Vol.2018-OS-144 No.8
2018/7/31

ページ共有を利用した Multi-Variant 監視効率化手法

清水 祐太郎[†1,a)]　山田 浩史[†1,b)]

概要：システムソフトウェアには未だに C/C++ などの型安全でない言語で記述されているものも多く，依然として攻撃の脅威に晒されている．ASLR や SSP のような対攻撃性を高めるセキュリティ機構が実装されているが，いずれの機構も情報の漏洩によって容易に回避できる可能性がある．そこで，多くの攻撃が特定のセキュリティ機構やそのバージョンに依存することに着目して，アプリケーションの複製を同時に実行して異なるセキュリティ機構を同時に稼働させる Multi-Variant 監視機構 (MVEE) が存在する．しかしながら，メモリを大量に利用するアプリケーションに対して MVEE を適用する場合にはメモリの浪費が懸念される．物理メモリの逼迫によってスラッシングが発生などすると，MVEE の監視下にない他のプロセスに対しても性能に悪影響を及ぼす恐れがある．本研究では，内容が一致するページを併合することでビッグメモリアプリケーションに対して効率的に MVEE を適用する方法を提案する．また，複製したアプリケーション同士でロックの取得順を強制することで，マルチスレッドアプリケーションに対しても対応する．Linux kernel 4.13.9 に提案手法の実装を行い，物理メモリの消費量を計測したところ，最大でアプリケーションを単体で動作させたのと同程度までメモリの消費を抑制することができた．また，Redis に対しても提案手法を実装し，動作の確認を行った．

1. はじめに

ソフトウェアの脆弱性を利用した攻撃は広く知られている．様々な対策機構が提案，実用化されているにもかかわらず，依然としてソフトウェアは攻撃の脅威に晒されている．Java や C# のような型安全なプログラミング言語の登場によってこれらの問題は緩和されてているものの，未だ多くのシステムソフトウェアは C/C++ などの安全でない言語で記述されているものも多い．そこで，近年のオペレーティングシステム (OS) やコンパイラには，脆弱性に対する攻撃を成功しにくくするためのセキュリティ機構が実装されている．Address Space Layout Randomization (ASLR) [1] ではアプリケーションの実行時に仮想アドレス空間にマッピングするバイナリやスタック，ヒープなどのアドレスをランダムにする．そうすることで攻撃の際に必要となるデータが存在するアドレスを不明確にする．Stack Smashing Protector (SSP) [2] では，関数のスタックフレームの上位に Canary と呼ばれるランダム値を配置しておく．関数から返る際にこの値をチェックすることでスタック破壊を検知し，アプリケーションを停止させることができる．しかしながら，いずれの機構も情報の漏洩を伴うバグが存在している場合は，攻撃者は容易に機構を回避することが可能となってしまう．

そこで，多くの攻撃が特定のセキュリティ機構やそのバージョンに依存することに着目して，複数のセキュリティ機構を稼働させる Multi-Variant Execution Environment (MVEE) [3–10] という手法が存在する．MVEE では，同一アプリケーションを異なるセキュリティ機構下や異なるバージョンで稼働させながら，それぞれの動作を比較する．具体的には，アプリケーションの複製 (バリアントと呼ぶ) を別々に実行し，それぞれが発行するシステムコールやプロセスの状態をモニタが監視する．挙動が異なるバリアントを検知するとシステムは攻撃を受けていると認識し，全てのバリアントを直ちに終了させることができる．MVEE と極めて近しい動作を行うが，全く同じアプリケーションではなく，バージョンが多少異なるアプリケーションを同時に動作させる N-Version [11, 12] と呼ばれる手法もある．これらの手法では，ASLR や SSP といった既存のセキュリティ機構を活かしながら，よりセキュリティを向上させることができる．同一アプリケーションを実行しても，仮想アドレス空間や Canary 値は異なる．そのため，特定のバリアントのみを狙った Exploit が他のバリアントでも攻

[†1] 現在，東京農工大学
Presently with Tokyo University of Agriculture and Technology
[a)] simiyu@asg.cs.tuat.ac.jp
[b)] hiroshiy@asg.cs.tuat.ac.jp

撃を成功させる確率は非常に低い．

しかしながら，Memcached [13] などの In-memory DB のようなメモリを大量に利用するアプリケーション (ビッグメモリアプリケーション) に対して MVEE を適用する場合ではメモリの浪費が懸念される．バリアント同士は互いに独立した仮想アドレス空間を持っている．そして，それぞれが別々の物理メモリを確保して利用している．たとえば，Amazon DynamoDB Accelerator (DAX) [14] では 488 GiB までメモリを搭載することが可能であり，こうしたアプリケーションに MVEE を適用すると大量のメモリが必要となる．これにより，バリアント数を増やすことが困難となる．また，このメモリの逼迫は，近年主流であるサーバ統合の集約率を低下させてしまう．

本研究では，ビッグメモリアプリケーションに対して効率的に MVEE を適用する方法を提案する．提案機構では，全バリアントに対して与えられる入力は等しいため，内部に記録されるデータも同じものになることに着目する．提案機構は，各バリアントのメモリを走査し，同一内容のメモリ領域を検出，共有しながら実行を進める．これにより，ビッグメモリアプリケーションの複数稼働時の使用メモリ量を削減する．

本論文の構成は次のとおりである．第 2 章で本研究の背景と問題点について触れ，第 3 章でその解決策として本手法を提案する．第 4 章では提案手法を実現するためのシステムの設計，第 5 章では Linux カーネルおよびアプリケーションへの実装方法について述べる．第 6 章において提案手法の評価を行い，第 7 章で本研究のまとめと今後の課題を示す．

2. 背景

2.1 既存のセキュリティ機構

システムプログラムにおいて，メモリ管理やメモリへのアクセス方法のミスに起因する脆弱性が存在する．領域外参照や Use-After-Free [15], 未初期化領域の読み込みによるセンシティブなデータのリークなどが挙げられる．Java や C# のような型安全なプログラミング言語ではこれらの問題は緩和されてきた．しかし，未だ多くのプログラムは C/C++ などの安全でない言語で記述されているため，これらの問題は解決はされていない．そのため，これまでに脆弱性に対しての攻撃を通しにくくすることを目的とした緩和機構が開発されてきた．

Address Space Layout Randomization (ASLR) [1] とは，バイナリやスタック，ヒープなどをランダムな位置に配置するセキュリティ機構である．攻撃者はライブラリ内に存在しているコードやヒープやスタックに配置されたデータなど，既にメモリ内に存在している情報を利用して Exploit を構築することが多い．これらを無作為に配置することによって，攻撃のために必要なデータの位置を推測されにくくする．Stack Smashing Protection (SSP) [2] は，スタックオーバーフロー [16] による局所変数およびスタックフレームの破壊を緩和・検知する機構である．これは gcc [17] をはじめとするコンパイラによって実現される．SSP では，関数の引数や他の局所変数の改竄を難しくする．オーバーフローし得る配列よりも下位に局所変数を配置し，引数も同様に下位に移動する．また，Canary と呼ばれるランダムな値を局所変数とベースポインタとの間に配置することでこの機構ではベースポインタとリターンアドレスの改竄を検知することもできる．Control Flow Integrity (CFI) [18] では，関数の間接呼び出し時に遷移先を検証することで，攻撃者の用意した処理に実行を遷移させることを防ぐ．この機構はコンパイル時に，呼び出しの直前および戻った直後に検証のためのコードを挿入することで実現する．それぞれの遷移先と戻り元には固有の ID を持たせておき，遷移直前にこれを設定，遷移直後に正しい ID を持っているかどうかを確認する．このような検証処理を挟むことで，vtables overwrite による別の関数の呼び出しや，スタックオーバーフローによるリターンアドレスの書き換えを基にした ROP 攻撃などを防ぐことができる．

これらのセキュリティ機構は攻撃を緩和することはできるが，決して完璧であるとは言えない．ASLR や SSP は，メモリアドレスや Canary といったセンシティブな情報が漏洩してしまうと，容易に突破が可能になってしまう．攻撃者は漏洩した情報を基に Exploit を組むためである．また CFI は，可変長引数関数を間接呼び出しする場合には，動作が確定的ではないため対応できないことが知られている [19]．ここで述べた既存のセキュリティ機構は全て併用することは可能だが，なかには併用が不可能な機構も存在する．例えば，AddressSanitizer(ASan) [20] と MemorySanitizer(MSan) [21] がそれに当たる．ASan は Use-After-Free や Heap Overflow, Stack Overflow のようなメモリエラーを検出する．MSan は未初期化領域の読み込みを検出する．MSan では低位アドレスをアクセス不能領域として保護するが，ASan では低位アドレスを Shadow Memory として確保し，状態を管理するために利用している．このように互いの機構が競合してしまうため，これらはいずれか片方ずつしか適用できず，同時に利用することはできない [11].

