スレッドの開始時と，対象が `arch_prctl` で TLS の基点を変えたときだけである．セグメント記述子に収まらない 4 GiB 以上の基点の場合に限り，従来のカーネルの API による切り替えを用いる．さらに `getpid` については，スレッドごとの領域に PID を保持し，入口のアセンブリだけで返す経路を設けた．この経路は FS の切り替えも状態の保存も行わない．シグナルの配送を待っている間や終了の処理中はこのキャッシュを無効にし，通常の経路を通す．

5.4 資源の提供

ファイルは，Ned の名前空間の機能で作った名前空間を通じて提供する．shim は絶対パスの先頭の/を除いて名前空間を辿り，重複した/と . を正規化し，.. は拒否する．相対パスの基点は名前空間の根に固定する．動的 ELF の interpreter や共有ライブラリも，この名前空間に登録したものだけが読める．名前空間で見つからないファイルについて，ランタイムの起動に用いた ROM の名前空間を代わりに探すことはしない（4.2 節）．ファイルの `mmap` は，読み取りの capability から対象用の dataspace へ内容をコピーして実現する．内容が変わらないファイルについては，名前空間から取得した capability とその写像を `close` の後も保持し，同じ名前の次の `openat` で再利用する．

匿名メモリの `mmap` は，Linux 用のメモリ割り当て元から dataspace を確保し，対象のために予約した範囲に配置する．Fiasco.OC では，メモリの解放が RCU の猶予期間の待機を伴うため，解放のたびに数ミリ秒を要する．そのため，匿名メモリは最大 1 MiB の dataspace からページ単位で切り出して割り当て，解放されたページは dataspace を返さずに保持し，内容をゼロで埋めてから次の `mmap` で再利用する．この大きさを超える要求には，個別に dataspace を確保する．保持している間も，そのメモリは元の割り当て元の割り当て量に計上されたままであり，割り当て量が不足したときは使われていない dataspace を返却する．shim は，予約した範囲，dataspace の所有，対象から見える番地と保護の区間を分けて管理し，部分的な `munmap` や `mprotect` では対象の区間だけを分割する．更新後の区間を先に作り，region manager の更新に成功してから管理情報を置き換えるため，割り当て量を超えて失敗した場合も元のマップは保たれる．`PROT_NONE` の区間は region manager に配置しない．読み取り可能な領域として残すと，ページフォルトの処理を通じて保護を迂回できてしまうためである．

標準出力と標準エラー出力も fd 表の記述として持ち，書き込みはログサーバーの capability の呼び出しになる．

5.5 スレッドと fork

`CLONE_VM` を指定した `clone` では，同じ task の中にネイティブなスレッドを作る．UTCB，ランタイムの TLS，ネイティブなスタックの初期化は L4Re の libpthread に任せ，対象が `CLONE_SETTLS` で指定した TLS とは分けて管理する．システムコールの処理は，`getpid` のキャッシュ経路を除き，1 つのネイティブな mutex で直列化する．futex では，待機中のスレッドの一覧を shim が持ち，値の照合と待機の登録をこの mutex の下で行った後，mutex を解放してネイティブな条件変数で眠る．起床では，この一覧から番地の一致するスレッドを選んで起こす．

`fork` と，`CLONE_VM` を指定しない `clone` では，次の手順で子 task を作る．

(1) 割り当て量を制限した factory から，新たな task，スレッド，region manager，親への通知用の IPC ゲートを作る．
(2) ELF の領域，スタック，TLS を含む対象のメモリを，新たな dataspace へすべてコピーする．コピーオンライトは用いない．
(3) 子 task にランタイムを新たにロードし，ランタイム用のスタックと TLS を用意する．親のランタイムのヒープや TLS はコピーしない．
(4) 子のランタイムが対象のメモリと fd 表を復元し，0 番地のトランポリンを子の shim の入口を指すように作り直す．そのうえで，保存したレジスタと FS を復元し，システムコールの直後から再開する．

子 task には，ファイルの名前空間，出力先，スケジューラーなど，親が持つ capability を必要な権利で委譲し，起動用の ROM の名前空間やランタイム用の割り当て元は追加しない．メモリのコピーと子のランタイムが使う資源は家族単位の割り当て量に課金され，子孫も同じ割り当て量を共有する．

4.3 節で述べたプロセスの間の協調のため，継承したファイル記述の読み取り位置は親子で共有する dataspace に置く．子の初期化の完了と終了は，親への通知用の IPC ゲートで受け取る．このゲートは親 task のスレッドに結び付け，子には呼び出しの権利だけを渡す．

ⓒ2026 Information Processing Society of Japan

5.6 実装上の制約

4.3 節で述べた未対応の意味論に加えて，プロトタイプには次の制約がある．配置の番地は空きから決定的に選ぶため，ASLR はない．JIT のように，書き込みと実行を行き来するコードには対応していない．入口で保存する浮動小数点の状態は `FXSAVE64` の範囲に限られ，AVX などの拡張状態は保存しない．

6. 書き換え方式の互換性

L4Re 上のプロトタイプで動作するプログラムの範囲は，捕捉の方式だけでなく，shim がどこまで Linux の意味論を実装したかによって決まる．そのため，L4Re 上の結果だけでは，動作しないプログラムの原因が捕捉の方式にあるのか，意味論の不足にあるのかを区別できない．本章では，捕捉したシステムコールをそのまま Linux カーネルへ転送する Linux 上のプロトタイプを用いて，意味論の不足を除いた，書き換え方式そのものの互換性を調べる．

