プロセスのリソース隔離によるfork爆弾攻撃の防止手法

中川 岳[1,a)]　川田 裕貴[1]　追川 修一[2]

概要：fork 爆弾攻撃は，プロセスを高速かつ大量に生成して，対象システムのリソースを枯渇させるサービス拒否攻撃である．リソース枯渇に対しては，オペレーティングシステムのリソース管理機構によって対策が可能であるが，fork 爆弾攻撃の防止という点では十分ではない．そこでこれまでプロセス生成頻度などに着目して fork 爆弾攻撃を引き起こしているプロセスを検出，停止する方法が検討されてきた．しかしながら，その方法には fork 爆弾ではないプロセスまでも，誤って停止してしまう危険性も伴う．この論文では，fork 爆弾攻撃の防止という観点から，既存のオペレーティングシステムのリソース管理機構と，fork 爆弾攻撃に対処する先行手法の問題点を整理し，それとは異なる新しい fork 爆弾の防止手法を提案する．提案手法は，fork 爆弾の可能性があるプロセス群を発見しても，それらを停止しない．その代わりに，そのプロセス群のリソース利用を他のプロセスと「隔離」して，一定以下に制限する．これにより偽陽性検出したプロセスを誤って停止することを回避できる．Linux を対象に提案手法を実装し，その効果を検証したところ，提案手法により，fork 爆弾攻撃によるリソース枯渇を防止できることがわかった．また，いくつかの想定ケースでオーバーヘッドを計測したところ，いずれも提案手法を実装する前に対して，15%以下のオーバーヘッドで提案手法が実現できることがわかった．実際のアプリケーションを用いた実用的な性能評価では，提案手法の実装前と変わらない性能でワークロードが処理できることもわかった．

キーワード：オペレーティングシステム，メモリ管理，資源管理，マルウェア対策

A fork Bomb Attack Prevention Method with Process Resource Quarantine

1. はじめに

fork 爆弾攻撃は，プロセスを高速かつ大量に生成して，対象システムのリソースを枯渇させるサービス拒否攻撃（Deniel of Service 攻撃: DoS 攻撃）[1] の 1 つである．新たなプロセスが生成されれば，そのプロセスのために新たなリソースの割り当てが起こる．このプロセス生成に伴う資源の割り当てを繰り返すことで，メモリ空間，CPU 時間，プロセス管理テーブルといった，有限リソースの枯渇が起こる．その場合，本来動作すべきプロセスが開始できない，また動作中のプロセスの動作が阻害されるといった問題が起こる．この攻撃に対処するためには，問題のあるプロセスを停止することが必要である．しかしながら，プロセス管理のためのシステムプログラムも，その動作にリソースが必要である．そのため，攻撃が発生しても有効な対策が取れず，コンピュータシステムの再起動を余儀なくされるケースもある．このように，コンピュータシステムのリソースを枯渇させ，システムの正常なサービス提供を阻害する攻撃が，fork 爆弾攻撃である．

fork 爆弾攻撃の経路は多岐にわたる．この攻撃は攻撃対象のシステムで任意のプログラムを実行可能であれば，実施することができる．そのため，攻撃対象システムに一般ユーザとしてアクセス可能であれば攻撃が可能である．また，ネットワークサービスを提供するサーバプログラムに，ネットワーク経由で任意のプログラムの実行が可能になる脆弱性があった場合には，その脆弱性を経由して，fork 爆弾攻撃が可能である．この場合は，攻撃対象のシステムに

[1] 筑波大学大学院 システム情報工学研究科
コンピュータサイエンス専攻
Department of Computer Science, University of Tsukuba
1-1-1 Tennodai, Tsukuba, Ibaraki 305–0006, Japan
[2] 筑波大学 システム情報系 情報工学域
Division of Information Engineering, Faculty of Engineering, Information and Systems
1-1-1 Tennodai, Tsukuba, Ibaraki 305–0006, Japan
[a)] gnakagaw@cs.tsukuba.ac.jp

ⓒ 2015 Information Processing Society of Japan

対するアクセス権限が無い状態で攻撃が可能である．サーバプログラム経由で任意のプログラムが実行可能になる脆弱性は，多数報告されている [2–7]．また存在が公表されていない脆弱性や未修正の脆弱性を利用したゼロデイ攻撃 [8] も存在するため攻撃経路を完全に断つことは難しい．システムに対する攻撃の意図がない場合でも，プログラミングミスにより，大量のプロセス生成とリソースの枯渇が起こる可能性がある [9]．また，利用者とオペレーティングシステムのインタフェースであるシェルプログラムについても，コマンドの組み合わせによっては fork 爆弾攻撃を引き起こすことが知られている [10]．