2.2 Multi-Variant Execution Environment

MVEE では一つのアプリケーションのレプリカとなるプロセスを作成し，それぞれの挙動を比較することでセキュリティの向上を図る．2.1 節で挙げた既存のセキュリティ機構では，先述の通り情報の漏洩によって途端に攻撃に対する防御力を下げる．MVEE では，各レプリカごとに仮想アドレス空間や Canary 値も異なっている．そのため，特定のレプリカのみを狙った攻撃がたとえ成功したと

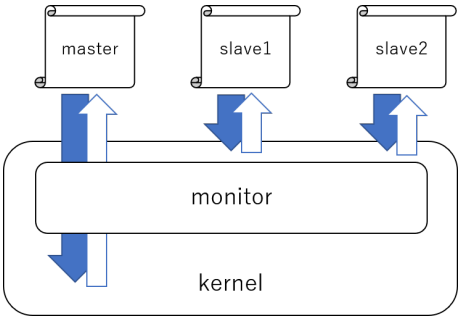

図 1 モニタとバリアント

しても，他のレプリカに対しては失敗するため，攻撃の検知及び緩和が可能となる．また，互いに干渉して同時に適用することのできなかったセキュリティ機構も，バリアントごとに部分的に適用 [11] することも可能になる．

MVEE は図 1 に示すように，モニタとレプリカ（バリアント）から成る．モニタはバリアント全体の動作の指揮を行う役割を担う．モニタが行うバリアントの監視はあくまで挙動の比較であり，メモリの内容や状態の完全一致を狙うものではない．そのため，ASLR や SSP のようなオペレーティングシステムやコンパイラに組み込まれているセキュリティ機構を活かすことができる．

動作の比較を行うために，バリアントは一定の間隔で同期を行う必要がある．一つの命令が実行されるごとにプロセスの状態を取得することも可能だが，これはあまりにもオーバーヘッドが大きいため実用的ではない．最も一般的な同期のタイミングは，プロセスがシステムコールを発行する際である．システムコールはユーザアプリケーションが単体で行うことのできない処理をカーネルに依頼するものである．情報の漏洩や新たなプロセスの立ち上げなど，攻撃者が何かしらの任意の動作をプログラムに行わせようとした際には必ず行われる処理である．

モニタは特定のバリアントをマスターと定め，他のバリアントをスレーブとして扱うマスターバリアントの発行したシステムコールとその結果を記録しておき，各スレーブバリアントが同期ポイントに達するタイミングで動作を比較する．異なる挙動を検出した場合，モニタは攻撃を受けたと判断し，直ちに全バリアントの実行を停止させることができる．

2.3 関連研究

MVEE については，モニタの実現方法や高速化，マルチスレッドへの対応など様々な側面から研究がなされている．MvArmor [4] はハードウェアの仮想化支援を受けたユーザアプリケーションレベルの MVX （= MVEE） モニタである．モニタは Dune [22] を利用して仮想環境上で動作する．そのため，モニタはカーネルに手を加えずに特権命令を発行することができる．モニタは仮想環境においてプロセスの状態に直接アクセスすることができるため，コンテキストスイッチの増加を避けることができる．アプリケーションが発行した syscall を受け取ったモニタは，vmcall を発行する．この際，VM Exit が行われてホストのカーネルにロードされた Dune module に処理が渡るが，syscall ごとに毎回この処理を行うのはコストが大きい．そこで，本手法ではメモリ管理や pid の取得など，一部のシステムコールは VM Exit せずにモニタ内で処理を行う．

Orchestra [5] はユーザ空間で動作するモニタである．sys_ptrace() システムコールを用いてバリアントを監視することで，OS カーネルを変更せずにモニタの実装を実現する．同期ポイントはバリアントがシステムコールを呼び出す際としており，システムコールの番号や引数の比較を行う．本手法では脅威モデルとしてスタックオーバーフローを据えている．gcc に変更を加えることで，スタックの成長方向が逆 [7] となるプログラムを用意する．このプログラムを通常のプログラムと共に動作させることで，一方ではスタックオーバーフローによるリターンアドレスが成功するが，もう一方では失敗するようになる．

GHUMVEE [3] と ReMon [9] を拡張することで実装された Taming Parallelism [6] では，特定のバリアントにおけるイベントの実行順序を補足し，他のバリアントで再現することでバリアント間における動作の多様化を防ぐ手法を提案している．MVEE にはマルチスレッドへの適用が困難という問題が存在する．バリアントごとでスレッドのスケジューリング順序が異なると，それに伴ってシステムコールや共有メモリへのアクセスなどの順番が変化し，動作が多様化してしまう恐れがある．動作の多様化を検知したモニタは，これが攻撃によるものなのか否か区別することができず，誤検知に繋がる．本手法ではアプリケーションがロックを取る順番を一致させ，結果的に同一順序でメモリにアクセスを行うようにする．ロックの順序を監視するためにバイナリレベルでロックに相当する命令を検出し，LLVM [23] を用いて命令の前後に同期用関数を挿入する．この関数内部で専用のシステムコールを発行し，適当なロックの取得を試みているスレッドのみ許可し，他はブロックする．また，ストール時間を減らすために WoC と呼ばれる論理クロックの概念を導入し，依存関係のあるロック操作の順序のみを再現する．

Disjoint Code Layouts [10] では，GHUMVEE [3] を用いて ROP 攻撃の成功確率を格段に低く抑える手法を提案している．対象バイナリが同一アドレスにマッピングされる場合，いずれのバリアントにも同じアドレスに同じ gadget が存在しているため，MVEE を用いたとしても ROP は成功してしまう．そこで，本手法ではバイナリのロード時にバリアント同士で被らないアドレスに配置を行うようにし，事実上 ROP 攻撃を不可能にする．モニタは sys_mmap() をフックし，PROT_EXEC が立っていた場合には配置アドレスを書き換えて処理を継続する．この処理

をローダに対しても適用することで，バイナリの独立した位置への配置を実現する．

Bunshin [11] は，セキュリティ機構を部分適用させたバリアントを作動させる N-Version システムである．メモリエラーに対する様々な対策手法が提案されているが，それらはオーバーヘッドが大きく，中には実行時間が 2 倍以上になるようなものも存在する．対策を一部分にのみ適用すると，実行時間の回復は見込めるがその反面保護性能は低下してしまう．また，複数の対策手法を同時に適用することが難しい場合がある．本手法では，複数の対策手法を別々に施したバリアントを用意し，それらを統合して監視する．コストが大きい対策手法については，同一の手法を一部ずつ異なる部分に対して適用したバリアントを用意することでオーバーヘッドの減少を試みる．部分適用の分割部分の決定のために対策手法の適用の有無でプロファイリングを行い，check code を挿入する．対策手法の適用と check code 挿入の位置によってバリアントの実行時間を調整し，それぞれのオーバーヘッドが同じになるようにする．バリアントの同期ポイントはシステムコールの呼び出し時としている．

