インメモリデータベース向けのマルチバリアント監視機構

榎本 秀平[1]　山田 浩史[1]

概要：インメモリデータベースは Web サービスを構成する上で重要なコンポーネントであり，それらを対象とした攻撃はセキュリティ上の脅威として広く認識されている．既存のオペレーティングシステムやコンパイラには，それらのサービスに対する攻撃緩和を目的としたセキュリティ機構が提供されているが，特定の攻撃に対する防御を目的としており，網羅的な防御は難しい．この問題は Multi-variant Execution Environments (MVEE) を導入することで解決される．MVEE は複数のアプリケーションインスタンス (レプリカ) を同時に稼働させ，各レプリカの振る舞いを監視することで異常検出が可能なランタイム環境であり，セキュリティ機構を適用したアプリケーションをレプリカとするような MVEE によって多様な攻撃の検出が可能となる．しかしながら，MVEE ではレプリカ数分の物理メモリが消費されるため，大量のメモリを消費するデータベースサービスへの適用が困難となる．本研究では，インメモリデータベースに対して効率的に MVEE を適用し，セキュリティ強化と省メモリ化を同時に達成する手法を提案する．本提案機構は各レプリカへ入力として渡されるアイテムデータが等しい点に着目し，レプリカ間でアイテムデータの格納ページを共有することで省メモリ化を実現する．提案手法を Linux 4.4.185 において実装し，Memcached 1.5.22 を用いた物理メモリ消費量の測定実験を行った．結果，MVEE 未適用の単体実行と比較して約 39.23 % 程度の消費量増加で MVEE 実行が可能であることが確認された．

キーワード：インメモリデータベース，オペレーティングシステム，セキュリティ

1. はじめに

インメモリデータベース (インメモリ DB) は Web サービスを構成する上でなくてはならないコンポーネントのひとつである．インメモリ DB は管理データをメモリ上にキャッシュすることで，データアクセスの高スループットが達成され，Database Management System (DBMS) の MySQL [1] や Key-Value Store (KVS) の Memcached [2], Redis [3] は多くの Web サービスに取り入れられている．たとえば，Netflix では Memcached をベースとしたメモリキャッシュシステム [4] を開発しており，Facebook や Twitter においても Memcached を用いた大規模メモリキャッシュシステム [5,6] を運用している．

既存のデータベースの多くは C/C++ といった型安全でないプログラミング言語によって実装されているため，メモリに起因するバグや脆弱性を数多く含んでいる．たとえば，Redis と Memcached では 2020 年 2 月から 3 月にかけてそれぞれ脆弱性 [7,8] が報告されている．どちらの脆弱性についてもバッファオーバーフローが原因であり，攻撃者はネットワークサービス経由でアプリケーションをクラッシュさせることが可能となる．このような攻撃からアプリケーションを保護するため，既存のオペレーティングシステム (OS) やコンパイラは様々なセキュリティ機構を提供している．たとえば，Address Space Layout Randomization (ASLR) [9] や Stack Smashing Protector (SSP) [10,11] がある．これらの機構は攻撃者がターゲットとするメモリ領域を攻撃することを困難にするが，攻撃の完全な防御を目的としていないため，攻撃者による回避や無効化が可能である．また，より強固なセキュリティ機構として Google Sanitizers [12] がある．Google Sanitizers は Address Sanitizer (ASan) [13]，Memory Sanitizer (MSan) [14]，Undefined Behavior Sanitizer (UBSan) [15] など様々な種類があり，いずれもコンパイル時にアプリケーションを書き換え，実行時にアプリケーションのメモリ領域をトラッキングすることでメモリエラーを動的に検出する．しかしながら，各 Sanitizer はそれぞれ特定のメモリエラーの検出のみを目的として設計されているため，1 Sanitizer で防御可能な攻撃パターン数には限りがある．また，各 Sanitizer は 1 アプリケーションに 1 Sanitizer を適用する想定の下で設計されているため，複数 Sanitizer 適用によって網羅的に攻撃を防御することは不可能である．

[1] 東京農工大学
Tokyo University of Agriculture and Technology

以上の点を改善するため，Multi-variant Execution Environments (MVEE) [16–21] を用いることで既存のセキュリティ機構を活用しながらアプリケーションの耐攻撃性を高めることができる．MVEE は，同一のアプリケーションインスタンス (レプリカ) を複数生成した上で同時に稼働させ，各レプリカの振る舞いをモニタが監視し，レプリカ間で異なる振る舞いを観測すると異常が生じたとみなす．MVEE を用いて，複数のセキュリティ機構を適用したレプリカを稼働させることで，検出可能な攻撃の種類を増やすことが可能となる [16].

MVEE はレプリカ数分の物理メモリを消費するため，大量のメモリを消費するアプリケーションへの MVEE 適用は困難となる．たとえば，インメモリ DB の Amazon Dynamo DB Accelerator (DAX) [22] では，最大 488 GiB の物理メモリを消費する．このようなインメモリ DB に対する MVEE 適用はシステム全体の物理メモリ量を超過する可能性があり，レプリカの起動失敗や，OOM Killer [23] によるレプリカ稼働途中での強制終了の恐れがある．また，MVEE の適用が成功した場合であっても，物理メモリ量の制約によって稼働可能なレプリカ数には限りがある．したがって，多様なセキュリティ機構を複数のレプリカに適用し，網羅的に攻撃を防御することは困難となる．

本研究では，インメモリ DB 適用下の MVEE において物理メモリ消費量を削減する手法を提案し，提案手法に基づいた機構である TwinsDB を設計した．TwinsDB では，各レプリカへ入力として渡されるアイテムページが等しい点に着目し，レプリカ間でアイテムデータの格納ページを共有することで，全レプリカの合計物理メモリ消費量を抑制する．提案手法によって，多様なセキュリティ機構が適用された複数のレプリカを稼働させることが可能となり，インメモリ DB の耐攻撃性が強化される．

本研究の貢献を以下に示す．

- MVEE において，レプリカ間で同一ページを共有する機構 TwinsDB を提案した．提案機構は次の特徴を持つ．インメモリ DB のレプリカを複数稼働するような状況において物理メモリ消費量が抑制され，MVEE 従来のセキュリティ効果についても保持される．また，適用にあたってはアプリケーションソースコードの変更を必要としない (3 章).
- TwinsDB のソフトウェア設計を示した．ランタイムオーバヘッド削減のためにインメモリ DB のコマンドに基づいたシステムコール同期を行い，高速なページ共有のために選択的なページスキャンを行う (4 章).
- Linux Kernel 4.4.185 に対して実装を行い，パフォーマンスと物理メモリ消費量の測定実験を行った．実験の結果，Memcached 1.5.22 において，MVEE 未適用の単体実行と比較して 39.23 % 程度の消費量増加で MVEE 実行が確認であることが確認された (5，6 章).