6.1 Linux 上のプロトタイプ

Linux 上のプロトタイプは，`LD_PRELOAD` で読み込まれる共有ライブラリとして実装した．起動時のコンストラクターで，読み込み済みの main，共有ライブラリ，vDSO の実行可能な領域を走査して `syscall` 命令を書き換え，0 番地にトランポリンを置く．shim は捕捉したシステムコールを Linux カーネルへ転送し，`execve` では `LD_PRELOAD` を環境変数に入れ直して，実行後のイメージにも shim を読み込ませる．

shim の初期化を終えた後は，Linux の Syscall User Dispatch を有効にする．許可する範囲は，shim がネイティブのシステムコールを発行するアセンブリのコードをまとめた区間だけであり，それ以外から実行された `syscall` 命令はすべて SIGSYS となる．SIGSYS のハンドラーは，システムコール番号と命令の番地を記録してプロセスを終了させる．これにより，書き換えの漏れは黙ってカーネルで実行されることなく，拒否として観測される．Syscall User Dispatch の設定は `fork` や `clone` の子に継承されないため，shim は子が対象のコードを再開する前に設定し直す．ただし，shim の初期化より前に実行されるコードは，この検査の範囲外である．また，`LD_PRELOAD` を用いるため，静的リンクされたバイナリは対象外である．これは Linux 上での注入の手段による制約であり，ELF をランタイム自身が読み込む L4Re 上のプロトタイプには当てはまらない．

6.2 調査の方法

対象は，Debian trixie（amd64）のパッケージのうち，popcon の順位の上位から，コマンドラインのプログラムを含むものを順に選んだ 100 件である．成功の見込みによる選別はしていない．各パッケージについて，実行前に代表的な操作を 1 つ定め，同じ準備済みの VM のスナップショットから，shim なしの実行（以下，native）と shim ありの実行を行い，終了コード，出力，生成物を比較した．shim ありの実行では，システムコールごとに捕捉のログを出力した．VM は KVM 上の QEMU（2 vCPU，3 GiB，ネットワークなし）であり，0 番地のマップのための `vm.mmap_min_addr` の変更を行った．対象のパッケージと版，代表的な操作，期待した結果，判定の一覧は，再現の手順とともに 5.1 節で示したリポジトリの `docs/debian-cli.tsv` に置いた．VM の元には，Debian が配布する trixie の genericcloud イメージ（20261021-2638）を用いた．

これとは別に，Go と Node.js で高負荷の試験を行った．Go はランタイム自身が libc を経由せずにシステムコール命令を発行する例であり，Node.js はシステムコールを libc を通じて発行するが，JIT コンパイラーと多数のスレッドを持つ例である．試験はいずれも，次の負荷を同じプロセスの中で 16 巡繰り返すものであり，数値は 16 巡の合計である．Go では，Go 1.22.2 で作成し動的リンカから起動する ELF を用いて，128 本の OS スレッドに固定した goroutine からの `gettid` の直接の呼び出し約 819 万回，4,096 個の goroutine によるスケジューリング，累積 25 GiB 以上のヒープの確保とガベージコレクション，4 GiB のファイルの書き込みと読み戻しの検算，HTTP の 32 万要求，子プロセス 1,024 個の生成と回収を行った．Node.js では，配布されている v24.18.0 のバイナリを用いて，V8 によって最適化される関数の 128 万回の呼び出し，累積 32 GiB の `TypedArray` の確保とガベージコレクション，Worker 2,048 個の生成と終了，Go と同じファイルと HTTP の負荷，子プロセス 1,024 個の生成と回収を行った．

6.3 結果

100 件のうち 90 件で，native と動作が一致し，shim による捕捉も確認できた．一致しなかったものは 5 件，shim を注入できなかったものは 4 件，VM の環境が前提を満たさず比較できなかったものは 1 件であった．Syscall User Dispatch による，捕捉されなかった `syscall` 命令の拒否は，100 件のいずれでも観測されなかった．ただし，これは各パッケージの 1 つの代表操作についての結果であり，パッケージのすべての機能が動作することを示すものではない．

Go と Node.js の試験は，いずれも最後まで完了し，検算に成功して，出力は native と一致した．また，QEMU 自身に shim を注入したところ，システムモードでは TCG によって Linux を起動し，その中の試験を完了した．一方，ユーザーモードでは，6.4 節で述べるシグナルの処理で不一致が生じた．

ⓒ2026 Information Processing Society of Japan

6.4 非互換の分類

一致しなかったもの，注入できなかったもの，調査の過程で見つかった失敗を，原因によって次の 4 つに分類する．後の 2 つは，native，書き換えなしで走査だけを行う実行，ログなしの実行，ログありの実行を比較する対照実験によって切り分けた．

6.4.1 書き換えの副作用

書き換え方式そのものに起因する失敗である．`call *%rax` は戻り番地をスタックに積むため，対象の red zone の上端 8 バイトを壊す．red zone を使ってシステムコールの前後に値を保持する Rust の rustix では，この破壊によって動作が変わった．また，Go の試験では，shim の `clone` の処理が対象のスタックの上で動作したため，goroutine の小さなスタックの下限を越えて書き込み，子プロセスの生成の後に失敗した．後者は，`clone` の処理を専用のスタックで行うように修正し，解消した．さらに，書き換えによってシステムコールが実際に実行される位置が shim の中へ移るため，割り込まれたときのプログラムカウンターに依存する処理が成り立たなくなる．QEMU のユーザーモードは，ブロックするシステムコールとシグナルの競合を避けるため，シグナルが割り込んだ位置がシステムコール命令の前後の特定の範囲にあるかを検査する．書き換えた後は，割り込まれる位置が shim の中になるため，この検査が成り立たず，`SA_RESTART` を指定したシグナルの処理で，停止や順序の入れ替わりが生じた．この問題は `jmp` による方式でも同様に生じ，red zone の破壊とは異なり，書き換えて shim で実行するという方式に共通する．また，0 番地にトランポリンを置くこと自体が，NULL 付近の読み出しはページフォルトになるという前提を壊す．OpenJDK の HotSpot は，参照が NULL であれば読み出しでページフォルトが生じることを利用し，これを捕らえて `NullPointerException` を送出する．トランポリンが読み出せると，この読み出しはトランポリンの内容を値として返し，例外は送出されない．Temurin 21.0.12 で NULL の配列の長さとフィールドを読む試験では，インタープリターと C2 でコンパイルしたコードのいずれでも例外が送出されず，`nop` の並びである `0x90909090` が値として返った．`jmp` による方式でも，書き換えを行わずに 0 番地へ同じページを置くだけでも同じ結果となったことから，この失敗は 0 番地の使用に起因する．これに対しては，Zpoline の論文 [8] が述べるとおり，Intel MPK（PKU）を用いてトランポリンを実行専用にすれば，読み出しはページフォルトになる．PKU を有効にした QEMU の TCG 上で，PKU に対応した Ubuntu のカーネル（Linux 6.8.0）を用いて試したところ，上記のすべての場合で `NullPointerException` が送出された．