2.4 問題点

MVEE ではバリアントを生成した分だけ物理メモリは消費される．バリアントはそれぞれ独立した仮想アドレス空間を持っている．Linux には無駄なページの複製および物理ページの消費を低減するため，Copy-on-Write と呼ばれる機構が存在する．sys_fork() などによってページの複製が必要となった際，一時的に同一物理ページを参照するようにし，書き込みがあった際に初めて複製を行う．MVEE のバリアントについても同様である．sys_fork() してバリアントを作成した際，マッピングされたページに書き込みを行うまでは物理ページは共有されている．しかし，ひとたび書き込みを行えば新たな物理ページが確保される．バリアントは並列に実行されてはいるが，メモリにアクセスするタイミングは必ずしも一致するということはない．そのため，常にページを共有しておくことは不可能であり，最終的にほとんどのページを個別の物理ページに割り当てることになる．

大量にメモリを消費するプログラムに MVEE を適用する場合，メモリ使用長は過剰に増大してしまう．MVEE を適用して n 個のバリアントを作成した場合，消費する物理メモリは n 倍近くとなる．数 MiB 程度のメモリを使用するアプリケーションに MVEE を適用した場合ではそれほど大きな問題にはならない．しかしながら，memcached のようなひとつのサービスで数 GiB のメモリを利用する状況では，たった一つのサービスとそのレプリカで，ほとんどの物理メモリを消費してしまう場合も考えられる．例えば，memcached で 10GiB を消費する状況を考えると，1 つのレプリカを生成したバリアント全体では 20GiB ものメモリを消費してしまうことになる．これは memcached に限らず，実際に Amazon DynamoDB Accelerator (DAX) [14] を提供する Amazon Elastic Compute Cloud (Amazon EC2) [24] のインスタンスでは 488GiB まで利用可能としている．物理メモリの逼迫によってスラッシングが発生などすると，MVEE の監視下にない他のプロセスに対しても性能に悪影響を及ぼす恐れがある．

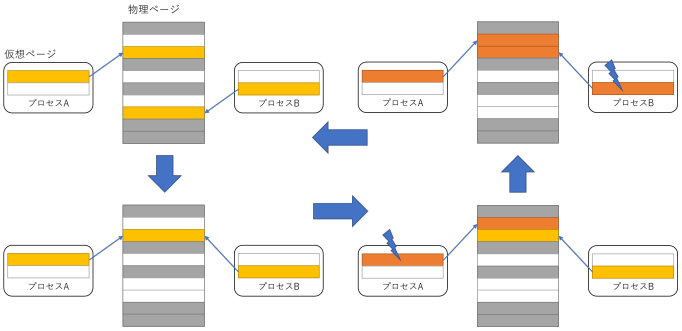

図 2 ページの併合・分裂サイクル

3. 提案

本研究では，メモリを大量に扱うアプリケーションを対象とした MVEE を提案する．提案手法では，こうしたアプリケーションを MVEE 上で効率的に実行する．提案手法は以下のデザインゴールに基づいている．

(1) 物理メモリ使用量を抑える．
MVEE を利用している状況においても資源の消費を抑制し，他のプロセスへの影響を極力避ける．

(2) ユーザアプリケーションのソースコードを改変しない．
利用者がソースコードに手を入れることなく提案手法を利用できることは，適用のハードルを下げることに貢献する．

(3) Multi-thread アプリケーションをサポートする．
近年のアプリケーションではワーカを分散させて，比較的空きのあるワーカに処理を割り振ることが行われている．そのため，複数スレッドに対してもモニタリングを行える必要がある．

MVEE では，全てのバリアントには等しい入力が与えられるため，ページ内部に保持されるデータもまた等しい．併合を行わない場合は，それぞれのバリアントで個々の物理メモリを確保し利用している．したがって，内容が重複するページが複数存在することになる．本手法では，これら内容が等しいページを全バリアントを通して一つにまとめる．ページの併合・分裂サイクルを図 2 に示す．併合先となる物理ページを内容が一致する物理ページの中から一つ選び，全てのバリアントの仮想ページからの参照をこれに向ける．結果として，アプリケーションを単体で動作させた状態と同等程度までメモリ消費の削減が期待できる．

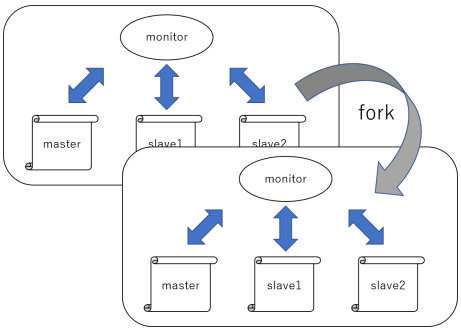

図 3　fork 時のモニタとバリアント

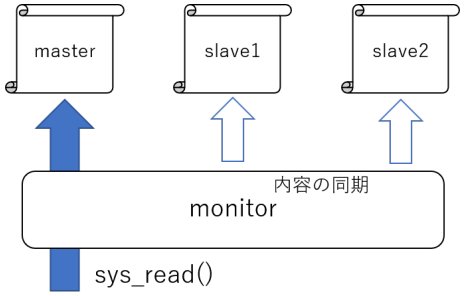

図 4　read の同期

あるバリアントがこの併合したページに対して書き込みを行おうとした際，直接ページに書き込みが行えてしまうと他のバリアントでは不整合が生じる可能性がある．このような場合には再び新たな物理ページを用意し，参照を指し直すことで分裂させる．

提案手法を実現するためには，定期的に全バリアントの仮想ページを走査する必要がある．しかしながら，MVEE のモニタがこれを行うのはコストが大きく，バリアントの動作速度を著しく低下させる恐れがある．また，MVEE に単純にページの併合を適用しようとしても高い併合率は見込めない．なぜなら，ページ内部にメモリアドレスを含む場合は，その値は ASLR によってバリアント同士で一致しないためである．本研究ではこれらの課題に対処できるように機構を設計し，コストの低いページ併合とバリアント同士の高いページ併合率を実現する．

4. 設計

4.1 MVEE モニタ

4.1.1 モニタ・バリアントの生成

MVEE のモニタを生成する際には，ユーザプログラムが自身を監視対象に追加するためのシステムコールを発行する．このとき，カーネルはモニタを新たなカーネルスレッドとして生成し，対象プロセス固有の管理構造にモニタとして記録する．その後にプロセスの複製によりバリアントの生成が行われるが，このとき管理構造も同時に複製されるため，モニタの情報が引き継がれる．

今回実装した MVEE は，モニタ，マスターバリアント，1 個以上のスレーブバリアントで 1 組の監視構成である．構成を図 3 に示す．1 つのモニタは 1 組のバリアントのみを監視し，他のバリアントの組の動作には関知しない．監視対象のバリアントが sys_fork(), sys_clone() した場合には再びモニタを生成する．バリアントそれぞれの子プロセスの管理構造に，生成したモニタを記録することで新たな 1 組の監視構成を成す．

4.1.2 システムコールの監視

モニタはリングバッファを用いてアプリケーションが発行したシステムコールを管理する．リングバッファには発行したシステムコールの番号および引数が保存される．モニタはマスターおよびスレーブバリアントの全てがシステムコールを発行したことを確認すると，リングバッファに記録された情報を精査して実行の可否を決定する．

モニタの動作は，バリアントが発行しようとしているシステムコールによって異なる．システムコール番号が一致していることは必須であるが，引数についてはアドレスが含まれる場合があるため，単純に値のみを比較することはできない．実行することでプロセス内部にのみ変更が加わるシステムコールであれば，バリアント全てでそのまま実行することは何ら問題ではない．しかしながら，I/O のような外部に対して影響を及ぼすシステムコールでは，それぞれが実行してしまうと不整合が生じる恐れがある．これを考慮して，マスターバリアントのみ実行を許可し，スレーブバリアントの実行は不許可にする必要が生じる場合がある．