オペレーティングシステムのリソース管理機構を用いれば，fork 爆弾攻撃のような，リソース枯渇攻撃からシステムを守ることができる．しかしながら，既存のオペレーティングシステムのリソース管理機構は，fork 爆弾による攻撃を防ぐには不十分である．そこで fork 爆弾による攻撃の検出と回復についてはいくつかの提案が行われてきた [9,11]．これらの手法は，コンピュータシステムで動作するプロセスの数や，プロセス生成の頻度に基づき，fork 爆弾攻撃を起こしているプロセス群を検出，停止することで，システムを回復させる．しかしながらこれらの手法には，fork 爆弾攻撃の原因ではないプロセスまでも，fork 爆弾攻撃を引き起こすプロセスとして検出し，誤って停止する問題がある（偽陽性検出問題）．その問題を回避するためには，fork 爆弾検出のための閾値を緩和する必要があるが，それは一方で，真に検出すべき fork 爆弾攻撃を検出できない問題を引き起こす．

この論文では，fork 爆弾攻撃の防止という観点から，既存のオペレーティングシステムのリソース管理機構と，fork 爆弾攻撃に対処する先行手法の問題点を整理し，それとは異なる新しい fork 爆弾の防止手法を提案する．提案手法は，fork 爆弾の可能性があるプロセス群を発見しても，それらを停止しない．その代わりに，そのプロセス群のリソース利用を他のプロセスと区別して，一定以下に制限する．このような，問題のあるプロセスを，他のプロセスから切り離してリソース管理することを，本論文ではプロセスのリソース隔離と呼ぶ．この提案手法では，正常なプロセス群を fork 爆弾を引き起こすプロセスとして検出した場合でも，プロセスの停止は起こらない．偽陽性検出されたとしても，一定の制限下で，プロセスは自由に活動ができる．そのため，先行手法で問題になっていた，正常なプロセスの停止や，fork 爆弾検出のための閾値を緩和などの問題を解決できる．

Linux を対象に提案手法を実装し，その効果を検証したところ，提案手法により，fork 爆弾攻撃によるリソース枯渇を防止できることがわかった．また，いくつかの想定ケースでオーバーヘッドを計測したところ，いずれも提案手法を実装する前に対して，15%以下のオーバーヘッドで提案手法が実現できることがわかった．また実際のアプリケーションを用いた実用的な性能評価では，提案手法の実装前と変わらない性能でワークロードが処理できることもわかった．

Program 1 fork 爆弾攻撃を引き起こすプログラムの例

```
1 #include <unistd.h>
2 int main(void){
3     while(1){
4         fork();
5     }
6 }
```

本論文の構成は以下の通りである．第 2 節では，本論文が対象とする fork 爆弾攻撃について述べ，既存の fork 爆弾攻撃への対策手法とその問題点を整理する．第 3 節では本論文が提案するプロセスのリソース隔離について述べる．第 4 節では，提案手法の評価実験について述べる．第 5 節では，まとめの今後の課題について述べる．

2. 研究の背景

この節では，まず本研究の背景である fork 爆弾攻撃について述べ，既存の対策手法とその問題点について議論する．

2.1 fork 爆弾攻撃とその攻撃経路

fork 爆弾攻撃は，プロセスを高速かつ大量に生成して，対象システムのリソースを枯渇させるサービス拒否攻撃（Deniel of Service 攻撃: DoS 攻撃）[1] の 1 つである．

Program 1 は fork 爆弾攻撃を引き起こすプログラムの例である．ここでは while (true){} で示されるブロックは無限ループを表し，fork() は子プロセスを生成する手続きの呼び出しであるとする．このプログラムが動作すると，繰り返しごとに子プロセスが生成される．その生成された子プロセスは，同様に子プロセスを生成する．このように再帰的に子プロセスを生成が繰り返されると，結果として短時間に大量のプロセスが生成される．この高速かつ大量のプロセス生成はメモリ空間，CPU 資源，プロセス管理情報テーブルといった様々なリソースの枯渇を引き起こす．このリソースの枯渇が発生すると，そのシステムの安定性は損なわれ，そのシステムが提供すべきサービスが阻害される．

fork 爆弾攻撃は攻撃対象のシステムで任意のプログラムを実行可能であれば，特権権限なしに容易に実施することができる．プロセスの生成は，一般的に，管理者特権のない一般ユーザにも許可されている操作である．そのため，攻撃者が対象システムに一般ユーザとしてアクセス可能であれば，fork 爆弾による攻撃が実施可能である．また，ネットワークサービスを提供するサーバプログラムにネットワーク経由任意のプログラムを実行する脆弱性があっ