6.4.2 捕捉の漏れ

Linux 上のプロトタイプは起動時に 1 度だけ走査するため，その後に読み込まれる共有ライブラリや，実行時に生成されるコードは書き換えの対象から漏れる．当初は vDSO も走査から除外していたため，vDSO の中の `getrandom` の代替経路が漏れ，17 件で Syscall User Dispatch による拒否が観測された．vDSO を走査の対象に加えた後，100 件の結果は上記のとおり拒否なしとなった．

6.4.3 Linux での注入の手段による制約

setuid や setgid のプログラムでは，`LD_PRELOAD` が無視されるため，shim を注入できない（cron，sudo，fuse3）．AppArmor のプロファイルが shim のライブラリの読み込みを拒否した例もあった（cups-daemon）．man-db では，アプリケーション自身の seccomp のフィルターが，shim が `execve` の前に環境変数を複製するために用いる `process_vm_readv` を拒否し，子プロセスを起動できなかった．initramfs-tools では，`execve` で環境変数に入れ直した `LD_PRELOAD` によって，生成したイメージに shim のライブラリが含まれた．いずれも，Linux に shim を `LD_PRELOAD` で注入し，Linux の機能で動作を補っていることによる．

6.4.4 観測による変化

比較のために出力したログが動作を変えた例である．sshd では，ファイルの大きさの上限を設定した子プロセスが，ログの書き込みで SIGXFSZ を受けて終了した．anacron では，shim の初期化のメッセージがジョブの出力とみなされ，メールを送る経路へ分岐した．sshd はシステムコールごとのログを出力しない実行では一致し，anacron は書き換えを行わず走査だけを行う実行でも同じ分岐が生じたことから，いずれも捕捉ではなく出力に起因する．なお，x11-xkb-utils で発生した不一致は，ライブラリが初期化せずに書き出す 96 バイトの領域の差であり，読み直した内容は一致した．

6.5 red zone の破壊の緩和

red zone の破壊は `call` による書き換えに固有の問題であるため，`syscall` 命令とその後続の命令を合わせて 5 バイト以上を `jmp` に置き換え，近くに置いたスタブへ移す方式を試作した．スタブからの入口は，最初の書き込みより前にスタックポインターを red zone の外まで下げるため，red zone を壊さない．移設する後続の命令は，レジスタと即値だけを使う一部の命令に限り，条件を満たさない箇所は従来の `call` のまま残す．rustix の 3 つの版と 2 つの Rust のコンパイラーで作った 6 つの構成では，従来の方式で再現した red zone の破壊が，この方式ではいずれも生じなかった．これは限られた命令列に対する試作であり，一般的な命令の再配置を実現したものではない．

6.6 L4Re への示唆

4 つの分類のうち，Linux での注入の手段による制約は，ランタイムが ELF を読み込み，ホストのカーネルの機能

を用いない L4Re 上のプロトタイプでは生じない．観測による変化も，比較のためのログを出力しなければ生じない．書き換えの副作用のうち，shim が対象のスタックを使う問題は，入口で直ちに shim 用のスタックに切り替える L4Re 上の実装（5.3 節）では生じないが，red zone の破壊は同様に残る．L4Re 上の実装も 0 番地のトランポリンを読み出し可能な権限でマップしているため，NULL 参照の検出の失敗も同様に残る．捕捉の漏れについては，L4Re 上の実装は `mmap` や `mprotect` で実行可能になる領域も公開の前に走査するため，後から読み込まれる共有ライブラリは対象に含まれる．一方，書き込みと実行を行き来する生成コードには対応しない．

以上から，書き換え方式は一般的なプログラムの実行を大きく妨げないといえる．また，書き換えの副作用や捕捉の漏れによって対象が誤動作した場合も，L4Re 上ではその影響は task に与えた capability の範囲を超えない（3.5 節）．

7. 評価

本章では，Fiasco.OC/L4Re 上のプロトタイプについて，次の 3 点を評価する．第 1 に，Linux としての権限が，shim ではなく task に与えた capability によって決まるか．第 2 に，L4Re 上でどのようなプログラムが動作するか．第 3 に，システムコールの処理にどれだけのコストがかかるか．

7.1 評価の環境

評価には，L4Re のスナップショット 26.09.0 に含まれる Fiasco.OC と L4Re を用いた．ホストの CPU は Core Ultra 9 285K である．機能の試験は，WSL2 上の QEMU 8.2.2 による 1 vCPU，512 MiB の VM で行い，KVM と TCG の両方で同じ結果を得た．性能の測定は，Windows 上で直接実行した Windows 版の QEMU（v10.2.0-12105-g0f12d445bd）と Windows Hypervisor Platform（WHPX）による 2 vCPU，512 MiB の VM で行い，QEMU をホストの 2 つの論理 CPU に，測定対象をゲストの CPU 0 に固定した．比較に用いた Linux（Alpine Linux のカーネル 6.18.58-0-virt）も，同じ構成の VM 上で実行した．