4.1.3 外部との通信

バリアントが外部と情報をやり取りするためにはモニタの介入が必要となる．sys_read() によってデータを受け取る際には，一旦マスターバリアントのみがシステムコールを実行してメモリ内にデータを格納する．スレーブバリアントに対しては図 4 のように戻り値および内容を同期し，あたかもシステムコールが実行されたかのように見せかける．

sys_write() によってデータを受け渡す際には，内容が完全に一致しているかどうかを検証してから実行を行う．read と同様に，この場合でも実際にシステムコールの実行を行うのはマスターバリアントのみであり，スレーブバリアントに対しては戻り値を同期する．ここで出力するメモリの内容が完全に一致しているかどうかを検証することにより，センシティブな情報であるメモリアドレスを漏洩させることはできなくなる．ASLR によってページ単位でランダムに配置が行われているため，アドレスが含まれる箇所を write しようとした際に内容が一致しなくなるためである．

4.1.4 マルチスレッドアプリケーションへの対応

マルチスレッドアプリケーションに対して MVEE を適用することは容易ではない．バリアントごとでスレッドの

スケジューリング順序が異なると，それに伴ってシステムコールや共有メモリへのアクセスなどの順番が変化し，動作が多様化してしまう恐れがある．動作の多様化を検知したモニタは，これが攻撃によるものなのか否か区別することができず，誤検知に繋がる．

本研究では，マスターバリアントのロックの取得順をスレーブバリアントに対しても強制することでこれを解決する．通常，pthread のスレッド同士は pthread_mutex_lock(), や pthread_mutex_trylock() を呼び出すことでロックを取得する．しかしながら，ロックの競合が起こるなどしてユーザ空間内でロックの取得が完了しなかった場合，sys_futex() システムコールを用いることで仲裁をカーネルに依頼することができる．

マスターバリアントにおいて sys_futex() が発行された場合は，スレッドが取得したロックとその順序をモニタで記録することができる．スレーブにおいても同様に sys_futex() が発行されると，記録したデータを基にしてマスターバリアントのスレッドとスレーブバリアントのスレッドの実行順が同等になるように制御を行うことが可能になる．

しかしながら，ユーザ空間においてロックの取得が完了してしまう場合には sys_futex() が発行されず，モニタがロックの取得処理に介入する余地はない．すなわちスレッド動作順序の多様化による攻撃の誤検知に繋がる．この問題に対応するため，pthread_mutex_lock() および pthread_mutex_trylock() が呼び出された際には必ず sys_futex() を呼び出すように変更を加える．

4.2 同一内容ページの併合

ページ内容の比較及び併合処理は，モニターやバリアントとは独立したカーネルスレッドで行う．併合の処理では各プロセスが利用している仮想アドレス空間を走査し，内容が一致するページを見つけ出す必要がある．そのため，アプリケーションや MVEE モニタが逐次的に行おうとすると，その間に本来の処理を行うことができず，スループットの低下を招く．別スレッドに処理を委ねることで，MVEE の動作に影響を及ぼすことを避けることができる．

ページの併合は，複数の異なる仮想ページのページテーブルから，図 5 のように同一の物理ページを指すように変更することで実現する．ページテーブルから外された物理ページは参照されないため，解放することができる．しかし，同一内容のページを併合したとしても，各バリアントは再び該当ページに書き込みを行う場合がある．その際に，直接ページに書き込みが行えてしまうと他のバリアントでは不整合が生じる可能性がある．そこで，書き込み時には Copy-on-Write [25] によって再び別の物理ページが設けられ，分裂するようにする．それ以降は別に設けたページを利用することで，他のバリアントには影響を及ぼさずに既に併合していたページに対して書き込みを行うことができる．

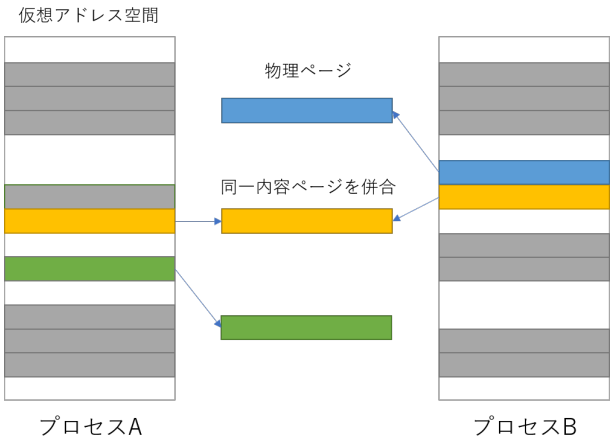

図 5 プロセス間でのページの併合

書き込み頻度の高いページを併合することはコストが大きい．先述の仕組みにより，併合と分裂を頻繁に繰り返すことはシステム全体の性能劣化を招く恐れがある．内容の一致する全ページを併合するのではなく，書き込み頻度が低く安定しているページのみを対象とするべきである．

4.3 ポインタのハッシュ化

単純なデータのみを含むページだけではなく，ポインタを含むデータ構造が存在するページに対しても併合は行うことができることが好ましい．ページ内にアプリケーションのデータのみが含まれる場合はバリアント間でページ内容は完全に一致する．しかしながら内部にポインタを含む場合は，そのポインタは ASLR によってランダムに配置されたアドレスであるため，バリアント間で一致する可能性は非常に低い．この場合，単純にページを併合することはできない．これらのページに対しても併合が行えるように，ポインタの表現に手を加えることを考える．

本研究ではポインタをハッシュ値として扱うことで，メモリアドレスを含むページに対してもバリアント間で併合を可能にする．このポインタのハッシュ値を，以後ハッシュポインタと呼称する．

4.3.1 ハッシュ化とアドレスの解決

ハッシュポインタは，アドレスと一対一で対応する一意な値であるが可逆である必要はない．生成したハッシュポインタと実際の仮想アドレスを対応付けるテーブルを保持することで，ハッシュポインタを利用してメモリにアクセスする際に解決を行う．

アドレスをハッシュに変換する作業，およびその解決はカーネル内で行う．プログラムはメモリアドレスをメモリに書き出す際に，生のアドレスを引数にとって変換用のシステムコール sys_hashptr_gen() を発行する．依頼を受けたカーネルは所定の手順に従ってアドレスをハッシュ値へと変換し，テーブルに対応付けを追加する．ハッシュ値へ

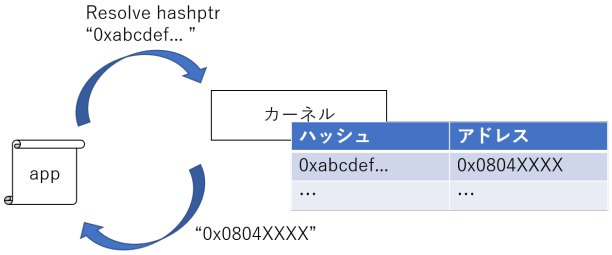

図 6　ハッシュポインタの解決

の変換には，プロセス固有のランダム値で排他的論理和を取るなどの処理を挟むため，復元は容易ではない．変換したハッシュ値を受け取ったプログラムは，これをポインタとして扱いメモリに記録する．ハッシュポインタから仮想アドレスを解決する場合も同様である．図 6 に示すように，プログラムは解決したいハッシュポインタを引数に解決用のシステムコール sys_hashptr_res() を発行し，カーネルはハッシュポインタテーブルを走査する．対応付けられた仮想アドレスが存在する場合はこれを返し，プログラムはメモリにアクセスすることができる．

