Vol.2010-CSEC-50 No.36
2010/7/2

サンプリングベース侵入検知システム

小川 夏樹[†1]　大山 恵弘[†1]

異常ベースの侵入検知システムに関して，今まで多くの研究が発表されている．活発に研究されているシステムの一つは，システムコールの捕捉に基づくものである．それらのシステムでは，システムコール呼び出しにおいて監視対象のアプリケーションプログラムを停止させ，システムコール引数や呼び出しスタックなどの重要なデータを検査する．しかし残念ながら，それらの多くは，オーバヘッドが大きいという問題を共有している．この問題を解決するために，本論文ではサンプリングベース侵入検知システム Sido を提案する．Sido は，従来の検査タイミングの一部においてのみ，侵入を検知するための検査を行う．サンプリングは確かに検知精度を低下させるが，性能と精度への要求の間のトレードオフにおける，適切な場所をユーザに提供することができる．我々は単純なコールスタック検査と VtPath 法を用いたサンプリングベース侵入検知システムを実装した．本論文ではそのシステムの有用性を評価するための予備実験の結果も示す．

A Sampling-Based Intrution Detection System

Natsuki Ogawa[†1] and Yoshihiro Oyama[†1]

A great deal of literature on anomaly-based intrusion detection systems has been published. One class of actively studied systems are based on system call interception. They suspend a monitored application program at system call invocations, and check critical data such as system call arguments and call stacks. Unfortunately, many of them have a common problem of large overheads. To address this problem, we propose Sido, a sampling-based intrusion detection system, that checks for intrusions at a portion of the original timings. Indeed sampling reduces the detection accuracy, but it provides users a appropriate trade-off point between requirements of performance and accuracy. We implemented sampling-based intrusion detection schemes using a simple call stack inspection and the VtPath technique. We show the result of preliminary experiments for evaluating the effectiveness of our system.

1. はじめに

コンピュータシステムへの攻撃に対処するために，侵入検知システムが提案されている．侵入検知システムによって，攻撃に対して早期に対策を講じることができ，被害を少なくすることができる．侵入検知の手法は，シグネチャベース検知と異常ベース検知に分けることができる．シグネチャベース検知は，攻撃の特徴を表現するデータ（シグネチャ）を持ち，ネットワーク上を流れるパケットや OS 上の動作とシグネチャを比較し，シグネチャと同じものが観測されたら攻撃を受けていると判断する．シグネチャベース検知には高い精度で攻撃を検知できるという利点があるが，未知の攻撃に対処することはできないという欠点がある．一方，異常ベース検知は，通常のプログラムの挙動やネットワークの通信情報を保存しておき，それらの特徴と異なる動作を異常とみなす．この方法には，未知の攻撃も検知することができるという利点がある．本研究では異常ベース検知に基づく侵入検知システム（異常検知システム）を扱う．また特に，OS 上でのプログラムの動作を監視するホストベース型異常検知システムを扱う．

これまでに数多くのホストベース型異常検知システムが提案されてきた[1)–6)] が，それらの多くはシステムコールの検査を行うものであるため，実行時オーバーヘッドが大きかった．例えば，Sekar らが提案した FSA ベース侵入検知システム[4)] は，1 回のシステムコール検査に 250ms のオーバーヘッドが，Feng らが提案した VtPath 法[5)] では，1 回のシステムコール検査に 150ms のオーバーヘッドが生じると記されている．文献[6)] のシステムでは，カーネルモジュールを用いてオーバヘッドを大幅に削減しているが，それでも検査によっては 1 検査あたり 5ms を超えるオーバヘッドが加わる．

本研究は，検査を間引くことにより検査のオーバーヘッドを減少させる侵入検知システム（サンプリングベース侵入検知システム）Sido の構築を目的とする．具体的な方法としては，従来行っていた検査のうち一部だけを実行する，すなわちサンプリング検査を行うことで，検査回数を減らす．このようにすると，検査を間引く割合によってオーバーヘッドを変えることができるので，ユーザが許容したオーバーヘッドで侵入検知を実現できるという利点がある．ただし，サンプリング検査の導入により，検知精度は下がる．そこで本研究では，サンプリング検査により，検知精度がどの程度下がるのかを調査する．