2. 背景

2.1 インメモリ DB に対する攻撃

C/C++ をはじめとする型安全でないプログラミング言語実装によるアプリケーションはメモリに起因するバグを数多く含むことが知られている．これらのうち，悪用可能なバグ (脆弱性) を突いてアプリケーションが攻撃されることによって，サービスのダウンや機密データのリーク，悪意あるコードの実行などがなされる危険性がある．たとえば，代表的な攻撃としてバッファオーバーフロー [24] や Uninitialized Read がある．

バッファオーバーフロー攻撃はプログラムが入力データサイズを正しく制限しない脆弱性を持つ場合に発生する．攻撃者はプログラム側が想定する本来のデータサイズを超過したデータをメモリに挿入することが可能となるため，メモリ領域の改竄や破壊が可能となる．たとえば，Memcached の脆弱性 CVE-2020-10931 [7] はバッファオーバーフロー起因でアプリケーションをクラッシュさせることが可能である．CVE-2020-10931 修正前はクライアントから指定されたサイズ分のメモリコピーを行っているため，Memcached 側の想定を超える大きなサイズを指定することでメモリ領域が破壊され，クラッシュが起こる．

Uninitialized Read 攻撃はメモリ領域の動的確保時に領域内を初期化しない場合に発生する．未初期化の動的確保領域には，その領域を解放する前のデータが残されているため，攻撃者は解放前のデータをリークさせることが可能となる．また，解放後の領域を誤って参照するような脆弱性も同時に存在する場合には，任意のテキストコードを実行可能な Use-After-Free (UAF) [25] 攻撃に発展可能なケースがある．

2.2 Multi-variant Execution Environments

メモリに起因した攻撃からアプリケーションを保護するために OS が提供するセキュリティ機構として Address Space Layout Randomization (ASLR) [9] がある．ASLR はプロセス内の仮想アドレス配置をランダム化することで攻撃者がターゲットとするメモリ領域を推測することを困難にする．たとえば，ASLR によってプロセス起動毎にテキストコードの仮想アドレスが変更されるため，攻撃者がプログラムの制御を奪った場合であっても，ターゲットとする関数やライブラリなどのペイロードに制御を移すことは困難となる．コンパイラが提供するセキュリティ機構としては Stack Smashing Protector (SSP) [10,11] や Google Sanitizers [12] がある．SSP はコンパイル時に Cannary と呼ばれるランダムな値をスタックフレーム中に挿入することによってスタックバッファオーバーフロー攻撃をランタイムで検出する．スタックフレーム中のリターンアドレスを書き換える場合，Cannary の値についても書き換えるこ

とになるため，リターンアドレスへのジャンプ前に Cannary が改竄されていないか確認することで不正なアドレスへのジャンプを防御することが可能となる． Google Sanitizers はアプリケーション実行時にメモリ領域をトラッキングすることで，メモリに起因するエラーを動的に検出する機構であり，Address Sanitizer (ASan) [13]，Memory Sanitizer (MSan) [14]，Undefined Behavior Sanitizer (UBSan) [15] など様々な種類がある．各 Sanitizer は検出対象のメモリエラーの種類が異なり，ASan はバッファオーバーフローと UAF，MSan は Uninitialized Read，UBSan は Null ポインタ参照や ゼロ除算などの未定義動作を対象とする．

以上の既存セキュリティ機構には，いくつかの問題点や考慮すべき特性が存在する．まず，ASLR や SSP は攻撃の完全な防御でなく，緩和を目的とした機構であるため，攻撃者によって回避や無効化が可能である．たとえば，ASLR はターゲットの仮想アドレスをリークすることによってランダム化を無効化することが可能であり，SSP についても Cannary 値をリークすることによって，Canarry チェックを正常に通すような攻撃コードを組み立てることが可能である．次に，Sanitizer は 1 アプリケーションにつき 1 Sanitizer 適用を想定した設計であるため，複数 Sanitizer の併用が不可能である．たとえば，ASan では仮想アドレス空間のうち低位アドレスを Shadow Memory と呼ばれるトラッキング用メタデータの管理領域として使用するが，MSan ではアクセス不可領域とするため，同時に適用することができない [16]．前述のように各 Sanitizer が検出可能なメモリエラーの種類は異なるため，複数 Sanitizer を適用することによって防御可能な攻撃パターン数を増やすことは不可能である．

既存のセキュリティ機構における問題点を改善するため，Multi-variant Execution Environments (MVEE) [16–21] を用いて既存機構の特性を活かしながらアプリケーションの対攻撃性を強化する研究 [16,17,19] がなされている．MVEE とは，アプリケーションインスタンス (レプリカ) を複数生成し，各レプリカを同時に実行させながら，それらの振る舞いをモニタが監視するようなランタイムである．既存研究では，モニタはレプリカのシステムコール発行をトリガーとして，各レプリカのシステムコール番号や引数が一致しているか確認する．一致している場合は各レプリカの実行を継続させ，一致していない場合は異常状態とみなして各レプリカの終了や再起動を行う．また，全レプリカのうち，ある一つをリーダーレプリカ，残りをフォロワーレプリカとし，モニタはリーダーのシステムコール実行結果をフォロワーに共有するための同期処理を行う．

MVEE を用いた対攻撃耐久性の追加は，攻撃発生時に各レプリカの状態に差異が生じるような設計にすることで実現される．たとえば，各レプリカに ASLR と SSP を適用した MVEE において，攻撃者がセキュリティ機構の回避としてポインタや Cannary のリークを試みた場合にはレプリカ間のシステムコール出力内容に差異が発生するため，モニタはリアルタイムに攻撃を検出し，防御することができる．また，1 レプリカに 1 Sanitizer を適用することで，1 アプリケーションに複数 Sanitizer を適用している状態を実現することができる [16].

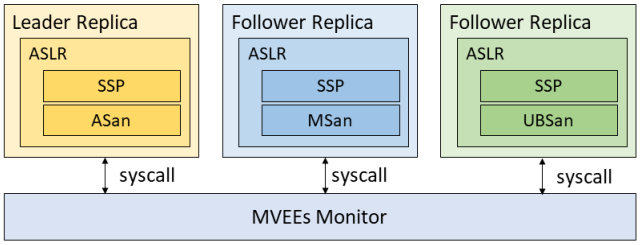

図 1 MVEE におけるレプリカとモニタの構成

2.3 関連研究

MVEE モニタは専用のアプリケーションとして動作する形式や，レプリカに組み込む形式，カーネルに組み込む形式など様々な種類が提案されている．これらの多くはセキュリティとパフォーマンスがトレードオフの関係にあり，MVEE 適用の目的によって最適な形式は異なる. アプリケーションのセキュリティ強化を目的とした MVEE [16,17,19] では，レプリカからモニタに対する攻撃を防ぐため，レプリカとモニタは分離された状態で動作する形式が取られる．また，ライブアップデートといったアプリケーションの信頼性や可用性の向上を目的とした MVEE [20,21] では，レプリカとモニタの通信を高速化するため，レプリカにモニタを組み込んだ状態で動作する形式が取られる．

Orchestra [19] はスタックバッファオーバーフロー攻撃の検出を目的とした MVEE である．モニタはユーザアプリケーションとして動作し，sys_ptrace システムコールを用いてレプリカのシステムコールを監視する．攻撃の防御にあたっては，レプリカ間でスタックの進行方向を逆にすることで，リターンアドレス改竄を検出する．

Remon [17] は既存のセキュリティ機構における防御効果の増幅とランタイムオーバヘッドの削減を目的とした MVEE である．Remon では各レプリカに ASLR，SSP を適用することで攻撃時にレプリカ間差異を発生させる．また，レプリカ内部で動作するモニタ (IP-Monitor) とユーザアプリケーションで動作するモニタ (CP-Monitor) を組み合わせており，攻撃に利用される可能のあるセンシティブなシステムコールの同期は CP-Monitor，センシティブでないシステムコールの同期は IP-Monitor で行うことで高速な MVEE ランタイムを実現する．

Bunshin [16] は異なるセキュリティ機構の同時適用によるセキュリティ強化を目的とした MVEE である．ASan,

MSan，UBSan のように 1 アプリケーションに対して複数適用が不可能なセキュリティ機構を各レプリカに一つずつ適用することで，1 アプリケーションに複数 Sanitizer を適用している状態を実現する．モニタはカーネル内のコンポーネントとして動作し，レプリカ間同期は 1 システムコールごとに同期を行う方式とセンシティブなシステムコールに絞って同期を行う方式の二種類を提供する．