この機構に対応させるために，プログラムのコードには手を加える必要はない．ソースコードが存在する場合，LLVM [23] によってコンパイル時にポインタへのアクセスを捕捉することが可能である．メモリアドレスを書き出す直前には変換用の，読みだした直後には解決用のシステムコールを発行する関数を挿入する．そうすることで，プログラムはメモリのハッシュポインタを読み書きする以外では，これを利用していることを意識せずに動作することができる．

ソースコードが存在しないバイナリに対しても，ポインタのハッシュ化機構は一部対応可能である．全ての anonymous page を対象とすることはできないが，ヒープ領域に対しては malloc() および free() 等のヒープチャンクを操作する関数をフックすることで対応ができる．memcached [13] のような In-memory DB などは，ヒープを用いてデータを格納することが多い．ユーザプログラムがポインタを含んだデータ構造を内部に持たせなかったとしても，ヒープにはチャンクのフリーリストを構成するためのリスト構造が保持されている．これもまた，ページの併合を阻害する要因の一つである．そこで，glibc [26] に実装されているヒープの操作関数は用いずに，同等の機能を実装したコードを LLVM でコンパイルした共有ライブラリを利用する．これにより，リスト構造のポインタに対してもハッシュ化することができる．

バリアント間でページを併合するためには，生成されるハッシュポインタが全バリアントを通して一致している必要がある．マスターバリアントのアドレスを基に生成したハッシュポインタを，スレーブバリアントに対しても配布することでこれを実現する．

4.3.2　ハッシュポインタテーブルの管理

ハッシュポインタテーブルは task_struct 構造体から参照することで，プロセスごとに紐づけて保持する．プロセスが sys_fork() した際には，子プロセスにおいてもこれまで変換したハッシュポインタを解決できる必要があるため，親プロセスが保持しているハッシュテーブルを参照できるようにする．一旦はテーブルを共有していても問題はないが，親または子プロセスが新たなハッシュを生成した際に共有している同じテーブルにそれを登録することは好ましくない．したがって，このような場合にはテーブルを丸ごと複製し，新たなテーブルを task_struct 構造体に登録して利用するようにする．

sys_execve() によって仮想アドレス空間が一新される際には，以前のハッシュと仮想アドレスの対応は不要となる．そのため，sys_execve() 時にはこれまで利用していたテーブルのエントリは全て削除する．

4.3.3　ASLR との共存

今回提案するハッシュによるポインタの表現では ASLR の利点を損なう恐れがある．攻撃者が Exploit を作成する際，漏洩したメモリアドレスを利用することがある．MVEE では単一のバリアントを狙った Exploit を送ったとしても他のバリアントでは正常に動作しないため攻撃を検知することができる．しかしながらハッシュポインタを用いた場合，バリアント間ではハッシュの値は同期しているため，Exploit に含まれたハッシュポインタを自動で解決し，攻撃が成功してしまう可能性がある．

攻撃者にハッシュポインタを利用させないためには，ハッシュの値そのものを漏洩させないことと推測させないことが重要である．MVEE では，4.1.3 小節で述べたように write 時に内容を比較しているが，ハッシュ値は一致しているためこのままでは容易に漏洩してしまう．そこで，出力しようとしている領域を 8byte 単位でアライメントしたうえで，その範囲にハッシュポインタが含まれるかどうかを確認する．ハッシュポインタが存在している場合は不正な動作として実行を停止する．しかしながら，この方法でハッシュ値の漏洩を完全に防ぐことはできない．glibc に実装されている printf 系関数の，その使用方法の誤りに起因する Format String Bugs (FSB) [27] と呼ばれるバグが存在する．このバグを利用した書式指定文字列攻撃 [28] など，システムコールを直接用いずに値を別の形式に変換して出力することでハッシュ値を漏洩させることも考えられる．

だが，漏洩させたハッシュ値から，攻撃者が必要とするハッシュ値を生成することは非常に難解である．4.3.1 小節に示したように，このハッシュ値はメモリアドレスそのものではなく，処理を施した値を基に計算している．メモリアドレス自体とそれを 2byte シフトした値，さらにランダム値の排他的論理和を取ったものである．総当たりで計算

することでハッシュ値からこの元の値を復元することは原理的には可能であるが，64bit 空間でこれを試行するのは現実的ではない．仮に元の値が復元できたとしても，ASLR によってメモリアドレスもページサイズ単位でランダムとなっているため任意のオフセットを加えたメモリアドレスのハッシュを求めることは非常に難しい．それに加えて，ハッシュポインタは既に変換したアドレスとそのハッシュの組み合わせしか解決できないという性質もある．以上の事から，ASLR とハッシュポインタを組み合わせることで，攻撃者が任意のアドレスのハッシュを生成して攻撃を成功させる可能性は非常に低く抑えることができるといえる．

5. 実装

本章では．提案機構の実装について述べる．対象とする OS カーネルは Linux 4.13.9 とする．OS カーネルには MVEE のモニタと，ハッシュポインタの生成および解決の機能を実装し，またページの併合を行う．アプリケーションがハッシュポインタを利用するようにするため LLVM の Pass を作成し，ソースコードに手を加えることなくバイナリに変更を加える．また，本提案手法をアプリケーションが容易に利用できるようにするため，API を作成しライブラリを提供する．

5.1 Linux カーネルへの実装

本節では，MVEE のモニタとハッシュポインタを実現するために OS カーネルに対して行った実装について述べる．ページの併合については，現在 Kernel Samepage Merging (KSM) [29] の機構を利用している．

5.1.1 モニタとバリアントの管理構造

各バリアントは，同一の mvees_monitor 構造体を共有し，その構造体へのポインタを task_struct 構造体内に保持する．mvees_monitor 構造体には，5.1.2 小節で後述するリングバッファや，モニタそのものの task_struct 構造体への参照が含まれる．

モニタは，mvees_tasks 構造体を持っている．この構造体には各バリアントの task_struct 構造体への参照が保持される．これによりマスターバリアント，および全てスレーブバリアントが管理されることになる．また，バリアントが保持している mvees_monitor 構造体への参照も持つことでリングバッファへのアクセスが可能になる．図 7 にこの管理構造を示す．

5.1.2 リングバッファを利用したシステムコールの監視

各バリアントが発行するシステムコールのリクエストは，全て syscall_req 構造体のリングバッファに記録される．記録した後，バリアントはスリープし，モニタがチェックするまで待機する．この構造体に記録する内容はシステムコールの番号および引数，またモニタによってシステムコールの実行が許可されたか否かを示すフラグである．

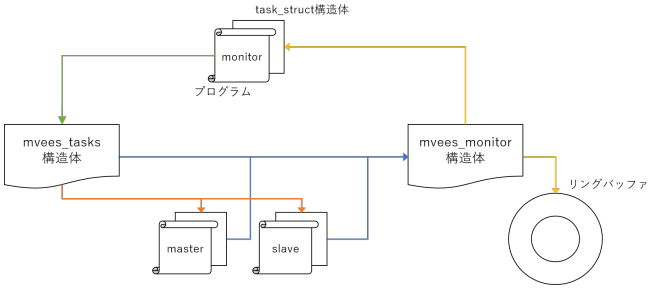

図 7 モニタとバリアントの管理構造

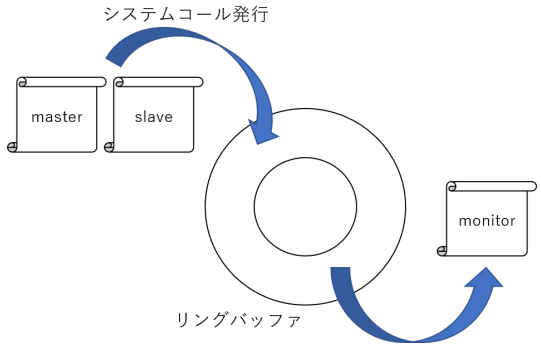

図 8 リングバッファによる管理

図 8 に示すように，モニタはリングバッファに記録されたシステムコールのリクエストを順番に取り出し処理を行う．全バリアントのリクエストが出揃った時点で比較を行う．比較の結果，一致が確認できればシステムコールに応じて syscall_req 構造体のフラグに許可を記録し，バリアントのプロセスを起床させる．システムコールによっては実行の許可ではなくスキップを命じる場合もある．この場合，バリアントはシステムコールの実行を行わずに戻り値同期の処理に飛ぶ．全バリアントに対するチェックが終了したのち，リングバッファが空の間はモニタはスリープし，無駄なリソースの消費を避ける．