†1 電気通信大学
The University of Electro-Communications

Vol.2010-CSEC-50 No.36
2010/7/2

本論文は以下のように構成されている．2 章では，提案手法について述べ，その実装を 3 章で説明する．4 章では実験の結果を示しそれについての議論を行う．5 章では，関連研究について述べる．6 章で本論文をまとめる．

2. 提 案 手 法

ここではまず本システムの概要を説明し，その後侵入検知の手法とサンプリング手法について述べる．

2.1 システム概要

監視対象となるアプリケーションを実行するプロセス（アプリケーションプロセス）を，侵入検知処理を実行するプロセス（モニタプロセス）が監視する．本システムには，正常時の状態を学習する訓練モードと，検知を行う検知モードがある．訓練モードでは，システムコール発行時，あるいは一定時間毎にアプリケーションプロセスを停止させ，アプリケーションプロセスのコールスタックの情報を取得する．訓練モードでの実行終了時に，コールスタックの情報から，正常動作のモデルを作成する（後述）．検知モードでは，同様にシステムコール発行時，あるいは一定時間毎にアプリケーションプロセスを停止させ，コールスタックが正常動作のモデルに合致しているかどうかを検査する．本システムではサンプリング検査を，システムコールの呼び出し時の数回に 1 回や，一定時間毎に 1 回検査を行う．このサンプリング検査を行うことによって，検査時にかかるオーバーヘッドを減らすことができる．

サンプリング検査により，本来なら検知できたはずの攻撃を見逃す可能性がある．検査を行っていない時間のうちに完了する攻撃は，本システムには検知されない．しかし実際の攻撃は，システムコールを多数回発行するようなものや，ある程度長い実行時間がかかるものが多いと考えられる．よって，攻撃コードの処理の量に比べて，ある程度サンプリングの検査間隔が短ければ，検知を回避することは困難だと思われる．

本研究では，検知方式（正常動作モデルの作成方式）に関して 2 種類，サンプリング検査の方式に関して 2 種類を実装した．以降でそれぞれについて述べる．

2.2 検 知 方 式

2.2.1 単純スタック検査法

コールスタックには，関数引数や局所変数の他に，リターンアドレスが積まれている．コールスタックに含まれるリターンアドレスを，スタックのトップから抜き出していき，その列を関数呼び出しの履歴として記録しておく．**図 1** に，関数呼び出し履歴作成の例を示す．

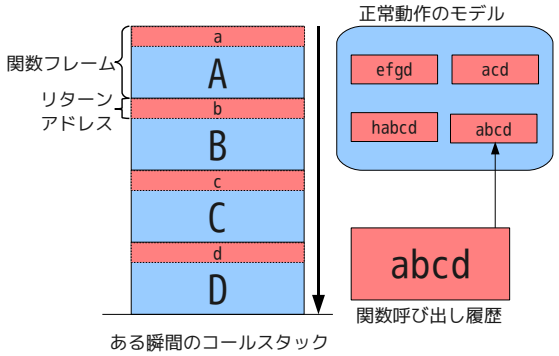

図 1 関数呼び出し履歴の生成
Fig. 1 Extraction of execution context information

ある瞬間のコールスタックが，図のような状態だとすると，関数呼び出し履歴は abcd のようになる．この履歴を，アプリケーションプロセスの正常動作を表すものとして保存しておく．検知モードでは，訓練モードで保存されていない呼び出し履歴が確認された場合，異常とみなす．

2.2.2 VtPath 法

コールスタックの情報を積極的に用いた検知手法である．Feng らにより提案されたこの手法では，2 つの連続したシステムコール呼び出し時のコールスタックの差分を取り出し，VtPath とよばれる仮想的な関数の遷移のパスを生成する．**図 2** は，N 回目のシステムコール呼び出し時と N+1 回目のシステムコール呼び出し時のスタックの状態が示されている．A〜G は関数フレームを表しており，図の上側がスタックトップ，図の下側がスタックボトムである．2 つのスタックから共通している関数フレーム C，D を取り除き，両者の差分として前回のスタックトップから AB を，今回のスタックトップから CFE を取り出し，それらの関数フレームを表現する情報を連結したものを VtPath とする．図 2 の例では，VtPath は abgfe となる．訓練モードでは VtPath を各システムコール呼び出し間で生成し，VtPath の集合を正常動作のモデルとして用いる．検知モードでは、検査の際に作られる VtPath がこの集合に含まれていれば，正常であると判断する．