7.2 task に与えた権限

表 1 に，プロトタイプで各 task に与えた capability を，Fiasco.OC/L4Re の capability を単位として示す．権利の R，W，S は，Fiasco.OC の capability の権利のビットである．R は capability を持つために常に必要な権利であり，W と S の意味はオブジェクトのインターフェースごとに定まる．たとえば dataspace では，書き込み可能な写像を得るために W が必要になる．上段は，ランタイムの起動と実行に必要な capability であり，すべての task に与える．下段は，Linux の資源に対応する capability であり，試験の内容に応じて与えるかどうかを変える．ただし，SQLite の試験では `mem_alloc` の割り当て量を 4 MiB とし，`linux_process` を与える試験では `factory` の割り当て量を 512 KiB とした．

`mem_alloc` からは，ランタイムのほか，静的な非 PIE の ELF のセグメント，初期スタック，0 番地のトランポリンを確保する．動的 ELF と静的 PIE のセグメント，および `mmap` の匿名メモリは `linux_memory` から確保する．起動用の `rom` は，ランタイムと対象の ELF の 2 つだけを公開する名前空間であり，ランタイムが対象を読み込んだ後，対象に制御を移す前に破棄する．`linux_process` と `zcomp_binary` は，fork を行う試験でだけ与える．fork で作る子 task には，子自身の task，スレッド，region manager，親への通知用の IPC ゲート（R），親と共有するカーネルの factory，スケジューラー，ログに加えて，親が持つ `linux_*`の capability を，割り当て元は RS，それ以外は R で委譲する．子 task のメモリの割り当て元は `linux_process` であり，起動用の `mem_alloc` は渡さない．

これらの制限は，Moe, Ned, カーネルを変更せず，Ned の設定だけで行った．ただし，上段の capability は侵害された shim からも使える．たとえば `log` への出力や，`mem_alloc` の割り当て量の範囲でのメモリの確保は，Linux の資源に対応する capability を与えなくても可能であり，`parent` の権利も RWS のままである．また，Moe の factory は dataspace だけでなくログなどのオブジェクトも生成できる．これを生成できないようにするには Moe の変更が必要になるため，本プロトタイプでは行っていない．したがって，3.5 節で述べた権限の上限は，表 1 の上段と，与えた下段を合わせたものである．

7.3 権限の強制

同じ ELF を，与える capability だけを変えて実行し，結果が capability によって決まることを確認した．結果を表 2 に示す．

shim を迂回した結果は，shim を通らずに L4Re の IPC を直接発行する対象を用いて得た．この対象は，書き換えの対象にならないよう，`mov` 命令の即値の中に `syscall` 命令を含め，そこへ分岐して L4Re の ABI で名前空間を照会し，得た dataspace をマップする．この命令は，対象が動的リンクする共有ライブラリの中に置いた．公開されたファイルは内容まで一致し，公開されていないファイルは名前空間が `L4_ENOENT` を返して取得できなかった．起動用の `rom` を経由した取得もできなかった．また，迂回した操作の前後で shim の呼び出しの回数が変わらないことから，これらの操作が実際に shim を経由していないことを確認した．対照として，同じ方法で数えたとき，直前に発行した `gettid` と `getpid` では回数が 2 増えることも確かめた．

すなわち，ファイルを公開するかどうかは，shim の判断

ⓒ2026 Information Processing Society of Japan

表 1 task に与えた capability

名前	種類	権利	用途と制限
`task`	Task	RWS	自 task の capability の表とメモリのマップの操作
`main_thread`	Thread	RWS	自スレッドの制御
`rm`	Region manager	RWS	自アドレス空間の予約，配置，解除
`itas`	IPC ゲート	RWS	ネイティブなスレッドと UTCB の管理の補助
`parent`	IPC ゲート	RWS	Ned への終了の通知など
`mem_alloc`	Moe の factory	RS	ランタイムと非 PIE の静的 ELF の配置．割り当て量 1 MiB
`factory`	カーネルの factory	RWS	task，スレッド，IPC ゲートなどの生成．割り当て量 256 KiB
`scheduler`	スケジューラーの proxy	RS	CPU 0，優先度 2 以下に制限
`log`	Log	R	ランタイムの診断の出力
`rom`	Namespace	R	ランタイムと対象の ELF だけを公開．対象の実行前に破棄
`linux_stdout`	Log	R	標準出力と標準エラー出力
`linux_files`	Namespace	R	公開するファイルと共有ライブラリ
`linux_memory`	Moe の factory	RS	PIE と動的 ELF の配置と `mmap`．割り当て量 64 KiB〜8 MiB
`linux_process`	Moe の factory	RS	fork のメモリの複製と子のランタイム．割り当て量 64 KiB〜4 MiB
`zcomp_binary`	Dataspace	R	fork で子にロードするランタイムの ELF

表 2 与えた capability と結果

資源	与えた capability	shim を経由した結果	shim を迂回した結果
標準出力	`linux_stdout` あり / なし	出力される / `EACCES`	–
ファイル	`secret.txt` を公開	内容が一致	内容が一致
	`secret.txt` を非公開	`ENOENT`	`L4_ENOENT`
	`linux_files` なし	`EACCES`	–
	起動用の `rom` の `secret.txt`	–	取得できない
匿名メモリ	`linux_memory` 64 KiB で 128 KiB を要求	`ENOMEM`	–
	`linux_memory` なし	`EACCES`	–
子プロセス	`linux_process` なし	`EACCES`	–
	`linux_process` 64 KiB	`ENOMEM`	–

ではなく，task に与えた名前空間の内容によって決まっている．shim の検査をすべて迂回した場合も，得られる権限は与えた capability と同じであった．ただし，これは特定の攻撃経路についての確認であり，すべての経路について権限が漏れないことを証明するものではない．