システムコールを実行した後の戻り値の比較も同様である．全バリアントは syscall_res 構造体のリングバッファに戻り値を格納し，モニタがチェックを行う．

5.1.3 ロック取得順序の強制によるマルチスレッド対応

マスターバリアントにおける各スレッドのロック取得順序は，lock_record 構造体に記録される．アプリケーションがスレッドを生成する際には sys_clone() が発行されるため，4.1.1 小節で述べたように対応するスレッドごとに一つのモニタが生成される．一つのバリアント集合から生成された全てのモニタが，同一の lock_record 構造体のリングバッファを参照することで協調してロック制御を行うことを可能にする．

各バリアント同士でのスレッドを識別するために，モニタの pid を識別番号として用いる．マスターバリアントの任意のスレッドがロックを獲得すると，そのスレッドのモニターの pid を lock_record 構造体に記録する．スレー

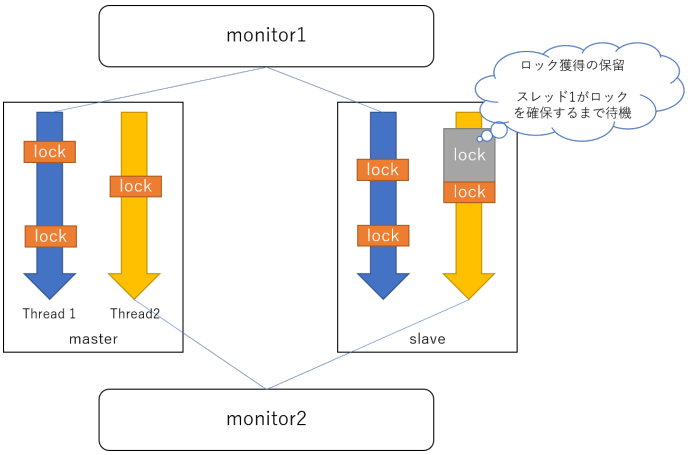

図 9　マルチスレッドのロック確保順序

ブバリアントのスレッドがロック取得を試みた際にそのモニタはリングバッファの先頭を参照し，識別番号が自身の pid と一致している場合はロック獲得を許可し，そうでない場合は 図 9 に示すように該当スレッドがロックを獲得してリングバッファのエントリを消化するまでスレッドを停止させる．そうすることにより，スレーブバリアントのスレッドについてもマスターバリアントとロックの獲得順序を同一にする．

5.1.4　ハッシュポインタの利用と管理

カーネルはアプリケーションに対して表 1 に示すシステムコールを提供する．アプリケーションから sys_hashptr_gen() の発行を受け取ったカーネルは，引数のメモリアドレスからハッシュ値を生成して返す．sys_hashptr_gen() ではその逆で，ハッシュ値から元のメモリアドレスを解決して返す．

ハッシュとメモリアドレスの対応は hashptr 構造体で管理されている．この構造体は二分木構造で接続され，ハッシュとアドレスのいずれからでも該当するエントリにたどり着けるようにしている．ハッシュをキーとした二分木はアドレスの解決に，アドレスをキーとした二分木は生成済みのハッシュを高速に返すために利用される．木の先頭はそれぞれ 256 個用意されており，ハッシュでは最上位バイト，アドレスでは下位 2 バイト目の値で振り分けられる．

ユーザプログラムがハッシュポインタの生成をカーネルに依頼した際，カーネルはアドレスをキーとして構築されている二分木を辿ってエントリを探す．該当エントリが存在している場合は即座に記録されているハッシュを返す．存在しない場合は新たに生成したハッシュ値とアドレスを格納したエントリを生成し，ハッシュとアドレスそれぞれ

表 1　システムコール一覧

syscall	機能
sys_hashptr_gen	ハッシュポインタの生成
sys_hashptr_res	メモリアドレスの解決
sys_write_nohash	内容からハッシュを除いた write

```
void main(int argc, char *argv[], char *envp[]){
        if(argc<3) return;
        mvees(atoi(argv[1]));
        execve(argv[2], &argv[2], envp);
}
```

図 10　簡易ランチャープログラム

の二分木につなげる．初めからアドレスの木を辿らずに，ハッシュ値の木のみを辿りエントリの有無確認および繋げる方が動作としては高速だが，アドレスによる二分木を扱っているのはハッシュの衝突を考慮したためである．

5.2　ユーザアプリケーションの対応

本章では，ユーザアプリケーションを本提案手法に対応させるために実装を行った機構について述べる．

5.2.1　MVEE の利用

アプリケーションのバリアントを作成し MVEE の監視モニタの配下に置くためには，4.1.1 で述べたように sys_mvees() システムコールを発行する必要がある．アプリケーションは作成したいスレーブバリアントの数を引数にとって，sys_mvees() を呼び出す．したがって，アプリケーションを監視対象とするためには起動した時点でこのシステムコールを呼び出すようにコードに変更を加える必要がある．しかしながら，それではソースコードが入手できないバイナリに対して適用が難しい．

既存のプログラムを監視する場合には，ランチャーとなるプログラムを介せばよい．図 10 に示すように，sys_mvees() を発行して自身を監視対象としてから sys_execve() によって既存のプログラムを起動する．バリアントとモニタの参照関係は task_struct 構造体に記録されているため，execve によって別のプログラムを起動したとしても MVEE の動作に必要な情報は引き継がれる．したがって，そのままモニタリングを継続することが可能となる．

5.2.2　ハッシュポインタの利用

アプリケーションのソースコードに変更を加えることなく提案手法を利用するため，LLVM を用いてコンパイルを行う．この際，ポインタに対するアクセスを検出し，指定する関数を呼び出す Pass を作成し用いる．ポインタ変数へ値を代入する処理では，その直前に Addr2Hash() 関数を呼び出し，その戻り値を変数へ格納する．ポインタ変数から値を取り出す際には，レジスタへ値を移動した直後に Hash2Addr() 関数を呼び出し，以後解決したアドレスを利用する．

5.1.4 小節で示したシステムコールを利用するため，ライブラリで API を提供する．先述の Addr2Hash() と Hash2Addr() 関数は，それぞれ sys_hashptr_gen() と sys_hashptr_res() を発行する関数である．

ヒープチャンクのフリーリスト構造もハッシュポインタに対応させるため，glibc のヒープの実装を模倣した独自の

ライブラリを作成した．このプログラムも上記同様の Pass を用いて LLVM でコンパイルすることでこれを実現する．既存のプログラムを動作させる際には，LD_PRELOAD で指定することで本ライブラリのヒープ操作系関数を利用するようになる．

6. 実験

6.1 実験環境

本提案手法の評価を，表 2 に示すコンピュータで行う．Dell PowerEdge T630 II は 16 コア，125GiB のメモリを搭載している．OS は Ubuntu 16.04.1 LTS ，カーネルは提案手法を実装した Linux 4.13.9 である．

6.2 システムのコスト評価

MVEE を適用した状態でマイクロベンチマーカを実行し，処理にかかる時間を測定する．スレーブバリアントは一つのみとする．MVEE を利用していない通常の状態でもベンチマークを測定し，MVEE によるコストを評価する．