ⓒ 2015 Information Processing Society of Japan

た場合には，その脆弱性を経由して，fork 爆弾による攻撃を実施することができる．この場合は，攻撃対象のシステムに対するアクセス権限が無い状態で攻撃が可能である．サーバプログラム経由で任意のプログラムが実行可能になる脆弱性は，多数報告されている [2–7]．また存在が公表されていない脆弱性や未修正の脆弱性を利用したゼロデイ攻撃 [8] も存在するため攻撃経路を完全に断つことは難しい．

システムに対する攻撃の意図がない場合でも，システムの一般利用者が結果的に fork 爆弾攻撃を引き起こす可能性もある．利用者によるプログラミングの過程で，子プロセスを生成することは珍しくない．この子プロセスを生成するプログラムに，意図せずプロセス生成を繰り返す瑕疵が含まれていれば，そのプログラムは fork 爆弾攻撃を同じ状況を作り出す．また，利用者とオペレーティングシステムのインタフェースであるシェルプログラムについても，コマンドの組み合わせによっては fork 爆弾攻撃を引き起こすことが知られている [10]．問題となるプロセス生成は，プログラミングの学習過程でも起こりうる．先行研究では，大学教育でのプログラミング実習において，受講者が fork 爆弾攻撃を意図せずに起こすことが報告されている [9]．これら例のように，システムを攻撃する意図がない場合でも，ユーザのプログラミングミスなどにより，結果的に fork 爆弾攻撃が発生することがある．

2.2 既存の対策手法とその問題点

オペレーティングシステムのリソース制限機構は fork 爆弾による攻撃を防ぐには不十分である．オペレーティングシステムは，一般に，プロセスやユーザ単位で利用可能なリソースを制限する機構を備えている．この制限機構により，ユーザが同時に実行できるプロセスの数を制限することができる．fork 爆弾攻撃によりシステムの不安定化は大量のプロセス生成によって引き起こされるため，この動作可能なプロセス数の制限により，ある程度防止することができる．しかしながら，fork 爆弾攻撃の内容によっては，動作可能なプロセス数の制限だけでは不十分なこともある．攻撃を引き起こすプログラムがプロセスの生成と同時に，ある程度のメモリを確保するようにプログラミングされていれば，少数のプロセス生成でもシステムを不安定化することができる．例えば，ある攻撃プログラムが，メモリを 100MiB 確保するプロセスの生成を繰り返すとする．この攻撃プログラムが 9 個のプロセスを生成した時点で，約 1GiB のメモリを占有することができる．この占有を防ぐには，ユーザが実行できるプロセス数を 10 個以下に制限する必要がある．しかしながら，この制限はシステムの実運用を考慮すると現実的でない．この例のように，実行可能なプロセス数の制限だけでは，fork 爆弾攻撃を十分に防ぐことは難しい．

前述したとおり，オペレーティングシステムに備わっているリソース制限機構は，fork 爆弾攻撃を防ぐには不十分である．そこで fork 爆弾による攻撃の検出と回復についてはいくつかの提案が行われてきた [9, 11]．これらの手法により，fork 爆弾攻撃に対処することができる．しかしながらこれらの手法には，実際には fork 爆弾攻撃の原因ではないプロセスを誤って停止する問題がある．

これらの手法はプロセス生成の頻度や，ユーザが実行するプロセスの数に基づいて fork 爆弾攻撃を検出する．この検出したプロセス群を停止することで，システムの不安定化の防止や，あるいは不安定化したシステムを回復させることができる．しかしながら，システムによっては，意図的に短時間に大量のプロセス生成が発生することもある．例えば，http サーバのプログラムが大量のリクエストに対応するために，短時間に大量の子プロセスを生成することがある．この子プロセスの大量生成はサービス提供のための正常な動作である．しかしながら，プロセス生成の頻度やプロセス数に基づいて，プロセスを停止する方式では，この正当なプロセス生成までもが fork 爆弾攻撃であると判定され，サーバプロセスが停止される可能性がある．もちろん，fork 爆弾攻撃と判定するためのパラメータを緩和することで，正当な，短時間での大量の子プロセス生成を許容し，このような偽陽性判定を回避することも可能である．しかしながらこの場合，その緩和した制限の範囲で，停止すべき fork 爆弾攻撃を引き起こすプロセスの動作も許容することになる．前述したとおり，少ない子プロセス数でリソースの枯渇を起こすことも可能であり，このような制限の緩和は偽陰性判定を招く．

以上のように，fork 爆弾による攻撃の発生を検出し，その原因となるプロセスを停止するアプローチには，偽陽性検出により，本来のサービスの動作を阻害する問題がある．偽陽性検出を回避するために，fork 爆弾検出のためのパラメータを調整することは，偽陰性検出により fork 爆弾による攻撃を検出できない問題を引き起こす．