7.4 互換性

L4Re 上の回帰試験は 12 ケースからなる．3 ケースは機能の試験であり，静的および動的にリンクした musl の ELF，TLS，fork，`CLONE_VM` によるスレッドと pthread，子プロセスの終了と回収，`mmap` の区間の管理，ファイルの操作を含む．2 ケースは 7.3 節の shim の迂回の試験である．残りの 7 ケースは表 2 の権限の試験であり，同じ ELF に与える capability と割り当て量だけを変えて，成功する場合と拒否される場合を確かめる．拒否される場合と直接の IPC を除き，同じ ELF を Linux 上でも実行し，結果を比較した．

加えて，外部から入手したバイナリを無編集で実行した．Alpine Linux で配布されている BusyBox 1.37.0（動的 PIE）では，`echo`，`cat`，`sha256sum`，`wc` の 7 ケースが，Linux 上の同じバイナリと同じ結果になった．SQLite 3.53.4 では，メモリ上のデータベースへの表の作成，更新，トランザクション，集計と，公開したデータベースファイルの読み取りが，Linux 上と同じ結果になった．いずれも，ELF の再ビルド，再リンク，事前の書き換えは行わず，動的リンカと musl libc も配布されたものをそのまま用いた．BusyBox と SQLite は `brk` などの未対応のシステムコールに `ENOSYS` を受け取るが，自身で `mmap` などの代替の経路へ移って動作を続けた．

一方，`execve`，コピーオンライトによる `fork`，`MAP_SHARED`，非同期なシグナル，ファイルの作成と書き込みは未対応であり（4.3 節），対話的なシェルなどはまだ動作しない．これらは，6 章で調べた書き換え方式とは別に，shim の実装として拡充が必要な部分である．

7.5 性能

L4Re と Linux ではシステムコールの背後の実装が大きく異なるため，個々のシステムコールの時間を両者で比べ

ⓒ2026 Information Processing Society of Japan

表 3 L4Re 上での処理時間 (中央値)

操作	経路	時間
`getpid`	キャッシュ経路	18.2 ns
`gettid`	通常（GDT で切替）	74.5 ns
`gettid`	通常（API で切替）	842.7 ns
4 KiB の確保と解放	Moe，カーネル	27.32 μs

ても，捕捉の方式の評価にはならない．そこで，L4Re 上での処理を，カーネルを経由しない経路と経由する経路に分けて，その時間を表 3 に示す．測定には，`syscall` 命令を直接実行する静的 ELF を用いた．各値は，VM を 3 回起動してそれぞれ 7 標本を取った 21 標本の中央値である．1 標本は，`getpid` と `gettid` では 100 万回（キャッシュ経路は 2 万回），メモリの確保と解放では 250 回の繰り返しであり，事前に 1 割の回数を実行してから測定した．時間は TSC の差分から換算し，ループと戻り値の検査を含む．

カーネルを経由しない通常の経路では，状態の保存，FS の切り替え，ディスパッチを含めて，1 回のシステムコールを約 75 ns で処理した．`getpid` のキャッシュ経路は，状態の保存も行わないため約 18 ns である．これに対して，FS の切り替えをカーネルの API で行い，1 回のシステムコールごとにカーネル遷移が 2 回生じる実装では，同じ処理に約 843 ns を要し，約 11 倍の時間がかかった．0 バイトの `write`，8 バイトの `pread`，`fstat` でも，同様に 9〜11 倍の差があった．システムコールの捕捉と変換の経路からカーネル遷移を除くことが，処理時間に直接効くことを示している（3.2 節）．一方，4 KiB の `mmap` と `munmap` の繰り返しには，1 回あたり約 27 μs を要した．この測定では 5.4 節のプールからページが再利用されるため，dataspace の確保と返却は生じない．それでも，ゼロで埋めるための一時的な配置と解除，対象の番地への配置，`munmap` での解除のたびに，region manager への登録と解除，Moe へのマップの要求，カーネルによるマップと解放が生じる．このコストは，shim ではなく，背後のサーバーとカーネルの実装で決まる．

表 4 に，配布されている BusyBox と SQLite での実行時間を示す．

Linux の列は，`exec` の完了から親が終了を回収するまでの時間である．L4Re の列は，対象へ制御を移してから対象が終了して戻るまでの時間であり，動的リンカの実行を含む．破棄を含む列は，これにランタイムによる資源の解放の時間を加えたものである．比は，それぞれ直前の列の値の Linux に対する比である．各値は，VM を 3 回起動してそれぞれ 3 回実行した 9 標本の中央値である．

計算が中心の負荷は，Linux とほぼ同じ時間で完了した．対象の実行だけを見れば，多くの負荷は Linux の 0.9〜1.3 倍程度であるが，メモリの確保が多い SQLite のメモリ上の DB では約 7 倍を要した．また，終了時にランタイムを破棄するまでを含めると，各負荷に 30〜60 ms 程度が加わる．これは主に，task が確保したメモリの解放で Fiasco.OC の RCU の猶予期間を待つためである．カーネルを変更すればこの待機を短くできることを確認したが，本論文ではカーネルを変更しない方針とし，改善は今後の課題とする．

以上のように，捕捉と通常の経路のコストは小さく，現在の性能の差は主にメモリの管理を担うサーバーとカーネルの実装に由来する．なお，L4Re 上の shim は Linux の意味論を完全には実装していないため，これらの値は同じ処理を行う Linux との厳密な比較ではない．