また，ハッシュポインタの利用に対するコスト評価も行う．ハッシュポインタを有効にした条件下でマイクロベンチマーカを実行し，処理に要する時間を測定する．

6.2.1 実験方法

マイクロベンチマークプログラムで，メモリに対する読み書きのスループットを計測する．確保するメモリサイズは 1GiB とし，各ページに対してシーケンシャルにアクセスを行う．これを 1 セットとし，10 回くりかえして処理に要した時間を測る．

メモリに書き込みアクセスを行う際，数回に一度システムコールを発行する．MVEE の同期ポイントの出現頻度を変化させ，コストの増加の程度を確認する．本実験では brk() システムコールを計測のために発行する．発行頻度は，メモリアクセス 10 万回に対して 0 回，1 回，5 回，10 回の 4 通りとする．これは，1GiB 分のページにアクセスする間にそれぞれ 0 回，2.6 回，13 回，26 回程度システムコールを発行することになる．ハッシュポインタの利用によるコスト評価では，システムコールは発行しない．

比較対象として，MVEE を用いないネイティブ環境での測定も行う．

6.2.2 実験結果

マイクロベンチマークを実行した際の，スループット結果を図 11 に示す．メモリにアクセスを行っている際にシステムコールの発行を行っていない場合では，MVEE モニタを利用した場合でもネイティブと比べて 4% ほどスループットが低下している．10 万回のアクセスのうち 1 回システムコールを発行した際には，40% ほどの遅延が発生している．5 回，10 回発行した場合にはそれぞれ 77%, 88% ほど遅延している．同期ポイントが増えるにつれて，スループットが低下が非常に大きくなる傾向にある．このような結果になったのは，同期の度に全バリアントが一旦停止するためであると考えられる．

表 2 実験環境

機器名	Dell PowerEdge T630
CPU	Intel(R) Xeon(R) CPU E5-2623 v4 @ 2.60GHz
キャッシュ	10240KB/コア
コア数	16
RAM	125GiB

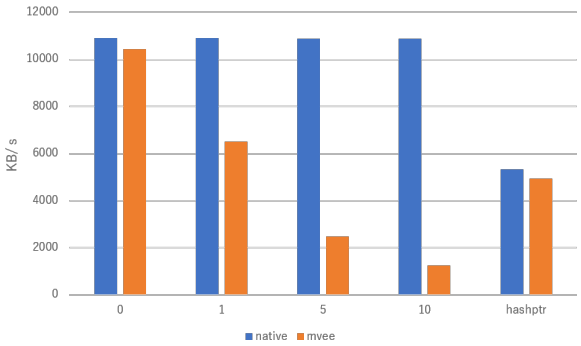

図 11 MVEE モニタとハッシュポインタのスループット

ハッシュポインタを利用した場合では，MVEE を用いない場合でも 50% 程度スループットが低下している．一つのページへのアクセスに対して，複数回アドレスのハッシュ化と解決が行われている．そのため，その度に発行されている sys_hashptr_gen() と sys_hashptr_res() が大きなボトルネックとなっている．また，同一アドレスのハッシュ化や解決が何度も行われる傾向にあるため，キャッシュなどの利用による変換の効率化が求められる．

6.3 実メモリ使用量評価

MVEE を適用した状態でマイクロベンチマーカを実行し，消費される物理メモリ量を測定する．ハッシュポインタを利用しない条件下でも同様に測定を行い，提案手法による物理メモリ使用量の削減を確認する．

6.3.1 実験方法

6.2 節で用いたものと同じベンチマークプログラムを用いて実験を行う．スレーブバリアントの数は一つとする．

本実験で確保するメモリサイズは 4GiB とする．始めに全ページを初期化し，あらかじめ必要な物理ページが全て確保された状態から測定を開始する．ページの初期化に用いるデータには，ポインタが存在するデータ構造を想定してアドレスを含むようにする．実験ではこれらのページに対して 10 セットだけシーケンシャルに書き込みを繰り返す．ただし，1 セットごとに 10 秒の休止を挟む．

また，本実験では偏りのある書き込みのワークロードを想定する．全体の 1%, 10%, 50% の領域に対してのみ書き込みを繰り返し，他の領域の内容に変更は加わえない．こ

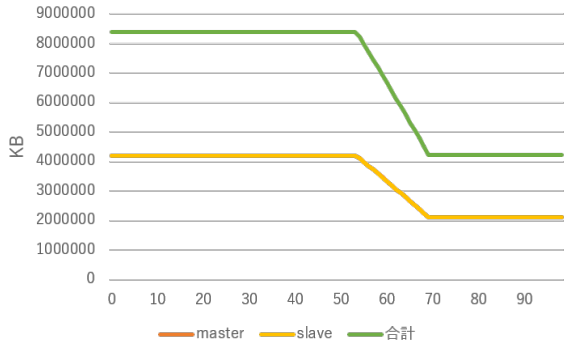

図 12　1%に偏ったワークロードでの物理メモリ使用量

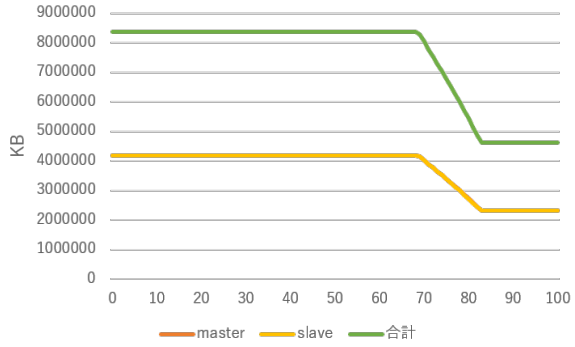

図 13　10%に偏ったワークロードでの物理メモリ使用量

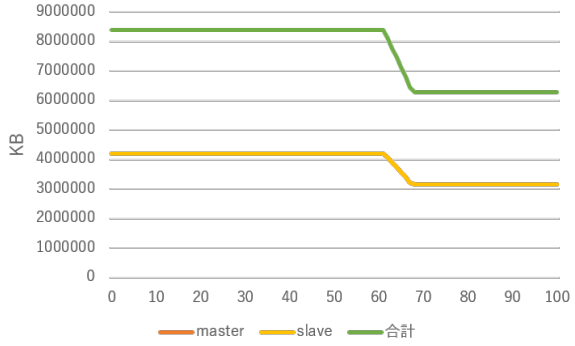

図 14　50%に偏ったワークロードでの物理メモリ使用量

のとき，各バリアントにおける物理メモリ使用量を procfs の smaps から 1 秒ごとに読み取る．

6.3.2　実験結果

ハッシュポインタを有効にした条件の下でマイクロベンチマークを実行した際の，物理メモリの使用量の遷移を図 12, 13, 14 に示す．マスターバリアントとスレーブバリアントの物理メモリ使用量は，どの条件下においても常にほぼ等しい．スレーブバリアント数は 1 つとしているため，計測開始直後の合計の物理メモリ使用量は，単体で動作させた場合の 2 倍である 8GiB となっている．

開始から 1 分前後の時点で KSM がページの併合を行っている．頻繁に書き込みが行われるページは併合の対象外となるため，50%に書き込みが偏ったワークロードでは 4GiB の半分が重複し，物理メモリ使用量の合計は 6GiB