このモデルを用いると，システムコール呼び出しとその時のコールスタックの状態に応じ

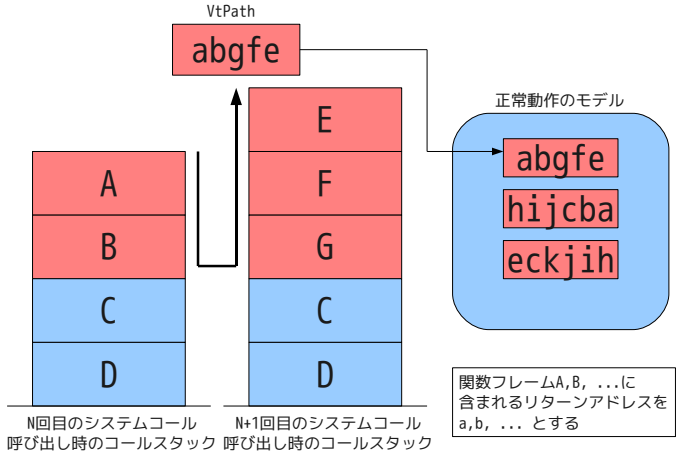

図 2 VtPath の生成
Fig. 2 Extraction of VtPath

た細かな状態分けが可能となる．しかしその反面，システムコール呼び出しの度にコールスタックを辿るため，オーバーヘッドが大きいという問題がある．

2.3 サンプリング方法

2.3.1 システムコールベースサンプリング

システムコールベースとは，システムコール発行時に検査を行う方法をベースとしたサンプリング方法である．単純検査スタック法では，システムコールが発行される数回に 1 回正常動作の収集や検査を行う．VtPath 法では，連続する 2 回のシステムコール発行をサンプリングして正常動作の収集や検査を行う．検査回数の設定は，ユーザが設定できるようになっている．

2.3.2 時間ベースサンプリング

時間ベースサンプリングとは，モニタプロセスがある適当な時間毎にアプリケーションプロセスを停止させ，その間に検査を行う方法である．時間ベースサンプリングでは，停止させるタイミングの決め方が難しい．時間の単位は，限りなく小さくできるからである．訓練モードにおいては，非常に短い間隔，具体的には，1μs に 1 回正常動作を収集する．検知モードでは，それよりも長い時間毎に 1 回アプリケーションプロセスを停止させて検査を行う．どの程度の時間に 1 回検査を行うかは，ユーザが設定することができる．

3. 実　　装

本システムは Linux 上に実装されている．アプリケーションプロセスの監視には，ptrace システムコールを用いている．

3.1 アプリケーションプロセスの監視

システムコールベースサンプリングと，時間ベースサンプリングでは，アプリケーションプロセスの監視方法が異なる．システムコールベースサンプリング時では，ptrace システムコールを用いてアプリケーションが呼び出すシステムコールを捕捉している．モニタプロセス側で ptrace を用いて，システムコール呼び出しごとにアプリケーションプロセスを停止させる．停止中に ptrace を使ってアプリケーションプロセスのコールスタック情報を取得する．

時間ベースサンプリングは以下のように実装する．モニタプロセスは一定時間毎にアプリケーションプロセスを停止させる SIGSTOP シグナルを送信する．停止後にモニタプロセスが ptrace でアプリケーションプロセスにアタッチして，アプリケーションプロセスから関数呼び出し履歴情報を得る．その後，モニタプロセスは ptrace でデタッチして，アプリケーションプロセスを再開させる．一定時間だけアプリケーションプロセスを走らせる処理は，モニタプロセスがその時間を引数とする nanosleep 関数を実行することにより実現している．