7.6 別の OS での適用例

筆者が開発している別のマイクロカーネル OS である pachaos にも，同じ書き換え方式による互換層を実装している．pachaos でも Zpoline による書き換えを用い，shim を対象と同じ保護単位に置く．一方，pachaos では，fork やメモリの管理など Linux の意味論の一部をカーネルの側でも実装し，プロセスの間の状態を協調用のサーバーで管理している．また，capability による権限の分離は L4Re ほど整っていない．そのため，本節の例は権限の強制の評価ではなく，この書き換え方式のもとで実際のアプリケーションが動作し得ることを示すものである．図 2 に，pachaos 上の xfce4 の端末で，Alpine Linux が配布する GCC 14.2.0 と Clang 20.1.8 を用いて zstd 1.5.7 をビルドした様子を示す．いずれのコンパイラーでもビルドは完了し，生成した `zstd` が動作することを確かめた．

8. 関連研究

8.1 マイクロカーネル上の Linux 互換

L4Linux[1] は，Linux カーネル全体を L4 上のユーザー空間のサーバーとして動作させる．現在の L4Linux は Fiasco.OC の vCPU[9] を用い，Linux プロセスを別の task として vCPU のユーザーモードで実行して，そのシステムコールをカーネルを経由した例外として受け取る．互換性は高いが，1 章で述べたとおり，L4Linux のサーバーは全クライアントが必要とする権限の和集合を持つ．Starnix[5] は，Zircon のプロセスのアドレス空間を互換層と Linux プログラムで分け，Linux プログラムを restricted mode で実行する．Linux プログラムが `syscall` 命令を実行するとカーネルが通常のモードへ戻し，同じプロセスの互換層がシステムコールを処理する．互換層と対象を同じプロセスに置く点は本手法と同じであるが，捕捉はカーネルの機構に依存する．Genode や Redox OS のように，POSIX の互換性をその OS 向けの libc として提供する方式もある [10], [11]．この方式は capability との対応を取りやすいが，対象をその libc で再ビルドまたは再リンクする必要があり，無編集のバイナリは実行できない．

ⓒ2026 Information Processing Society of Japan

表 4　アプリケーションの実行時間 (ms)

負荷	Linux	L4Re	比	破棄を含む	比
BusyBox `awk`（500 万回の計算）	1359.3	1258.5	0.93	1291.7	0.95
BusyBox `sha256sum`（8 MiB）	22.1	25.2	1.14	58.2	2.63
BusyBox ファイルの開閉と読み取り（1 万回）	14.8	18.2	1.23	51.0	3.44
SQLite 再帰 CTE（100 万件）	71.9	73.5	1.02	130.6	1.82
SQLite メモリ上の DB（5 万行）	31.8	220.1	6.92	276.7	8.71
SQLite 読み取りと集計	33.2	38.0	1.14	95.1	2.87

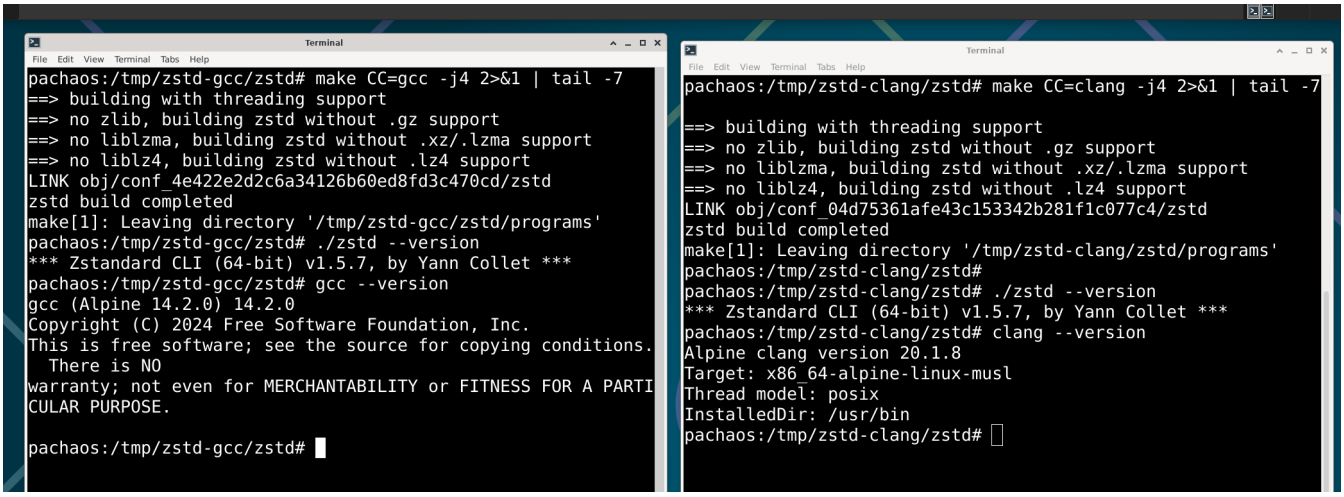

図 2　pachaos 上の xfce4 での zstd のビルド (左は GCC，右は Clang)