KMVX [18] はカーネルにおけるデータリークの防御を目的とした MVEE である．カーネルに持つデータがユーザ空間にリークされることは深刻な攻撃に発展する可能性があり，たとえば Linux におけるカーネルポインタのリークは KASLR [26] の回避に繋がる．KMVX ではカーネルをレプリカとして，二つのカーネルを同時実行し，ユーザ空間へのデータコピー時に各レプリカのデータをチェックすることでリークを検出する．

VARAN [17] はアプリケーションの信頼性向上とランタイムオーバヘッドの削減を目的とした MVEE である．VARAN はセキュリティ強化ではなく，MVEE におけるパフォーマンス向上を重視しているため，モニタがレプリカに組み込まれた構成となっている．モニタとレプリカが同一の仮想アドレス空間に存在するため，Orchestra のようなモニタが独立した構成の MVEE とは異なり，システムコール同期にかかるコストが最小となる．

VARAN をベースにした MVEE として MVEDSUA [21] がある．MVEDSUA はライブアップデートを目的とした MVEE であり，アプリケーションが古いバージョンのレプリカと新しいバージョンのレプリカを稼働させることで，ダウンタイムなく透過的にアップデートを実現する．アプリケーションバージョンが違うことによって発生するシステムコールの差異については，ユーザが事前にルールを作成し，モニタはルールに基づいて差異を許容する [27].

2.4 問題点

MVEE は通常の実行システムと比較して，レプリカ数分の物理メモリを余分に消費する特徴があり，特に大量のメモリを使用するアプリケーションへの MVEE 適用は物理メモリ消費量の大規模増加が避けられない．たとえば，インメモリデータベースの Amazon Dynamo DB Accelerator (DAX) [22] では，最大 488 GiB の物理メモリを消費するため，このようなインメモリデータベースへの MVEE 適用はシステム全体の物理メモリ量超過が懸念され，レプリカの起動失敗や OOM Killer [23] によるレプリカの強制終了が考えられる．また，全体の物理メモリ量を超過しない場合であっても物理メモリの逼迫によってスラッシングが発生し，Web サービス全体としてのパフォーマンスに影響を及ぼす可能性がある．

3. 提案

本研究では，インメモリ DB を対象とした MVEE 機構である *TwinsDB* を提案する．既存研究における問題点を解決するために，TwinsDB は以下のデザインゴールを満たすように設計する．

- インメモリ DB をレプリカとする MVEE では，レプリカの合計物理メモリ消費量が増大するため，複数レプリカの適用が困難である．したがって，レプリカの合計メモリ消費量を削減することによって複数レプリカの稼働を実現する．
- 複数レプリカを稼働させることによって，多様なセキュリティ機構の適用を実現し，インメモリ DB のセキュリティを強化する．
- インメモリ DB のオペレータやプログラマによる管理，運用を容易にするため，アプリケーションソースコードの変更を必要とせず，透過的な MVEE 適用を可能とする．また，多くのインメモリ DB に対応するため，マルチプロセスやマルチスレッド形式のレプリカを実行可能とする．

合計物理メモリ消費量を削減するため，各レプリカに入力として渡されるデータアイテムが等しい点に着目する．TwinsDB は **図 2** に示すようにレプリカ間で同一内容ページの共有を行うことで消費量を抑制する．データアイテムは同一の内容でありながら，各レプリカで別々のページとして確保されるため，物理メモリには，同一内容ページがレプリカ分だけ存在していることとなる．また，Twins DB はインメモリ DB ソースコードを変更することなく，多様なセキュリティ機構の適用および実行を透過的に実現する．レプリカ間同期機構はカーネル内のコンポーネントとして動作し，特定のシステムコールを選択的に監視および同期を行う．

以上のアプローチに基づいた設計において要求されるデザインチャレンジを以下に示す．

- **アプリケーションの制御フローを変更するセキュリティ機構を適用したレプリカの実行を許容する**
  ASan，MSan，UBSan などの Sanitizer をはじめとするセキュリティ機構は適用によってアプリケーションのシステムコール発行順序が変更されるため，モニタはセキュリティ機構によるシステムコール差異を異常状態と見なす可能性がある．このような状況を避けるため，各機構による固有の振る舞いを許容しながら MVEE としての実行を継続する手法を設計する．
- **レプリカ間同期によるランタイムオーバヘッドを軽減する**
  MVEE はレプリカ間のシステムコール同期において，レプリカとモニタの通信がパフォーマンスに大きく影

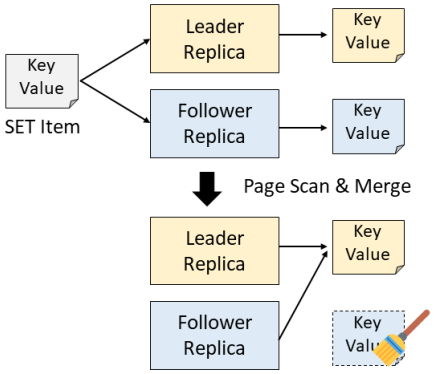

図 2 TwinsDB におけるレプリカ間ページ共有

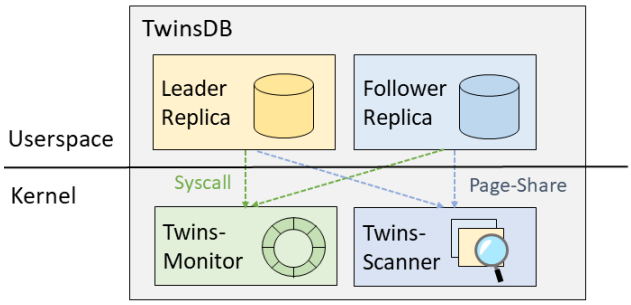

図 3 TwinsDB の構成

響する．また，インメモリ DB サーバの応答遅延は Web サービス全体のパフォーマンスに影響を及ぼす．オーバヘッドを最小にするため，インメモリ DB の動作特性に着目したシステムコール同期手法を設計する．

- **各レプリカの物理ページに同一内容のコンテンツが格納された際に高速に検出し，即座にページ共有を達成する**
  インメモリ DB に対する MVEE 適用は，各レプリカが大量の物理ページを生成し，そのうち共有可能なページは 1 ページのペアのみという状況である．このような状況においても高速に併合可能なページのペアを発見し，即座に併合されるようなページ共有手法を設計する．

4. 設計

図 3 に示すように，TwinsDB はレプリカ間のシステムコール監視および同期を行うためのコンポーネント Twins-Monitor とページスキャンを行うためのコンポーネント Twins-Scanner より構成される．はじめに Twins-Monitor における同期手法について述べ，次に Twins-Scanner におけるページ共有手法について述べる．

4.1 システムコールの同期

Twins-Monitor (モニタ) はシステムコールレベルでレプリカ間同期を行うコンポーネントであり，カーネル内で動作する．ユーザアプリケーションとして動作するモニタ形式とは異なり，カーネル内で動作することによってオーバヘッドの軽減とレプリカからの分離を達成することができる．しかしながら，レプリカの発行する全てのシステムコールを同期することは高いオーバヘッドを招くため，同期対象となるシステムコール数は可能な限り小さくする必要がある．また，攻撃に利用される可能性がある危険なシステムコールは列挙可能であるため，レプリカの整合性を保つ上で必要となる最小限のシステムコールとセキュリティ上の観点で同期が必要となるシステムコールに限定することで効率的な同期を実現する．