3.2 関数呼び出し履歴および VtPath の作成方法

関数呼び出し履歴と VtPath の作成には，ptrace でアプリケーションプロセスから得たコールスタックの情報を用いる．コールスタックには，関数呼び出し時に積むリターンアドレスが含まれている．スタックのトップからボトムまでの各フレームからリターンアドレスを取得し，それらのリターンアドレスを繋げて，関数呼び出し履歴や VtPath を作成する．

3.3 fork および exec への対応

ptrace で監視を行う際の注意点として，fork，exec 系システムコールへの対応がある．

fork システムコールへの対応を説明する．アプリケーションプロセス A が fork システムコールを発行し子プロセス A’ を生成したら，モニタプロセス M も fork して新たなモニタプロセス M’ を生成する．そして，モニタプロセス M’ が監視を開始する．このようにして，新たに生成されたアプリケーションプロセス A’ も監視することができる．

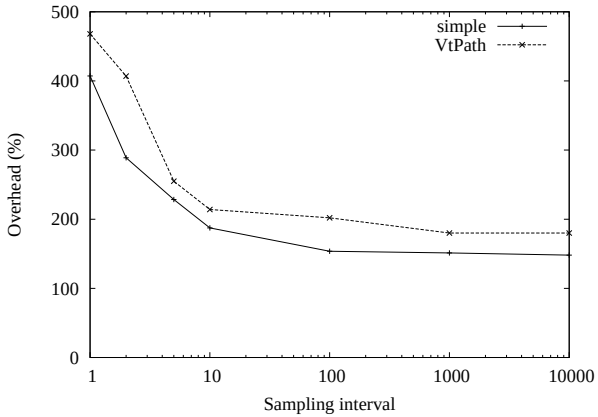

図 3 システムコールベースサンプリング使用時のオーバーヘッド
Fig. 3 Overhead when using system call-based sampling

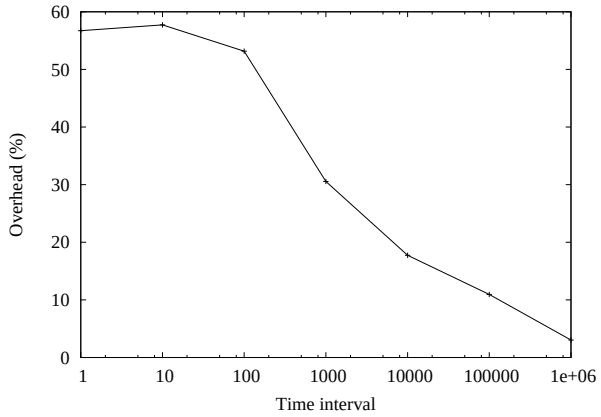

図 4 時間ベースサンプリング使用時のオーバーヘッド
Fig. 4 Overhead when using time-based sampling

また，時間ベースサンプリングでは，システムコールを捕捉していないので，`fork` システムコールの実行を補足することができない．`/proc` の情報から新しい子プロセスの有無を判断する方法を取っている．`ptrace` のオプションを使用して `fork` のみを捕捉するような設定をする方法もあるが，現在は実装されていない．

4. 実　　験

本章では，本システムを用いた実験とその評価について記す．まず，本システムのオーバーヘッドについて述べ，次に検知能力についての実験と議論を行う．実験環境は，OS が Linux Debian lenny，カーネルのバージョンが 2.6.26-2-686，CPU が Intel Core 2 Duo 1.86GHz，メモリが 1GB である．

4.1 オーバーヘッド

本システムによる実行の監視にかかるオーバーヘッドを検証するために，Apache Bench を用いて Web サーバ thttpd の性能を測定した．訓練モードで一定時間実行して正常動作モデルを作成した後，検知モードで thttpd プロセスを監視した．監視をされている thttpd プロセスに対して，`ab` コマンド（Apache Bench）でリクエストを送信した．サンプリングの検査間隔を変化させながらスループットを測定した．