3. 提案手法

本論文では，Linux を対象に，fork 爆弾を引き起こす可能性のあるプロセスのリソース隔離を提案する．提案手法は，従来手法と同様に，プロセス生成の頻度に基づいて，fork 爆弾による攻撃の予兆を検出する．提案手法では，検出したプロセス群をその場で停止しない．その代わりに，プロセス群が使えるメインメモリを制限することで，プロセス動作の継続を許す．先行手法と異なり，プロセスを停止しないので，検出が偽陽性であった場合の影響は小さい．この節ではその提案手法について述べる．

提案手法の処理の流れは，大まかに次の通りである．まず，提案手法が有効な Linux カーネルでは，fork, vfork, clone, execve といった，システムコールの呼び出しを監視し，その頻度を算出する．もし，システムにおけるプロセ

ⓒ 2015 Information Processing Society of Japan

ス生成の頻度が閾値を超えた場合は，システムに fork 爆弾攻撃の予兆があるものと判断し，その原因となっているプロセス群に対して，メインメモリ利用の制限を課す．制限対象となったプロセスは，その制限の範囲内で，動作を継続することができる．以下ではそれぞれの処理について詳しく述べる．

3.1 プロセス生成速度の測定

プロセスの生成は，fork, vfork, clone, execve といったシステムコールを経由して行う．提案手法では，ユーザプログラムから発行されるこれらのシステムコールをフックして，正規のシステムコールに処理を追加する．これはオペレーティングシステムが内部に持つ，システムコールハンドラテーブルを変更することによって実現する．

対象システムコールをフックした際には，そのシステムコールを発行したプロセス，そのプロセスが所属するプロセスグループ，ナノ秒単位のシステム時間を取得し，記録する．プロセス生成の記録は時系列順に並べて管理されており，この情報を用いる事で，どのような頻度でプロセス生成が行われているのかを把握することができる．通常のシステムコール処理を遅延させないために，通常は最低限の記録処理のみを行う．

一定のプロセス生成を記録すると，その記録を分析して，直近のプロセス生成の頻度を算出する．この一定のプロセス数は，パラメータ FORK_RECORD_DEPTH で指定する．このパラメータが 10 であれば，システムで起こる 10 回のプロセス生成ごとに，プロセス生成の頻度の計算を行う．

プロセス生成の頻度の計算は, FORK_RECORD_DEPTH の値を記録の先頭と最後尾の発生時間の差で除することによって求める．頻度の算出が終われば，記録していたデータは破棄され，また次のプロセス生成の記録データが記録される．

3.2 プロセスのリソース隔離

算出したプロセスの生成速度が一定の閾値を超えていた場合は，その原因となっているプロセスと，そのプロセスが所属するプロセスグループに属する全てのプロセスについてメインメモリの利用量制限を実施する．この時の閾値を FORK_SPEED_LIMIT とする．

このプロセス群のリソース制限には，cgroup を用いる．cgroup は Linux が実装する，プロセスのリソース管理機構である．cgroup では，オペレーティングシステムで動作するユーザータスクを cgroup 単位で管理する．この cgroup には，ある特定のリソースに関して，同じパラメータで管理されるタスクが登録される．この cgroup はそれぞれ管理対象のリソースごとに準備されたサブシステムに所属しており，そのサブシステムの機能により，リソースの制限が提供される．cgroup の管理は cgroup file system を介して行われる．つまり，パラメータや所属するタスクについては，cgroup file system 上のファイルを読み書きすることで設定を行う．

cgroup には，memory subsystem が実装されている．この memory subsystem を利用することで，プロセスが利用可能なメモリに関する制限を行うことができる．具体的には，使用可能なメインメモリの量，スワップ領域を含めたプロセスが利用可能なメモリの量などが挙げられる．例えばあるプロセス A について，そのメインメモリ利用量を 512MB に制限する際には，利用可能なメインメモリを 512MB に制限した memory subsystem に所属する cgroup を生成し，この cgroup にプロセス A を参加させれば，制限をかけることができる．この cgroup には，複数のプロセスを参加させることができ，提案手法ははこの機能を利用して，プロセスのリソース隔離を実現する．なお，このときのメインメモリの制限のパラメータを MEMORY_CAP と呼ぶ．

表 1 実験環境

ホスト計算機環境	
CPU	Intel Core i7-4960X (6 core, 12 thread)
RAM	30.0GB
Operating System	Linux 4.0.5
VMM	QEMU (KVM Support enable)
仮想計算機環境	
CPU	仮想 4 コア
RAM	1.0GB
Operating System	Linux 3.10.0
Swap	10.0GB