レプリカの整合性を保つ上で同期が必要となるシステムコールは以下の二種類に分類される．

- レプリカへの入力を統一するために同期が必要なシステムコール
- レプリカから外部の出力が一致しているか確認するために同期が必要なシステムコール

各レプリカへの入力が一致しない場合，各レプリカの仮想アドレス空間内に格納されるメモリ内容が変化し，システムコールの出力内容に差異が生じる可能性がある．このようなレプリカ間差異は攻撃の誤検知となるため，各レプリカへ入力が渡されるようなシステムコールが発行された場合にはモニタが干渉し，リーダーレプリカのみシステムコールを実行させた上で，その結果を残りのレプリカに渡す．たとえば sys_read システムコールでは，リーダーレプリカのみがファイルディスクリプタからの読み込みを行い，取得したメモリコンテンツと戻り値をフォロワーレプリカに共有する．リーダーの共有情報はリングバッファに格納され，フォロワーはリングバッファから取得することで実際にシステムコールを実行したかのように振る舞う．以上によって全レプリカのメモリ内容は統一され，攻撃発生時にのみレプリカ間差異が発生する．

攻撃発生時に正しく検出を行うため，レプリカから外部に対する出力を伴うようなシステムコールが発行された場合にはモニタが干渉し，全レプリカの内容が一致しているか確認した上で，リーダーレプリカのみ実行させる．たとえば，sys_write システムコールでは，全レプリカのシーケンス番号，引数によって指定されているポインタが指すコンテンツ，コンテンツの大きさが一致しているか確認する．

4.2 セキュリティ機構固有動作の許容

各レプリカに適用されるセキュリティ機構は固有のシステムコールを発行する場合がある．たとえば，Linux における ASan では /proc/self ディレクトリの読み込みを行い，自身のプロセスに関する情報取得を行う [16]．したがって，ASan 適用によって sys_open，sys_close といったファイルに関連するシステムコールが余分に発行されることになり，他のレプリカに他のセキュリティ機構を適用し

た場合にはレプリカ間のシステムコールに差異が発生する．

このような適用機構固有のシステムコール差異を吸収するため，モニタでは各レプリカがネットワークサービスを公開するシステムコールを発行するまでシステムコール差異を許容する．Sanitizer をはじめとする，アプリケーションの制御を一部変更するような機構はプロセス開始時に固有の動作を行う．たとえば，機構自身が使用するライブラリのロードや Shadow Memory といった機構自身がメタデータの管理に使用する領域のセットアップが該当し，前述の ASan 固有の動作についても起動時に実行される処理である．インメモリ DB において，攻撃者によるデータ挿入が可能となるのはネットワークサービス公開以降であり，それ以前についてはいかなる状態であっても攻撃は発生していないとみなすことができるため，全てのシステムコール差異を許容することで固有動作の吸収を可能とする．

4.3 インメモリ DB 向けの同期バッチ化

MVEE ではレプリカ間同期にかかるオーバヘッドが発生し，インメモリ DB をレプリカとする場合はデータベースサーバとしての応答遅延やスループット低下に繋がる．このような性能劣化は Web サービス全体へのパフォーマンスに大きく影響するため，同期オーバヘッドを軽減するための設計が求められる．

多くのアプリケーションは実行をプロファイリングすることで，ある処理単位内で呼び出されるシステムコール列を事前に把握することが可能である．この点に着目し，インメモリ DB におけるコマンド受付開始から応答送信までをひとつのフェーズと捉え，フェーズ内のシステムコール同期をバッチ化することで，同期にかかるオーバヘッドを削減する．

コマンド受付開始のシステムコールが発行されると，モニタはバッチモードを開始する．**図 4** に示すように，バッチモードではリーダーレプリカが先行してシステムコール実行を行い，実行結果をリングバッファに格納する．コマンドに対する応答を送信するシステムコールが発行されると，フォロワーレプリカが後行でリングバッファから実行結果の取得を開始する．全てのレプリカがコマンド応答送信のシステムコールまで到達したらバッチモードを終了し，通常の同期モードに戻る．たとえば，Memcached においてデータを取得する場合，クライアントから GET コマンドを送信するとバッチモードは開始され，VALUE を含む応答送信に到達するとバッチモードは終了する．

モニタはバッチモード内で発行されるシステムコール列の内容をあらかじめ認識しているため，攻撃者がバッチモード内で任意のシステムコールを発行した場合であっても，レプリカ間比較を行うことなく攻撃の検出が可能となる．

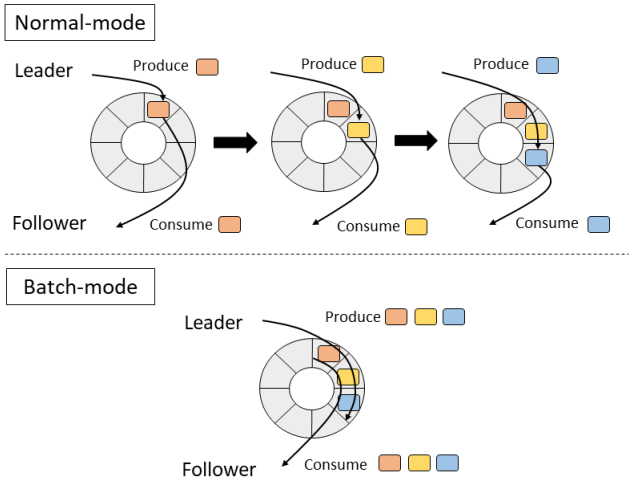

図 4 バッチモードにおけるリングバッファのコンテンツ格納と取得

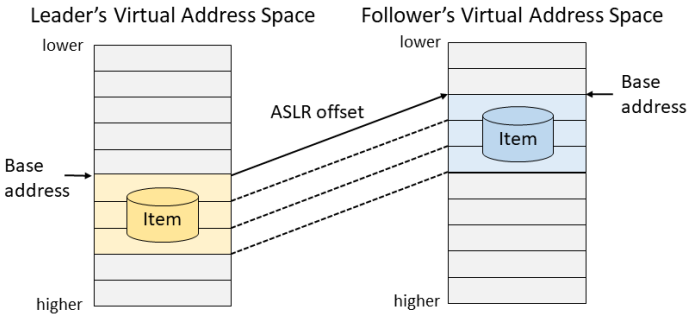

図 5 ASLR を考慮したページ探索と併合

4.4 共有ページの探索併合の高速化

Linux カーネルでは，物理メモリ消費を抑制するための機構として Copy-on-Write (CoW) や Kernel Same Page Merging (KSM) [28] を提供するが，これらの機構は MVEE 向けでないため，効率的なページ削減が期待できない．

CoW は sys_clone によって他プロセスへのメモリコピーが必要になった際に，実際に書き込みが行われるまでプロセス間でメモリページを共有する．CoW によって sys_clone においてはレプリカのページ削減が可能であるが，一時的かつ局所的な削減に過ぎず，ひとたび書き込みを行えば新たな物理ページが生成される．

KSM はプロセス間で同一内容のメモリページを併合することで物理メモリ消費量を削減する．KSM は 二種類の木構造 Stable Tree，Unstable Tree を持ち，仮想アドレス空間のスキャンによって得たページと同一内容のページがどちらの Tree に存在するか検索する．Stable Tree に存在する場合はヒットしたページとスキャンページを併合する．Unstable Tree に存在する場合はヒットしたページを Stable Tree に移した上でスキャンページを併合する．どちらの Tree にも存在しない場合にはスキャンページを Unstable Tree に追加する．MVEE では，あるレプリカの 1 ページと同一内容のページは別のレプリカの 1 ページのみであるため，KSM の共有方式は検索にかかるコストが高く，物理ページ数の増加に伴って共有速度の低下を招く．