8.2　ライブラリ OS

Drawbridge[2] は，Windows をライブラリ OS として再構成し，狭い ABI だけを持つ picoprocess の中でアプリケーションとともに実行する．picoprocess はホストのカーネルの変更によって閉じ込められ，ファイルを開くなどの判断は別プロセスのセキュリティモニターが行う．Bascule[3] は Drawbridge を基にした拡張の枠組みであり，本手法に最も近い構成を含む．Bascule の Linux のライブラリ OS は，libc，動的リンカ，libpthread を改変し，それらの中のシステムコール命令をエミュレーション層の呼び出しに置き換える．そのため，対象は互換の libc を動的リンクするアプリケーションに限られ，著者らはこの改変を実行時の書き換えで行えると述べるにとどまる．また，ホストの 1 つである Barrelfish では，ABI の実装をすべてプロセス内に置き，隔離は Barrelfish 自身の閉じ込めの機構に任せている．本手法は，この書き換えをロード時に任意のバイナリに対して行うことで，libc を経由しないシステムコールも含めて無編集で捕捉する．加えて，Linux の資源ごとに capability を対応させ，fork で生じた task にも必要な capability だけを委譲し，shim を迂回した場合にも権限が capability で決まることを確認した．Graphene[4] は，Linux アプリケーションのための同様のライブラリ OS である．改変した glibc を経由するシステムコールはカーネルを経由せずにライブラリ OS で処理されるが，glibc を経由せずに直接発行されたシステムコールは，ホストの Linux カーネルに seccomp でトラップさせてからライブラリ OS へ転送する．本手法は，libc を経由するかどうかにかかわらず命令そのものを書き換えるため，どちらの経路でもカーネルを経由しない．また，Graphene のホストはモノリシックなカーネルであり，隔離は picoprocess 単位の seccomp のフィルターと reference monitor によって行われ，複数プロセスにまたがる状態はライブラリ OS のインスタンス同士が協調して管理する．本手法は capability ベースのマイクロカーネルのマルチサーバー構成を前提とし，権限をオブジェクト単位の capability として与える．

8.3　システムコールの捕捉

ptrace，seccomp と SIGSYS，Linux の Syscall User Dispatch[12] は，いずれもカーネルを経由してシステムコールを捕捉する．gVisor[13] も，ptrace や KVM などのカーネルの機構を用いて捕捉したシステムコールを，ユーザー空間のカーネルで処理する．書き換えによる方式では，Zpoline[8] が 2 バイトの `syscall` 命令を同じ長さの `call *%rax` に置き換えることで，命令の境界を変えずに網羅的な書き換えを実現した．Zpoline の論文は制約として，初期化の後に読み込まれるコード，vDSO，0 番地を使えない環境を挙げているが，6 章で述べた red zone の破壊や，システムコールの実行位置に依存する処理は挙げていない．lazypoline[14] は，Syscall User Dispatch で捕捉した `syscall` 命令をその場で書き換えることで，実行時に現れるコードにも対応

ⓒ2026 Information Processing Society of Japan

する．syscall_intercept[15] や E9Patch[16] は，`syscall` 命令とその前後の命令を `jmp` に置き換え，移設した命令をトランポリンで実行する．6.5 節の試作は，この種の方式によって red zone の破壊を避けられることを確かめたものである．

8.4 capability とプロセス内の隔離

Capsicum[17] や WASI[18] は，POSIX の資源へのアクセスを capability として扱うが，アプリケーションの変更や専用の環境への再ビルドを必要とする．本手法は，無編集の Linux バイナリに対して，外側から capability による権限を強制する．また，NaCl[19]，ERIM[20]，Hodor[21] は，同じアドレス空間の中に信頼するモニターやライブラリを置き，ソフトウェアによる隔離や Memory Protection Keys でそれを対象から守る．Zpoline の論文も，フック関数を MPK で隔離できると述べている．本手法はこれと異なり，shim を対象から守らず，shim が侵害されることを前提として，権限の強制を task の外にあるカーネルとサーバーに置く（3.5 節）．

9. おわりに

本論文では，capability ベースのマイクロカーネル上の Linux 互換レイヤーについて，無編集の Linux バイナリの実行，カーネルを経由しないシステムコールの捕捉，capability による権限の強制の 3 つを両立させる構成を示した．本手法は，Zpoline によってシステムコール命令を書き換え，互換層である shim を対象と同じ非信頼の task に置き，権限の強制を task の外にあるカーネルとサーバーに任せる．Fiasco.OC/L4Re 上のプロトタイプは，Moe，Ned，カーネルを変更せずに実装した．

評価では，与える capability を変えると結果がそれに従って変わり，shim を迂回して直接 IPC を発行した場合にも同じ上限が保たれることを確認した．Linux 上のプロトタイプを用いた調査では，Debian の上位 100 パッケージのうち 90 件で動作が一致し，非互換の多くは Linux での注入の手段と観測に由来した．書き換え方式に固有の非互換としては，red zone の破壊，システムコールの実行位置への依存，0 番地の使用による NULL 参照の検出の失敗が見つかった．L4Re 上では，配布されている BusyBox と SQLite を無編集で実行でき，カーネルを経由しない通常の経路は 1 回のシステムコールを約 75 ns で処理した．これらの結果から，L4Re 上で動作するプログラムの範囲を主に決めているのは shim が実装した意味論の範囲であるといえる．ただし，書き換え方式に固有の非互換も L4Re 上に残る．

今後の課題は次のとおりである．第 1 に，`execve`，コピーオンライトによる `fork`，`MAP_SHARED`，非同期なシグナル，ファイルの作成と書き込みなどの意味論の拡充である．pachaos（7.6 節）では意味論を拡充して xfce4 などを動作させたが，意味論の一部をカーネルの側でも実装し，協調用のサーバーを用いており，4 章の設計のように権限を capability で強制したまま同じ範囲を実現できるかは明らかでない．第 2 に，書き換え方式に固有の問題への対処である．red zone の破壊は 6.5 節の方式の一般化によって，実行位置への依存は，シグナルの文脈に見せるプログラムカウンターを元のシステムコール命令の位置に合わせることによって対処できる可能性がある．NULL 参照の検出の失敗は Linux 上では PKU によって解消できたが，L4Re 上で同様にトランポリンを実行専用にできるかは確認していない．第 3 に，書き換えの漏れの扱いである．L4Re 上では，書き換えから漏れた `syscall` 命令は Fiasco.OC のシステムコールとして実行され，その影響は capability の範囲を超えないものの，Linux としての動作は保証されない．ロード時の走査を，漏れを検出して実行を拒否する検査として発展させることが考えられる．第 4 に，Fiasco.OC でのメモリの解放に伴う RCU の待機の短縮と，Moe の変更を伴う起動用の capability の絞り込みである．

謝辞 本論文の作成に当たり，構成と執筆について御指導を賜った明治学院大学経済学部国際経営学科の大野弘明教授に深く感謝する．

参考文献

[1] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S. and Wolter, J.: The Performance of μ-Kernel-Based Systems, *Proc. 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, pp. 66–77 (1997), DOI: 10.1145/268998.266660.