表 2 実験パラメータ

パラメータ	設定値
FORK_RECORD_DEPTH	100 record
FORK_SPEED_LIMIT	1500 fork/sec
MEMORY_CAP	379 MiB

4. 評価実験

本論文の提案手法は，fork 爆弾攻撃を引き起こすプロセス群を検出し，そのプロセス群を停止することなく，リソース利用を制限することで fork 爆弾攻撃によるシステムの不安定化を防止する．その効果とオーバーヘッドを検証するために，3 つの実験を行った．この節ではその実験の概要と実験結果について述べる．実験は仮想マシン上に構築した環境を用いて行った．表 1 に実験環境の概要を示す．また提案手法に設定したパラメータを表 2 に示す．

4.1 提案手法の効果の確認

提案手法により，fork 爆弾攻撃の検出とリソース制限が実現できているか確認するために，実際に fork 爆弾攻撃に

ⓒ 2015 Information Processing Society of Japan

よるリソース枯渇を引き起こすプログラムを実行し，システム状況の変化を観察した．

実験に用いたプログラムのソースコードを Program 2 に示す．実験プログラムは C 言語によって記述し，コンパイラ，リンカによって実行形式に変換して実行した．このプログラムのうち，fork() はプロセスを生成する手続きである．memset() は任意のメモリ領域を，指定されたデータを指定のサイズで埋める手続きである．usleep() はマイクロ秒単位でプログラムの実行を中断する手続きである．malloc() は任意のサイズのメモリ領域を動的に確保する手続きである．

このプログラムは，子プロセスの生成を 1 ミリ秒ごとに繰り返す．生成された子プロセスは生成された直後に，プロセス空間の特定の箇所にデータ 0x64（16 進数）を 512KiB 書き込む．Linux における子プロセス生成時のメモリコピーは Copy on Write 方式で行われ，子プロセス生成時には親プロセスのメモリ内容のコピーは行われない．子プロセスがメモリ内容を更新するときまで，コピーは遅延される．この memset() によるメモリの書き込みによって，子プロセスのために親プロセスから独立した領域が確保される．つまり，このプログラムによって生成された子プロセスは，それぞれ最低でも 512KiB のメモリリソースを消費する．この書き込みにより，生成された子プロセスは多量のメモリ領域を消費する．

Program 2 を提案手法を無効にした状態で実行した．そのときのシステム全体の利用可能なメインメモリの量（空きメインメモリ）の変化を図 1 に示す．計測は 1 秒おきに行った．Program 2 の実行は，空きメインメモリの測定を開始してから，3 秒後に開始した．グラフからは，その 3 秒後ごろから空きメインメモリが急激に減少し，その後ずっと低いままであったことを示している．分析の結果，利用可能なメインメモリ量の最小値は 60.0MiB であった．このようなメモリリソースの不足はシステムの不安定化の原因となる．

Program 2 を提案手法を有効にした状態で実行し，同様に空きメインメモリを測定した．図 2 にその変化を示す．この場合でも測定を開始して 3 秒後から，空きメインメモリが大きく低下している．分析の結果，空きメインメモリは 74.9MiB まで低下していたことがわかった．しかしながら，7 秒経過後から一転増加し，9 秒目以降は約 300MB で安定している．これは提案手法により Program 2 とその子プロセスが利用できるメインメモリに制限がかけられているためである．提案手法を無効にした時（図 1）に比べて，空きメインメモリを一定量，確保することができていることが読み取れる．