Twins-Scanner (スキャナ) では，Stable / Unstable Tree のような構造を使用せず，レプリカ間で同一内容のメモリ箇所を仮想アドレスの対応から即座に探索し，併合することで高速な消費量抑制を実現する．インメモリ DB をレプリカとする MVEE は各レプリカに対して入力として渡されるデータアイテムが等しいため，スキャナはアイテムが格納されるページに限定し，一定周期ごとに対象の仮想アドレス範囲を探索する．

仮想アドレス空間において，アイテムが格納される範囲をスキャナが認識するため，レプリカがアイテムの格納領域を動的確保した際には，確保領域の先頭仮想アドレスとサイズを専用のシステムコールによってスキャナに通知する．専用のシステムコールによる通知にあたっては LD_PRELOAD によって対象の関数のフックを透過的に行うため，アプリケーションソースコードの変更は伴わない．

ターゲットとなる仮想アドレスの範囲を得たスキャナは，リーダレプリカによって通知された仮想アドレスに紐づく物理ページの内容と，それに対応するフォロワーレプリカの物理ページの内容を比較し，一致している場合にはページの併合を行う．レプリカに ASLR が適用されている場合，リーダレプリカのページと対応するフォロワーレプリカのページの仮想アドレスが必ずしも同一とは限らない．ASLR を考慮したページ探索を行うため，**図 5** に示すようにスキャナは仮想アドレス通知時にリーダレプリカとフォロワーレプリカの先頭アドレスのオフセットを計算し，フォロワーレプリカのページ内容確認時にはオフセットを加算した仮想アドレスを用いることで，レプリカ間でペアとなるページ同士を選出することを可能とする．

ページの併合はフォロワーレプリカのページテーブルエントリ (PTE) を書き換え，リーダレプリカのページに向けた上でフォロワーのページを解放することで可能となる．併合がなされたページに対してあるレプリカが直接書き込みを行うと別のレプリカの整合性が失われるため，併合済ページに対する書き込みは禁止する必要がある．そのため，PTE 書き換え時には書き込み保護権限を設定し，書き込み時にはページを分裂させる．

4.5 マルチプロセス/スレッドへの対応

レプリカが sys_fork や sys_clone システムコールによって新たなプロセスを生成した場合，親プロセスと子プロセスをまとめてひとつの監視単位とするとシステムコール同期に破綻が生じる．これはプロセスのスケジューリングが非決定的であり，レプリカ間で親プロセスと子プロセスの実行順が異なる可能性があるためである．したがって，モニタはリーダーレプリカとフォロワーレプリカの子プロセスの組を新たなひとつの監視単位とし，親プロセスのレプリカの組とは別々に同期を行う．スレッドに関しても同様であり，新たなスレッドの組を新たな監視単位とする．

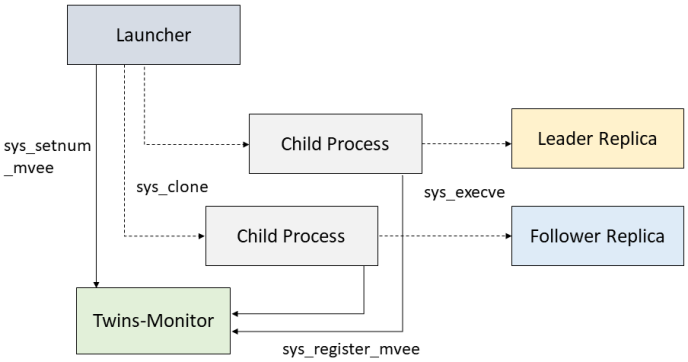

図 6 ランチャーを用いたセットアップ

マルチスレッドアプリケーションでは，共有メモリのアクセスについて考慮する必要がある．システムコールレベルの同期ではメモリアクセスのタイミング制御が不可能であるため，レプリカ間でスレッドの共有メモリアクセス順序が異なる可能性があり，システムコール差異の要因となりうる．本研究では，Bunshin [16] に基づいた対応方法を取る．これは多くのマルチスレッドアプリケーションが共有メモリアクセス前にロックを要求する点に着目し，ロックを取得するスレッドの順番がレプリカ間で同一になるように調整を行うことでシステムコール差異の発生を防ぐ．

5. 実装

5.1 モニタの起動とレプリカ実行の開始

インメモリ DB をレプリカとして稼働させるためには，モニタに対して稼働させるレプリカ数と各レプリカの登録を行う必要があり，これらは専用のシステムコールを発行することによって完了する．稼働予定数の登録については，稼働予定数を引数に取る sys_setnum_mvee システムコール，レプリカの登録については，引数を持たない sys_register_mvee システムコールによって行われ，登録数が稼働予定数に達するとモニタは処理を開始する．ただし，システムコール同期は全レプリカで特定のシステムコールが発行されるまでは行わない．詳細については 5.3 節にて述べる．

図 6 に示すように，モニタへの登録処理は専用のランチャープログラムによって行われる．まず，ランチャーは sys_setnum_mvee を発行し，モニタに対してレプリカの稼働予定数を通知する．次に sys_clone を発行し，子プロセスをレプリカ数分だけ生成する．生成されたそれぞれの子プロセスは sys_register_mvee によってモニタに対してレプリカの登録を行い，sys_execve によって対象のインメモリ DB のアプリケーションバイナリをプロセスのメモリ空間にロードする．以上の処理によってレプリカとしてモニタに登録され，それぞれのレプリカは実行を開始する．

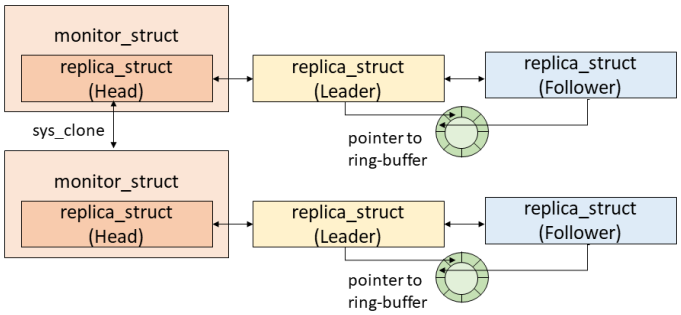

図 7 モニタとレプリカに関するデータ構造の構成

5.2 モニタの管理構造

sys_register_mvee によってレプリカ登録通知を受けたモニタは，replica_struct 構造体と monitor_struct 構造体を生成する．**図 7** にこれらの対応関係を示す．

replica_struct はレプリカごとにひとつ生成され，同一監視単位中の次のレプリカを指す双方向リストとなっている．replica_struct には，レプリカの Thread Group ID (TID) および Process ID (PID)，システムコール内容を同期するためのリングバッファへのポインタが含まれている．

monitor_struct は監視単位ごとにひとつ生成され，次の監視単位を指す双方向リストとなっている．マルチプロセスやマルチスレッドによって監視単位の新規生成が発生した場合には，新たな monitor_struct をリストに繋げる．monitor_struct には，replica_struct の先頭要素へのポインタが含まれている．

5.3 通常時のレプリカ間同期

レプリカの登録数が稼働予定数に達するとモニタは動作を開始し，レプリカが監視対象のシステムコールを発行する度に内容をリングバッファに記録する．リングバッファは syscall_log 構造体のエントリが循環するようなデータ構造となっており，syscall_log にはリーダレプリカのシステムコール番号，引数，戻り値が格納される．また，この段階ではまだシステムコール同期は行われず，sys_bind, sys_clone を除く全てのシステムコールは監視の対象外となっている．よって，リーダレプリカのシステムコール実行結果はフォロワーレプリカに共有されることなく，全レプリカは各々システムコールを実行する．したがって，この段階におけるセキュリティ機構固有のシステムコールは許容される．