[2] Porter, D. E., Boyd-Wickizer, S., Howell, J., Olinsky, R. and Hunt, G. C.: Rethinking the Library OS from the Top Down, *Proc. 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pp. 291–304 (2011), DOI: 10.1145/1950365.1950399.

[3] Baumann, A., Lee, D., Fonseca, P., Glendenning, L., Lorch, J. R., Bond, B., Olinsky, R. and Hunt, G. C.: Composing OS Extensions Safely and Efficiently with Bascule, *Proc. 8th ACM European Conference on Computer Systems (EuroSys '13)*, pp. 239–252 (2013), DOI: 10.1145/2465351.2465375.

[4] Tsai, C.-C., Arora, K. S., Bandi, N., Jain, B., Jannen, W., John, J., Kalodner, H. A., Kulkarni, V., Oliveira, D. and Porter, D. E.: Cooperation and Security Isolation of Library OSes for Multi-Process Applications, *Proc. 9th European Conference on Computer Systems (EuroSys '14)*, Article 9 (2014), DOI: 10.1145/2592798.2592812.

[5] The Fuchsia Authors: Starnix syscalls (online), 入手先 ⟨`https://fuchsia.dev/fuchsia-src/concepts/starnix/syscalls`⟩ (参照 2026-10-29).

[6] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H. and Winwood, S.: seL4: Formal Verification of an OS Kernel, *Proc. 22nd ACM Symposium on Operating Systems Principles (SOSP '09)*, pp. 207–220 (2009), DOI:

ⓒ2026 Information Processing Society of Japan

10.1145/1629575.1629596.

[7] Kernkonzept GmbH: The L4Re Operating System Framework (online), 入手先 ⟨https://l4re.org/⟩ (参照 2026-10-29).

[8] Yasukata, K., Tazaki, H., Aublin, P.-L. and Ishiguro, K.: zpoline: a system call hook mechanism based on binary rewriting, *Proc. 2023 USENIX Annual Technical Conference (USENIX ATC '23)*, pp. 293–300 (2023). 入手先 ⟨https://www.usenix.org/system/files/atc23-yasukata.pdf⟩ (参照 2026-10-29).

[9] Peter, M., Lackorzynski, A. and Warg, A.: Virtual Processors as Kernel Interface, *Proc. 12th Real-Time Linux Workshop (RTLWS '10)*, Nairobi, Kenya (2010). 入手先 ⟨https://www.osadl.org/fileadmin/dam/rtlws/12/Lackorzynski.pdf⟩ (参照 2026-10-29).

[10] Genode Labs: Genode Operating System Framework (online), 入手先 ⟨https://genode.org/⟩ (参照 2026-10-29).

[11] Redox OS Developers: Redox OS (online), 入手先 ⟨https://www.redox-os.org/⟩ (参照 2026-10-29).

[12] The Linux Kernel Documentation: Syscall User Dispatch (online), 入手先 ⟨https://docs.kernel.org/admin-guide/syscall-user-dispatch.html⟩ (参照 2026-10-29).

[13] The gVisor Authors: gVisor (online), 入手先 ⟨https://gvisor.dev/⟩ (参照 2026-10-29).

[14] Jacobs, A., Gulmez, M., Andries, A., Volckaert, S. and Voulimeneas, A.: System Call Interposition Without Compromise, *Proc. 54th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '24)*, pp. 183–194 (2024), DOI: 10.1109/DSN58291.2024.00030.

[15] pmem: syscall_intercept (online), 入手先 ⟨https://github.com/pmem/syscall_intercept⟩ (参照 2026-10-29).

[16] Duck, G. J., Gao, X. and Roychoudhury, A.: Binary Rewriting without Control Flow Recovery, *Proc. 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '20)*, pp. 151–163 (2020), DOI: 10.1145/3385412.3385972.

[17] Watson, R. N. M., Anderson, J., Laurie, B. and Kennaway, K.: Capsicum: Practical Capabilities for UNIX, *Proc. 19th USENIX Security Symposium* (2010). 入手先 ⟨https://www.usenix.org/legacy/events/sec10/tech/full_papers/Watson.pdf⟩ (参照 2026-10-29).

[18] WebAssembly Community Group: WASI: WebAssembly System Interface (online), 入手先 ⟨https://wasi.dev/⟩ (参照 2026-10-29).

[19] Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., Okasaka, S., Narula, N. and Fullagar, N.: Native Client: A Sandbox for Portable, Untrusted x86 Native Code, *Proc. 30th IEEE Symposium on Security and Privacy (S&P '09)*, pp. 79–93 (2009), DOI: 10.1109/SP.2009.25.

[20] Vahldiek-Oberwagner, A., Elnikety, E., Duarte, N. O., Sammler, M., Druschel, P. and Garg, D.: ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK), *Proc. 28th USENIX Security Symposium*, pp. 1221–1238 (2019). 入手先 ⟨https://www.usenix.org/system/files/sec19-vahldiek-oberwagner_0.pdf⟩ (参照 2026-10-29).

[21] Hedayati, M., Gravani, S., Johnson, E., Criswell, J., Scott, M. L., Shen, K. and Marty, M.: Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries, *Proc. 2019 USENIX Annual Technical Conference (USENIX ATC '19)*, pp. 489–504 (2019). 入手先 ⟨https://www.usenix.org/system/files/atc19-hedayati-hodor.pdf⟩ (参照 2026-10-29).

ⓒ2026 Information Processing Society of Japan