4.2 システムコール処理についてのオーバーヘッド

提案手法ではプロセス生成を検出するために，fork, vfork,

Program 2 実験プログラム

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <stdlib.h>
4
5  int main(void){
6      void *mem = malloc((1 << 10)*512);
7      int i, pid;
8      while(1){
9          pid = fork();
10         if(pid == 0){
11             memset(mem, 'd', (1 << 10)*512);
12         }
13         usleep(1 * 1000);
14     }
15 }
```

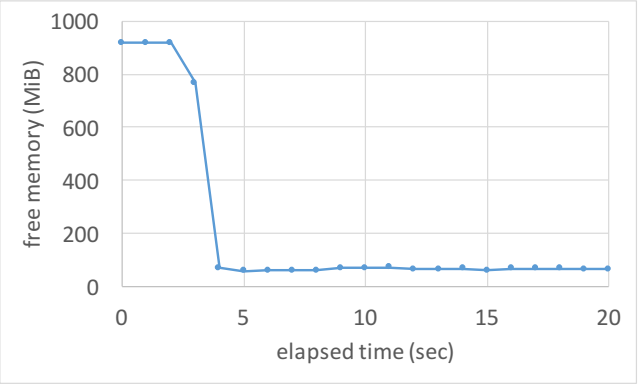

図 1 システムの空きメモリ量の変化（提案手法無効）

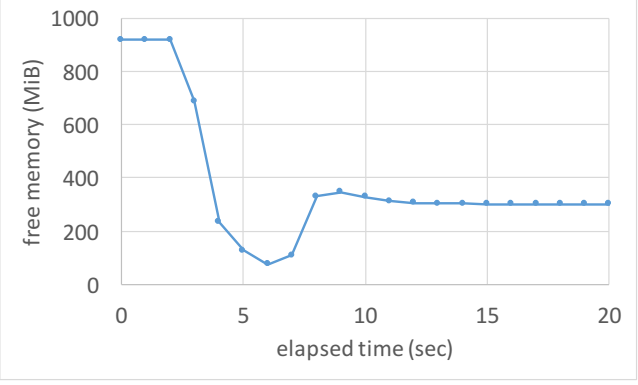

図 2 システムの空きメモリ量の変化（提案手法有効）

ⓒ 2015 Information Processing Society of Japan

clone, exec の 4 つのシステムコールをフックする．そのため，提案手法によりそれらのシステムコール処理にオーバーヘッドが加わる．この実験ではそのオーバーヘッドが実用上の問題となるか検証するために，システムコール処理時間の測定を行った．

システムコール処理の測定は Linux で標準 C ライブラリとして利用されている，glibc に含まれる fork() 関数の処理時間を測定することで実現した．提案手法がフックするシステムコールは 4 種類あるが，ユーザプログラムがこれらのシステムコールを直接呼び出すことは稀であること，Linux においてはライブラリ関数の fork, vfork は clone システムコールを呼び出すこと，いずれのシステムコールのフックについても同じ処理を行うこと，以上の理由から，ライブラリ関数 fork() の処理時間を測定，比較することでオーバーヘッドを評価した．fork() の呼び出しから，結果の返却までを記録し，それを 999 回測定した．

提案手法はプロセス生成にかかわるシステムコールを記録するため，その頻度によってオーバーヘッドが変化する可能性がある．そこで，システムコールを繰り返す頻度を変化させた 3 つの条件で測定を行った．この頻度の変化は，計測の繰り返しの合間に，適宜，間隔を開けることで実現した．特に条件 C は提案手法によりリソース制限が実施されることを意図している．このそれぞれの実験条件下で，提案手法の有効，無効を切り替えてライブラリ関数 fork() の処理時間を測定した．

- 条件 A: 1 秒間に 100 回のプロセス生成を行う条件
- 条件 B: 1 秒間に 1000 回のプロセス生成を行う条件
- 条件 C: 1 秒間に 1800 回のプロセス生成を行う条件

図 3 から図 5 に，それぞれの条件でのシステムコールの処理時間を示す．このグラフは，999 回計測したシステムコールの実行時間を，左から順番に縦方向にプロットしたものである．青のマーカーが提案手法が無効であるとき，オレンジのマーカーが提案手法が有効であるときのシステムコールの処理時間を示している．図から読み取れるように，ほとんどの場合で提案手法を有効にしても，処理時間は大きく変化していないことがわかる．しかしながら，定期的に約 400 マイクロ秒のオーバーヘッドが起こっていることも観測できた．これは提案手法が，一定回数のプロセス生成が起こった時に，プロセス生成の頻度を計算し，リソース隔離の処理を行っていることが原因だと考えられる．

表 3 から表 5 に，それぞの条件で測定したオーバーヘッドの処理時間を分析した結果を示す．いずれの条件でも，最小値は提案手法の有効，無効で大きな差はない．一方で，最大値については大きな差が確認された．これは図 3 から図 5 から明らかになった，プロセス生成の頻度計算とリソース隔離に伴うオーバーヘッドが原因であると考えられる．それぞれの表の最右列は，提案手法を有効にすることによって，平均処理時間がどの程度増加したかを示している．提案手法を有効にすることによって，最大で 14.6%のオーバーヘッドが発生しているが，この原因となった処理時間の増大は，それぞれの条件で 10 回発生している大きな処理時間の増大である．これは試行全体の 1%程度であり，コンピュータシステムの用途によっては実用上の影響は軽微であると考えられる．