レプリカ間同期は全レプリカで sys_bind システムコールの発行が行われたタイミングで開始される．sys_bind は IP アドレスとポート番号の組をソケットと対応付けるシステムコールであるため，sys_bind の発行によってネットワークサービスが開始するとみなし，以降のシステムコールをレプリカ間で同期することで，各レプリカの整合性を保ちながら攻撃を検出する．

同期パターンがレプリカに対する入力の統一と出力の監視に分類されることは 4.1 節にて述べた通りであるが，実装としてはさらに以下の五種類に分類される．

(1) T_PASS
　同期は不要
(2) T_COMP (e.g. sys_write，sys_sendmsg)
　実行前にコンテンツの内容確認を行う
(3) T_SYNC (e.g. sys_read，sys_recvfrom)
　リーダレプリカのみ実行し，リングバッファを通じて取得内容と戻り値を同期する
(4) T_CHECK (e.g. sys_open，sys_close)
　リーダレプリカのみ実行し，リングバッファを通じて戻り値を同期する
(5) T_SCHED (e.g. sys_clone)
　マルチプロセス/スレッド対応のための監視単位新規生成を行う

T_PASS に分類されるシステムコールは監視の対象外であり，同期を行わずに各々のレプリカが実行する．sys_bind が発行されるまでは，sys_bind，sys_clone を除く全てのシステムコールが T_PASS に該当し，発行がなされたタイミングで従来の同期パターンに切り替えられる．

T_COMP では，システムコール引数からユーザ空間のコンテンツ領域のポインタとサイズを取得し，コンテンツのハッシュ値を取ることで，全てのレプリカでハッシュ値が一致していることを確認する．コンテンツが一致している場合は，リーダレプリカのみ実行を許可し，リーダの戻り値を全てのフォロワーレプリカに同期する．

T_SYNC では，リーダのみ実行を許可し，リーダの取得コンテンツ，戻り値をフォロワーに同期する．T_SYNC がユーザ空間のコンテンツと戻り値を同時に同期する点に対して，T_CHECK は戻り値のみの同期を行う．

T_SCHED は，全てのレプリカに実行を許可し，新たな監視単位を生成する．T_SCHED に唯一該当するシステムコールである sys_clone の戻り値は新規プロセス/スレッドのスレッド ID が返される．この点を考慮し，sys_clone の引数に CLONE_THREAD フラグが含まれる場合には TGID は生成元と同様，PID は戻り値となるように replica_struct を生成し，フラグが含まれない場合には TGID と PID が戻り値となるように replica_struct を生成する．そして，monitor_struct についても生成した上で，monitor_struct, replica_struct それぞれの双方向リストを設定する．

5.4 バッチモードにおけるレプリカ間同期

現在の TwinsDB プロトタイプは Memcached におけるレプリカ間同期のバッチ化をサポートする．Memcached では，sys_epoll_wait システムコールによってクライアントからのコマンド受け付けを開始し，sys_sendmsg システムコールによってコマンド応答結果を送信する．したがっ

て，全レプリカで sys_epoll_wait が発行されたタイミングでバッチモードに移行し，sys_sendmsg が発行されるまでリーダの実行を先行させる．リーダが sys_sendmsg まで達したタイミングでフォロワーが後行で実行を開始し，フォロワーが sys_sendmsg まで達したタイミングでバッチモードを終了し，通常の実行モードに戻る．通常の実行モードに戻ったため，sys_sendmsg は T_COMP に従って同期が行われ，以降についても 5.3 節の同期パターンに従った同期が行われる．

5.5 スキャナの起動とページ探索の開始

5.3 節にて述べたように，モニタはレプリカの登録数が稼働予定数に達したタイミングで開始されるが，スキャナについても同様のタイミングで開始される．スキャナは専用のカーネルスレッドとして動作し，一定周期ごとにリーダレプリカの仮想アドレスにおける特定範囲を探索する．スキャナ開始時点では探索対象の仮想アドレス範囲が不明であるため，全てのレプリカから仮想アドレスが通知されたタイミングで探索を開始する．モニタへの通知は，posix_memalign を LD_PRELOAD によってフックし，確保したサイズと先頭の仮想アドレスを引数に取る専用のシステムコール sys_register_vaddr_mvee を発行することによって完了する．

5.6 ページの探索と併合

sys_register_vaddr_mvee によって全てのレプリカの仮想アドレスを得たスキャナはページの探索を開始する．まず，リーダと各フォロワーとの仮想アドレスオフセットを計算する．次にリーダの対象仮想アドレス範囲内について，ページごとにハッシュ値を取り，オフセットを反映させたフォロワーのページについてもハッシュ値を取る．ハッシュ値が一致している場合はページの併合を行い，一致しない場合は併合せずに次のページ探索を行う．

ページの併合はリーダの物理ページを参照するようにフォロワーの PTE を書き換え，ページの書き込み時に CoW による分裂が発生するように pte_wrprotect を設定し，該当のリーダページを rmap に登録する．元々使用していたフォロワーページは rmap から削除した上で put_page によってページを解放する．

5.7 マルチスレッドへの対応

マルチスレッドにおいてはロックの取得順の強制を行う必要があるため，Pthread Library におけるロック取得関数である pthread_mutex_lock ならびに pthread_mutex_trylock を LD_PRELOAD によってフックし，専用のシステムコール sys_synclock_mvees (synclock) を発行する．まず，synclock 内では，同一監視単位内の全てのレプリカが揃うまで実行をスリープさせ，揃い次第リーダレプリカにオーダー ID を割り当てる．オーダー ID は 0 から始まる整数値であり，モニタは各監視単位のリーダが持つオーダーとは別にロックの取得が完了した最新のオーダーを持つ．次に，リーダは自身に割り当てられたオーダーが最新のオーダーと一致するか確認を行う．一致しない場合は一致するまで実行をスリープする．一致する場合はフォロワーのスリープを解除した上で synclock を終了し，ロックを取得する．ロックの取得後には再度 synclock を発行し，モニタの持つ最新のオーダーをインクリメントすることで，一致待ちのレプリカのロック取得が許可される．

6. 評価実験

6.1 実験環境

TwinsDB の性能評価に使用するコンピュータを **表 1** に示す． HP ProLiant DL980 G7 は 8 個のノードより構成される NUMA マシンであり，各ノードには 10 個のコアと約 128 GiB のメモリが搭載されている．OS カーネルには TwinsDB が実装された Linux 4.4.185，Linux ディストーションには Ubuntu 16.04 LTS を使用した．

6.2 実験パターン

大量のメモリを消費するマイクロベンチマークプログラムをレプリカとした場合と Memcached 1.5.22 をレプリカとした場合で評価実験を行った．また，各評価実験は以下のパターンで行い，結果を比較した．

- TwinsDB 未適用の単体実行
- TwinsDB 適用のマルチバリアント実行

セキュリティ機構は ASLR，SSP，ASan を使用し，TwinsDB 適用時のレプリカ数は 2 ならびに 3 とする．単体の場合には ASLR + SSP + ASan を適用し，レプリカ数 2 の場合には，リーダレプリカに ASLR + SSP，フォロワーレプリカに ASLR + SSP + ASan を適用する．また，レプリカ数 3 の場合にはリーダおよびフォロワー 1 に ASLR + SSP，フォロワー 2 に ASLR + SSP + ASan を適用する．

6.3 マイクロベンチマークにおける評価

ベンチマークプログラムの動作内容を以下に示す．

(1) 10 GiB 分のページアライメント領域を動的確保する
(2) 確保領域を 4 KiB のブロックに分割し，各ブロックの先頭 1 KiB 分にランダムな値を書き込む
(3) 全ブロックへの書き込み完了後，各ブロックの先頭