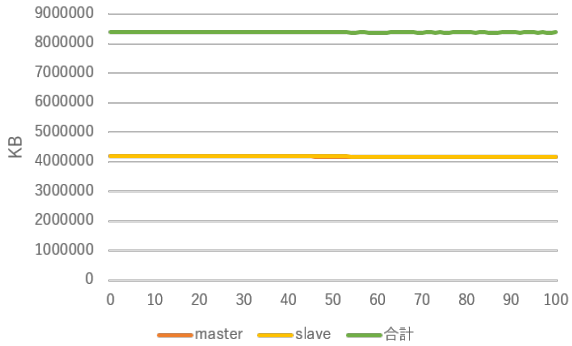

図 15　ハッシュポインタ無効時の物理メモリ使用量

程度となっている．それに対して，1%に偏ったワークロードでは，ほとんどアプリケーションを単体で動作させた場合と同等程度まで使用量の合計が減少している．

なお，ハッシュポインタを無効とした状態で書き込みが 1%に偏ったワークロードをかけた際の，物理メモリの使用量は図 15 に示すとおりである．書き込みが行われるページが非常に偏っていたとしても，ページにはメモリアドレスが含まれるためページが内容が一致せず，併合することはできない．

以上より，提案手法を MVEE に適用することで，ワークロードによっては最大でアプリケーションを単体で動作させた場合と同等程度まで物理メモリの使用量を削減することができることが確認された．

7.　おわりに

本研究では，大量にメモリを消費するアプリケーションに対して MVEE を適用する際に，同一内容のページを併合することで物理メモリの消費を低減する手法を提案した．しかしながら，ASLR によってプロセスの仮想アドレス空間はランダムに配置されているため，そのままではメモリアドレスが一致せずページを併合することはできない．そこで，アドレスをハッシュ値を用いて扱うようにし，バリアント同士でこの値が一致するようにしたことでページの併合を可能とした．

提案手法をカーネルに組み込み，またハッシュポインタを扱うように LLVM を用いてアプリケーションをコンパイルした．これらに対して性能を評価し，手法を適用していない場合との比較を行った．その結果，提案手法を適用していない場合ではバリアントの数だけ物理メモリが消費されているのに対して，適用した場合では最大でアプリケーション単体が消費するのと同等まで使用量を削減することができた．ただし，削減できる量はワークロードに大きく依存し，書き込みを行うページに偏りが少ない場合ではこれほどの使用量削減は期待しづらい．

今後の課題として，MVEE モニタと連携したページの併合機構の実装が挙げられる．現在，ページの併合は KSM

の実装を利用しているため，モニタとは独立してページのスキャンを行っている．この状態でも十分な物理メモリ使用量の削減は行えるが，併合後に再度 CoW によって分割されたページの再併合が行われないなどの不都合が生じるためである．また，より効率的な MVEE の監視機構の実装が必要である．今回行った MVEE のモニタによるコスト評価から，システムコールの発行が多いほどオーバーヘッドが非常に大きくなることが判明した．これは，システムコールを発行する度に全バリアントを同期させ，揃うまで停止をさせているためであると考えられる．マスターバリアントのみは入出力以外では停止させず後々比較を行うことで，スループットの低下を抑えた監視が期待できる．

参考文献

[1] Eklektix Inc. Kernel address space layout randomization [LWN.net]. https://lwn.net/Articles/569635/.

[2] Crispin Cowan, Calton Pu, Dave Maier, Heather Hintony, Jonathan Walpole, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. *Science*, p. 5, 1998.

[3] Stijn Volckaert, Bjorn De Sutter, Tim De Baets, and Koen De Bosschere. GHUMVEE-Efficient, Effective And FlexibleReplication. *International Symposium on Foundations and Practice of Security (FPS '12)*, pp. 261–277, 2012.

[4] Koen Koning, Herbert Bos, and Cristiano Giuffrida. Secure and efficient multi-variant execution using hardware-assisted process virtualization. *Proceedings - 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '16)*, No. Mvx, pp. 431–442, 2016.

[5] Babak Salamat, Todd Jackson, Andreas Gal, and Michael Franz. 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)*, p. 14, 2009.

[6] Stijn Volckaert, Bart Coppens, Bjorn De Sutter, and Michael Franz. Taming Parallelism in a Multi-Variant Execution Environment. *Proceedings of the Twelfth European Conference on Computer Systems (EuroSys '17)*, pp. 270–285, 2017.

[7] Babak Salamat, Andreas Gal, and Michael Franz. Reverse stack execution in a multi-variant execution environment. *Workshop on Compiler and Architectural Techniques for Application Reliability and Security*, pp. 1–7, 2008.

[8] Babak Salamat, Andreas Gal, Jackson Karthikeyan Todd Manivannan, Gregor Wagner, and Michael Franz. Multi-variant program execution: Using multi-core systems to defuse buffer-overflow vulnerabilities. *Proceedings of 2nd International Conference on Complex, Intelligent and Software Intensive Systems (CISIS '08)*, pp. 843–848, 2008.

[9] Stijn Volckaert, Bart Coppens, Alexios Voulimeneas, Andrei Homescu, Per Larsen, Bjorn De Sutter, and Michael Franz. Secure and Efficient Application Monitoring and Replication. *2016 USENIX Annual Technical Conference (USENIX ATC '16)*, pp. 167–179, 2016.

[10] Stijn Volckaert, Bart Coppens, and Bjorn De Sutter-member. Cloning your gadgets: Complete ROP attack immunity with multi-variant execution. *IEEE Transactions on Dependable and Secure Computing*, Vol. 13, No. 4, pp. 437–450, 2016.

[11] Meng Xu, Kangjie Lu, Taesoo Kim, and Wenke Lee. Bunshin: Compositing Security Mechanisms through Diversification. *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*, 2017.

[12] P Hosek and C Cadar. 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)*, pp. 339–353, 2015.

[13] Dormando. memcached - a distributed memory object caching system. http://www.memcached.org/.

[14] Amazon Web Services Inc. Amazon DynamoDB Accelerator (DAX). https://aws.amazon.com/dynamodb/dax/.

[15] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, and Tielei Wang. Preventing Use-after-free with Dangling Pointers Nullification. *Proceedings of the 2015 Network and Distributed System Security Symposium (NDSS '15)*, pp. 8–11, 2015.

[16] Crispin Cowan, Perry Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. *Foundations of Intrusion Tolerant Systems (OASIS '03)*, pp. 227–237, 2003.

[17] GNU Project. GCC, the GNU Compiler Collection. https://gcc.gnu.org/.

[18] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity-Principles, Implementations, and Applications. *Proceedings of the 12th ACM conference on Computer and communications security (CCS '05)*, p. 340, 2005.

[19] Priyam Biswas, Alessandro Di Federico, Scott A Carr, Prabhu Rajasekaran, Stijn Volckaert, Yeoul Na, Michael Franz, and Mathias Payer. Venerable Variadic Vulnerabilities Vanquished. *26th USENIX Security Symposium (USENIX Security '17)*, pp. 186–198, 2017.

[20] Konstantin Serebryany and Derek Bruening. AddressSanitizer: a fast address sanity checker. *Proceedings of the 2012 USENIX Annual Technical Conference (USENIX ATC '12)*, p. 10, 2012.

[21] Konstantin Serebryany. MemorySanitizer : fast detector of uninitialized memory use in C ++.

[22] Adam Belay, Andrea Bittau, and Ali Mashtizadeh. Dune: safe user-level access to privileged cpu features. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation (OSDI '12)*, pp. 335–348, 2012.

[23] llvm-admin team. The LLVM Compiler Infrastructure. https://llvm.org/.

[24] Amazon Web Services Inc. Amazon Elastic Compute Cloud (Amazon EC2). https://aws.amazon.com/ec2/.

[25] Eklektix Inc. User-space page fault handling [LWN.net]. https://lwn.net/Articles/550555/.

[26] GNU Project. The GNU C Library. https://www.gnu.org/software/libc/.

[27] Microsoft Corporation. Format String Bugs - MSDN. https://msdn.microsoft.com/en-us/library/ee823826%28v=cs.20%29.aspx?f=255&MSPPError=-2147217396.

[28] Fatih Kilic, Thomas Kittel, and Claudia Eckert. Blind

format string attacks. *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*, Vol. 153, pp. 301–314, 2015.

[29] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. *Proceedings of the linux symposium*, pp. 19–28, 2009.