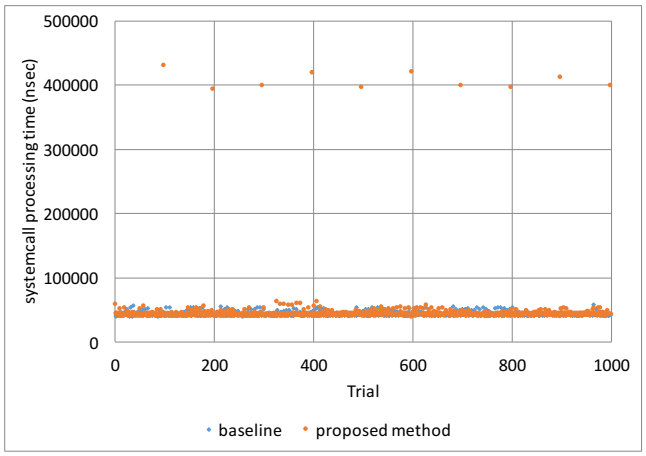

図 3　ライブラリ関数 fork() の処理時間（条件 A）

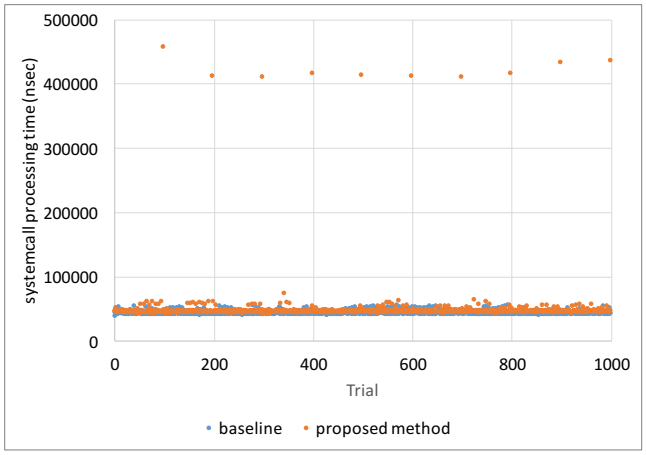

図 4　ライブラリ関数 fork() の処理時間（条件 B）

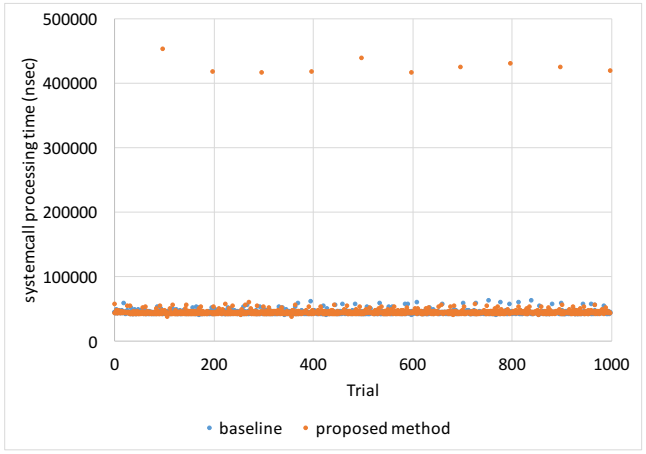

図 5　ライブラリ関数 fork() の処理時間（条件 C）

ⓒ 2015 Information Processing Society of Japan

表 3　条件 A におけるライブラリ関数 fork() の処理時間（単位: ナノ秒）

	最小値	最大値	平均値	平均増加率 (%)
提案手法なし	39486	58289	43914.4	0
提案手法あり	39849	430476	47840.1	8.9

表 4　条件 B におけるライブラリ関数 fork() の処理時間（単位: ナノ秒）

	最小値	最大値	平均値	平均増加率 (%)
提案手法なし	38662	55577	43781.1	0
提案手法あり	40845	457326	50177.4	14.6

表 5　条件 C におけるライブラリ関数 fork() の処理時間（単位: ナノ秒）

	最小値	最大値	平均値	平均増加率 (%)
提案手法なし	39468	62255	43692.1	0
提案手法あり	36668	451330	47642.6	9.0

4.3　アプリケーションへの性能影響

4.2 では，提案手法によって，定期的に 400 マイクロ秒程度の大きなオーバーヘッドが発生することがわかった．しかしながら，そのオーバーヘッドが生じる確率は，全体の 1%程度であることもわかった．このオーバーヘッドが，実際のアプリケーションにどの程度影響を与えるのかを検証するために，プロセスの生成を大量に発生させるワークロードを実行し，その実行時間を，提案手法が有効である場合，無効である場合で比較した．

ワークロードとしては，大規模なソフトウェアである Linux カーネルのビルドを採用した．このようなソフトウェアでは，通常，プログラムのソースコードは複数のファイルに分けて記述されており，それぞれのソースコードをコンパイルしてオブジェクトコードを生成し，それらをリンクするとで実行可能なソフトウェアを構築する．また，必要に応じてライブラリともリンクを行う．Linux 4.2.2 を対象にビルドを最小構成に設定 [*1] し，そのプロセス生成数を計測したところ，95 秒間で 12629 回のプロセス生成が行われていることがわかった．図 6 にビルドプロセスを通した，プロセス生成数の変化を時系列で示す．このビルドプロセスを，提案手法が有効である場合，無効である場合で実行し，処理完了までの経過時間を測定した．

測定結果を表 6 に示す．測定はそれぞれ 10 回行い，その結果を分析した．最小値，最大値，平均値それぞれについて，大きな差は確認できなかった．4.2 では，提案手法による大きなオーバーヘッドが確認されたが，そのオーバーヘッドが生じる確率は非常に小さく，実際のシステム性能への影響は軽微であることが確認できた．

5.　まとめと今後の課題

この論文では，fork 爆弾攻撃によるリソース枯渇から，

[*1] make allnoconfig でオプションを可能な限り無効にしたビルド設定が生成可能である

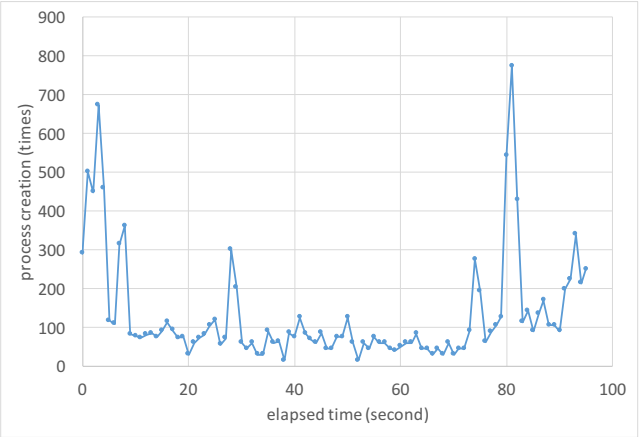

図 6　Linux 4.2.2 のビルドに伴うプロセス生成数の変化

表 6　Linux kernel のビルド処理時間（単位: 秒）

	最小値	最大値	平均
提案手法なし	70.74	72.86	71.036
提案手法あり	70.67	72.75	71.009