表 1 実験環境

Machine	HP ProLiant DL980 G7
CPU	Intel(R) Xeon(R) CPU E7-4870 @ 2.40GHz
CPU core	80
Memory	1009909 MiB

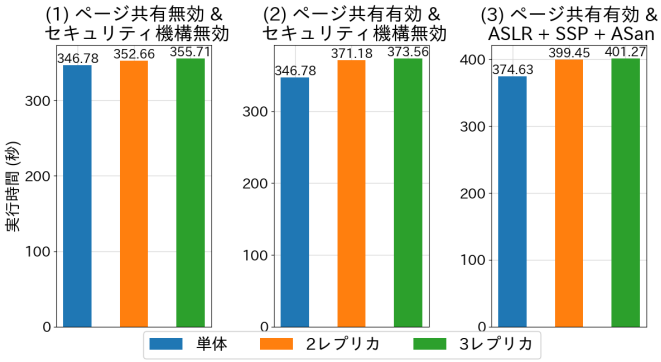

図 8 read-only ワークロードにおける実行時間比較

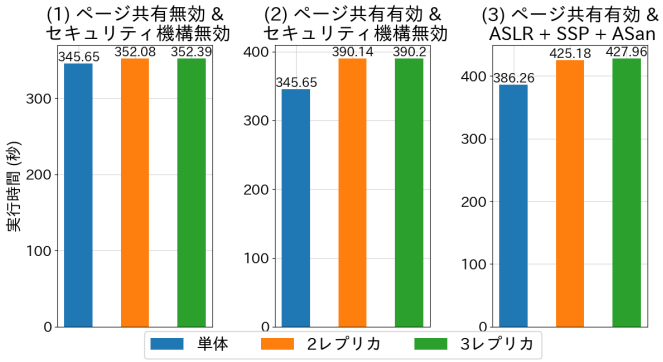

図 9 write-only ワークロードにおける実行時間比較

1KiB 分について以下のいずれかの操作を実行する
(a) 局所変数に読み込む (read-only)
(b) ランダムな値を書き込む (write-only)

ベンチマークプログラムは C 言語にて実装を行った．ブロックに対して行われる操作はプログラム起動時にいずれかが選択され，全ブロックに同一の操作が行われる．また，ランダムな値を生成するためのシードは固定の値を使用し，レプリカ間で共通のランダム値が生成される．

6.3.1 実験方法

6.2 節にて述べた各実験パターンについて，read/write-only ワークロードをそれぞれ実行した場合の実行時間と物理メモリ消費量を測定し，結果を比較する．実行時間計測にはマイクロ秒単位までの精度が保証される getrusage を使用し，物理メモリ消費量計測には全レプリカの Proportional Set Size (PSS) を取得し，合計する．

6.3.2 実験結果

まず，各ワークロードにおける実行時間の比較を **図 8**, **図 9** に示す．read-only ワークロードにおいて，TwinsDB を適用することによって発生するオーバヘッドは最大 7.72 % 程度であり，write-only ワークロードにおいては最大 12.89 % 程度であった．write-only ワークロードでは，ページを併合した後に書き込むことによって再度ページが分裂されるため，ページ分裂にかかる処理が追加オーバヘッドの要因となっている．

次に，各ワークロードにおけるメモリ消費量の比較を **図 10**, **図 11** に示す.read-only ワークロードでは単体と比較して最大 28.82 % 程度の PSS 増加であり，write-only ワークロードでは最大 58.12 % 程度の増加であった．

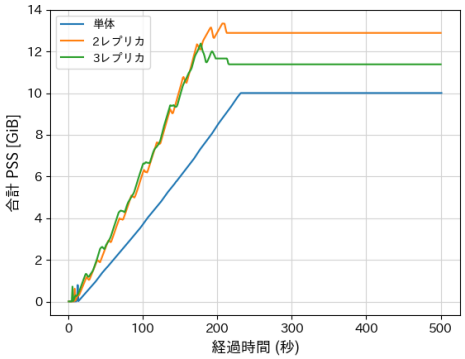

図 10 read-only ワークロードにおけるメモリ消費量比較

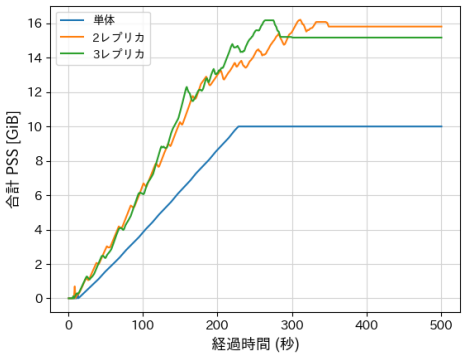

図 11 write-only ワークロードにおけるメモリ消費量比較

6.4 実アプリケーションにおける評価

6.4.1 実験方法

6.2 節にて述べた各実験パターンについて，Memcached

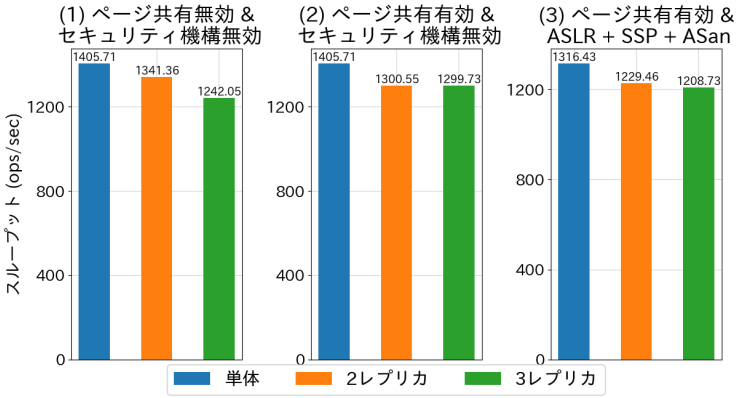

図 12 YCSB a ワークロードにおけるスループット比較

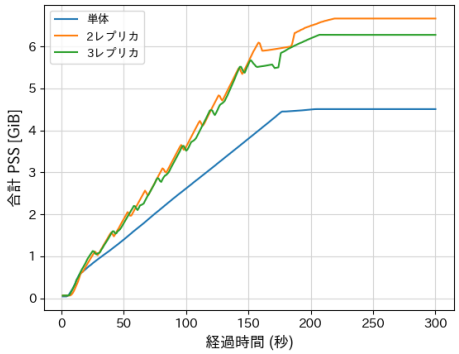

図 13 YCSB a ワークロードにおけるメモリ消費量比較

に対して YCSB [29] ベンチマークを実行した場合のスループットと物理メモリ消費量を測定し，結果を比較する．

ベンチマークはローカルホスト内から実行するものとし，YCSB Load によって 4 KiB のアイテムデータを 100,000 個ロード，YCSB Run によってワークロード a を 100,000 回実行する．ワークロード a はデータアイテムの読み込みと書き込みを 50 % ずつの割合で行う内容となっているため，ロード済アイテムデータに対する SET および GET コマンドが 50,000 回ずつ実行される．

物理メモリ消費量計測にはマイクロベンチマークと同様に PSS の合計値を用いる．

6.4.2 実験結果

スループットの比較を **図 12**，メモリ消費量の比較を図 12 に示す．スループットについては，TwinsDB 適用によって最大 11.64 % 程度の低下が生じることが確認された．メモリ消費については，単体実行と比較して最大 39.23 % 程度の増加によって実行が可能であることが確認された．