結果を**図 3**，**図 4** に示す．図 3 は，システムコールベースサンプリング使用時のオーバーヘッドを示している．simple は単純スタック検査法での検査を，VtPath は VtPath 法での検査を示している．横軸を表している検査間隔は，例えば 100 では，100 回システムコールが呼び出される度に 1 回検査する．検査間隔が 1 から 10 あたりまでは，検査間隔が大きくなるにつれてオーバーヘッドが減少しているが，それ以降はほぼ横ばいになっている．その原因は，システムコールをフックするオーバーヘッドがボトルネックとなっているからであると考えられる．現在の実装では，`ptrace` を用いているため，検査を行わない場合でも，システムコール呼び出しの度にアプリケーションプロセスは一度停止している．この処理に時間がかかっていると考えられるので，現状ではこれ以上サンプリング間隔を増やしてもオーバーヘッドは減らせないと考えられる．

図 4 は時間ベースサンプリング使用時のオーバーヘッドを示している．VtPath 法の検査に時間ベースサンプリングを適用することは難しいため，単純スタック検査法のみについて実装および実験を行った．そのオーバーヘッドは，システムコールベースサンプリング使用時よりも小さかった．1μs に 1 回検査を行ったときは，オーバヘッドが約 58 %であったが，1000000μs（1 秒）に 1 回検査を行ったときはほとんどオーバーヘッドが無かった．時間ベースサンプリング使用時には，システムコール呼び出しの度にプロセスが停止することがない．そのため，このようなオーバーヘッドの差が生まれたと考えられる．なお，1μs から 100μs までのオーバーヘッドの差が小さいのは `nanosleep` の精度の問題であると推測し

```
void unlink_all_files_in_dir(void){

  // ディレクトリとその中にあるすべてのファイルを消す関数

}

static int recv_and_handle_data(int soc)
{
  int nr;
  char buf[4];   // このバッファが溢れる

  nr = recv(soc, buf, 4096, 0);
  buf[nr] = '\0';
  printf("I received request: %s\n", buf);

  return nr; // ここで書き換えられたリターンアドレスに飛ぶ
}

int main(void){

  ...

  while (1) {
    int soc;

     ...

    recv_and_handle_data(soc);

     ...

  }
}
```

図 5 被攻撃サーバプログラム
Fig. 5 Attacked server program

ているが，現在調査中である．

4.2 攻撃の検知

本システムが攻撃を検知できるかをどうかを調べる実験を行った．

2.1 節での検知回避に関する議論を検証するために，**図 5** のようなサーバプログラムを，本システムで監視した．図 5 のサーバプログラムは，クライアントからのデータを表示するだけの簡単なプログラムである．このサーバプログラムにはバッファオーバーフロー脆弱性が含まれている．このサーバプログラムに対して，クライアントからバッファオーバーフロー攻撃（return-into-libc 攻撃）を行い，`unlink_all_files_in_dir` 関数を呼び出させる．`unlink_all_files_in_dir` 関数は，重要なディレクトリとその中にあるすべてのファイルを消す関数である．訓練モードでは，攻撃を行わずに実行し，本システムを動作させて

表 1 攻撃検知率
Table 1 Rate of the attrack detection

	検査間隔	検知率（%）
システムコールベース	40 回	100
	50 回	50
	60 回	60
	70 回	60
	80 回	30
	90 回	50
	100 回	20
	200 回	10
時間ベース	1μs	10
	10μs	0

正常動作のモデルを作成した．モデル作成の際にはサンプリングを行っていない．その後検知モードで監視を行った．

対象ディレクトリ中のファイル数は 30 個であり，ディレクトリとすべてのファイルを消すまでに発行するシステムコールの数は 35 である．なお，この実験ではディレクトリが残っていれば検知に成功したと判定する．

検知率を**表 1** に示す．検知率は各検査間隔に対して 10 回の試行を行った結果から求めている．まず，システムコールベースサンプリングで検知を行った．検査間隔を 1 に設定したところ，本システムは異常を検知して監視対象プロセスを終了させた．対象ディレクトリは残っていたので，攻撃が完了する前に検知することができたと言える．その後，徐々に検査間隔を大きくしていったところ，40 回に 1 回の検査までは攻撃を 100 %検知して防ぐことができた．40 回以上になると徐々に検知率が下がっていき，200 回に 1 回の検査では検知率は 10 %になった．