コンピュータシステムを保護する手法の提案を行った．これまでの先行研究でも，fork 爆弾攻撃の検出と原因プロセスの停止については提案されてきたが，それらには偽陽性検出により正常なプロセスまでも停止する危険性がある．これに対して，この論文での提案手法は，fork 爆弾攻撃の予兆を捕らえても，原因プロセス群を停止しない．そのかわりに，メインメモリの利用制限を課しながら，プロセスの実行を許容する．この工夫により，偽陽性検出により，無実のプロセスが停止される危険性を低減できる．

提案手法を実装し，評価実験を行ったところ，提案手法が fork 爆弾攻撃によるリソース枯渇を防止できること，一定の間隔で大きなオーバーヘッドが生じるが，実アプリケーションを用いた評価では性能低下が起きないことがわかった．

今後の課題としては，fork 爆弾検出時のリソース制限値を動的に決定すること，また状況の変化に応じて可変させることが挙げられる．現状では，リソース制限値は静的である．システムで利用可能なメインメモリの量は常に変化しており，本来ならば，システムの状況に応じて動的に決定するのが理想的である．そのための手法検討と実証実験が今後の課題である．

参考文献

[1] 寺田真敏：DoS 攻撃：1. DoS/DDoS 攻撃とは，情報処理，Vol. 54, No. 5, pp. 428–435 (2013).
[2] The MITRE Corporation: CVE - CVE-2014-001, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0001.
[3] The MITRE Corporation: CVE - CVE-2014-0088, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0088.
[4] The MITRE Corporation: CVE - CVE-2014-0112, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-

ⓒ 2015 Information Processing Society of Japan

2014-0112.
[5] The MITRE Corporation: CVE - CVE-2014-0226, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0226.
[6] The MITRE Corporation: CVE - CVE-2015-0117, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0117.
[7] The MITRE Corporation: CVE - CVE-2015-1920, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-1920.
[8] Bilge, L. and Dumitras, T.: Before We Knew It: An Empirical Study of Zero-day Attacks in the Real World, *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ACM, pp. 833–844 (online), DOI: 10.1145/2382196.2382284 (2012).
[9] Berlot, M. and Sang, J.: Dealing with Process Overload Attacks in UNIX, *Information Security Journal: A Global Perspective*, Vol. 17, No. 1, pp. 33–44 (online), DOI: 10.1080/19393550801929547 (2008).
[10] Kennel, D. A.: How Attackers Abuse Computing Systems, Technical report, Los Alamos National Lab (2015).
[11] Singh, R.: Fork Bomb Defuser (rexFBD).

ⓒ 2015 Information Processing Society of Japan