7. おわりに

本研究では，インメモリ DB のセキュリティ強化を目的とした MVEE である TwinsDB を提案した．既存研究の MVEE はレプリカのメモリ消費について考慮されていないため，インメモリ DB レプリカを複数稼働させるような MVEE の実行は困難であった．TwinsDB ではインメモリ DB レプリカに入力として渡されるデータアイテムが等しい点に着目し，アイテムページを高速に併合することでメモリ消費の抑制を行い，インメモリ DB レプリカの複数稼働を可能にする．また，レプリカ間でセキュリティ機構固有の振る舞いを吸収しながら，インメモリ DB の動作特性に着目したシステムコール同期を行うことで，セキュリティの強化とオーバヘッドの軽減を達成する．

TwinsDB を Linux カーネルに実装し，マイクロベンチマークと Memcached を用いた性能評価実験を行った．その結果，多様なセキュリティ機構の適用が可能であり，メモリ消費量を大幅に抑制しながら実行が継続された．また，ワークロードの種類によっては MVEE 未適用の単体実行に近いパフォーマンスとメモリ消費が達成可能であることが確認された．バッチモードの有効時と無効時における性能比較については今後実験を行う予定である．

今後として，より選択的なページ併合の設計が求められる．現在のスキャナは特定の仮想アドレス範囲内の全てのページを周期的に探索しているが，これらの中には既に併合済のページも含まれるため，インメモリ DB のメモリスケールの増加に伴って無駄な探索が多く発生する．このような無駄なページ探索を避けるためには，アイテムデータの格納タイミングや CoW によるページ分裂タイミングをトリガーとして該当の範囲のみを探索するイベントベースのスキャナによって改善が可能である．また，アイテムデータ格納ページにはポインタなどのメタデータが格納されている場合があり，このような場合にはポインタの差異によって併合が不可能となる．したがって，ASLR 有効時であってもページ中のポインタ差異を許容するような設計が求められる．

参考文献

[1] ORACLE: MySQL Database Service, `https://www.mysql.com/`. (accessed 2020-10-18).
[2] dormando: memcached - a distributed memory object caching system, `https://memcached.org/`. (accessed 2020-10-18).
[3] redis labs: redis: an in-memory database that persists on disk, `https://redis.io/`. (accessed 2020-10-18).
[4] Netflix: A distributed in-memory data store for the cloud, `https://github.com/Netflix/EVCache`. (ac-

cessed 2020-10-18).

[5] Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P., Stafford, D., Tung, T. and Venkataramani, V.: Scaling Memcache at Facebook, *10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, Lombard, IL, USENIX Association, pp. 385–398 (2013).

[6] Yang, J., Yue, Y. and Rashmi, K. V.: A large scale analysis of hundreds of in-memory cache clusters at Twitter, *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, USENIX Association, pp. 191–208 (2020).

[7] CVE-2020-10931 (memcached): (Remote DOS attack) 0day buffer overflow vulnerability reveal 629, `https://github.com/memcached/memcached/issues/629`. (accessed 2020-10-18).

[8] CVE-2020-14147 (redis): [FIX] revisit CVE-2015-8080 vulnerability 6875, `https://github.com/redis/redis/pull/6875`. (accessed 2020-10-18).

[9] Spengler B: PaX: The Guaranteed End of Arbitrary Code Execution, `https://grsecurity.net/PaX-presentation.ppt`. (accessed 2020-10-18).

[10] Crispan Cowan, Calton Pu, D. M. J. W. P. B. S. B. A. G. P. W. Q. Z.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, USA, USENIX, pp. 63–78 (online), available from ⟨https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention/⟩ (1998).

[11] Zhilong Wang, Xuhua Ding, C. P. J. G. J. Z. B. M.: To Detect Stack Buffer Overflow with Polymorphic Canaries, *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, Luxembourg City, Luxembourg, IEEE/IFIP, pp. 243–254 (online), available from ⟨https://ieeexplore.ieee.org/document/8416487/⟩ (2018).

[12] Google: AddressSanitizer, ThreadSanitizer, MemorySanitizer, `https://github.com/google/sanitizers`. (accessed 2020-10-18).

[13] Konstantin Serebryany, Derek Bruening, A. P. D. V.: AddressSanitizer: A Fast Address Sanity Checker, *Proceedings of the 2012 USENIX Annual Technical Conference*, Boston, MA, USENIX, pp. 309–318 (2012).

[14] Evgeniy Stepanov, K. S.: MemorySanitizer: Fast detector of uninitialized memory use in C++, *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization*, IEEE, pp. 46–55 (2015).

[15] Clang 12: UndefinedBehaviorSanitizer, `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`. (accessed 2020-10-18).

[16] Xu, M., Lu, K., Kim, T. and Lee, W.: Bunshin: Compositing Security Mechanisms through Diversification, *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, USENIX Association, pp. 271–283 (2017).

[17] Volckaert, S., Coppens, B., Voulimeneas, A., Homescu, A., Larsen, P., Sutter, B. D. and Franz, M.: Secure and Efficient Application Monitoring and Replication, *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, Denver, CO, USENIX Association, pp. 167–179 (2016).

[18] Österlund, S., Koning, K., Olivier, P., Barbalace, A., Bos, H. and Giuffrida, C.: KMVX: Detecting Kernel Information Leaks with Multi-Variant Execution, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, New York, NY, USA, Association for Computing Machinery, p. 559572 (online), DOI: 10.1145/3297858.3304054 (2019).

[19] Salamat, B., Jackson, T., Gal, A. and Franz, M.: Orchestra: Intrusion Detection Using Parallel Execution and Monitoring of Program Variants in User-Space, *Proceedings of the 4th ACM European Conference on Computer Systems*, EuroSys '09, New York, NY, USA, Association for Computing Machinery, p. 3346 (online), DOI: 10.1145/1519065.1519071 (2009).

[20] Hosek, P. and Cadar, C.: VARAN the Unbelievable: An Efficient N-Version Execution Framework, *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, New York, NY, USA, Association for Computing Machinery, p. 339353 (online), DOI: 10.1145/2694344.2694390 (2015).

[21] Pina, L., Andronidis, A., Hicks, M. and Cadar, C.: MVEDSUA: Higher Availability Dynamic Software Updates via Multi-Version Execution, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, New York, NY, USA, Association for Computing Machinery, p. 573585 (online), DOI: 10.1145/3297858.3304063 (2019).

[22] Amazon Web Service Inc: Amazon DynamoDB Accelerator (DAX), `https://aws.amazon.com/dynamodb/dax/`. (accessed 2020-10-18).

[23] Jonathan Corbet: Toward more predictable and reliable out-of-memory handling, `https://lwn.net/Articles/668126/`. (accessed 2020-10-22).

[24] Crispin Cowan, Perry Wagle, C. P. S. B. J. W.: Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, *Proceedings of the DARPA Information Survivability Conference and Exposition*, DISCEX ' 00 2, IEEE, pp. 119–129 vol.2 (2000).

[25] Byoungyoung Lee, Chengyu Song, Y. J. T. W. T. K. L. L. W. L.: Preventing Use-after-free with Dangling Pointers Nullification, *Proceedings of the 2015 Network and Distributed System Security Symposium*, The Internet Society, pp. 8 – 11 (2015).

[26] Kees Cook: Kernel address space layout randomization, `https://lwn.net/Articles/569635/`. (accessed 2020-10-20).

[27] Pina, L., Grumberg, D., Andronidis, A. and Cadar, C.: A DSL Approach to Reconcile Equivalent Divergent Program Executions, *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, USENIX Association, pp. 417–429 (2017).

[28] kernel.org: Kernel Samepage Merging, `https://www.kernel.org/doc/html/latest/admin-guide/mm/ksm.html`. (accessed 2020-10-18).

[29] brianfrankcooper: Yahoo! Cloud Serving Benchmark, `https://github.com/brianfrankcooper/YCSB`. (accessed 2020-11-02).