次に時間ベースサンプリングで検知を行った．検査間隔を 1μs 設定したところ，本システムは攻撃を 10 %の確率で検知することができた．しかし，それ以上検査間隔を大きくすると異常を検知する前にディレクトリが消されてしまった．この結果から，この攻撃にかかる時間はサンプリング検査間隔より短く，かなり短い時間で検査を行わなければ攻撃を検知できないことが分かった．

なお，Linux における `nanosleep` 関数で 1μs だけプロセスを止めようとしても実際にはプロセスのコンテキストスイッチに時間がかかるため，1μs よりも長時間止まっている可能性が高い．`nanosleep` の精度と実際の検査間隔の関係については今後さらに調査が必要で

ある．

5. 関連研究

サンプリング検査によりシステムのオーバヘッドを削減する研究は，過去に複数存在する．Aiken らの研究[7] では，バグを発見するためにアプリケーションプログラムに埋め込む検査をサンプリングする枠組みが提案されている．異なる場所で動作する同種のプログラムが実行情報を共有することにより，低いオーバヘッドでバグを発見する．本研究はアプリケーションプログラムに埋め込む検査ではなく，侵入検知システムによる検査のサンプリングを対象としている．

Application Community に関する研究[8] では，同種のアプリケーションを監視する複数のセキュリティシステムが分担して検査を行うことにより，検査のオーバヘッドを減らす方式が提案されている．本研究では，その研究とは異なり，単一のセキュリティシステムのみによる監視を想定している．また，オーバヘッドが大幅に削減できるならば，時には攻撃を見逃すことも許容できるという状況を想定している．

文献 9) および文献 10) の研究は，ネットワークパケットをサンプリング検査するネットワークベース侵入検知システムを提案している．本研究はそれらの研究とは異なり，ホストベース侵入検知システムを対象としている．

6. まとめと今後の課題

本論文では，検査間隔を間引くサンプリングベース侵入検知システムを提案した．そのシステムでは，実行時のコールスタックの情報を用いて，正常動作モデルを作成する．さらに，サーバアプリケーションを用いた実験を通じて，オーバーヘッドを減らせることと，サンプリング検査を行った場合でも攻撃を検知できることを示した．

今後の課題について述べる．まず，アプリケーションプロセスから取得できる情報には，コールスタック以外にも，システムコールの種類や変数の値などがあるが，それらを用いた異常検知手法に対するサンプリング検査方式についても今後考える必要がある．また，時間ベースサンプリングにおける，`nanosleep` の精度について，さらに調査する必要がある．

参考文献

1) Forrest,S., Hofmeyr,S., Somayaji,A. and Longstaff,T.: A Sense of Self for Unix Processes, *Proceedings of 1996 IEEE Symposium on Security and Privacy*, pp. 120–128 (1996).
2) Warrender,C., Forrest,S. and Perlmutter,B.: Detecting Intrusions Using System Calls: Alternative Data Models, *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pp. 133–145 (1999).
3) Wagner,D. and Dean,D.: Intrusion Detection via Static Analysis, *Proceedings of 2001 IEEE Symposium on Security and Privacy*, pp. 156–168 (2001).
4) Sekar, R., Bendre, M., Dhurjati, D. and Bollineni, P.: A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors, *Proceedings of 2001 IEEE Symposium on Security and Privacy*, pp. 144–155 (2001).
5) Feng,H., Kolesnikov,M., Fogla,P., Lee,W. and Gong,W.: Anomaly Detection Using Call Stack Information, *Proceedings of the 2003 IEEE Symposium on Security and Privacy*, pp. 62–75 (2003).
6) Mutz,D., Robertson,W., Vigna,G. and Kemmerer,R.: Exploiting Execution Context for the Detection of Anomalous System Calls, *Proceedings of 10th International Symposium on Recent Advances in Intrusion Detection (RAID)*, pp. 1–20 (2007).
7) Liblit,B., Aiken,A., Zheng,A. and Jordan,M.I.: Bug Isolation via Remote Program Sampling, *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 141–154 (2003).
8) Locasto,M.E., Sidiroglou,S. and Keromytis,A.D.: Software Self-Healing Using Collaborative Application Communities, *Proceedings of the 13th Annual Network and Distributed System Security (NDSS)*, pp. 95–106 (2006).
9) Gonzalez,J. and Paxson,V.: Enhancing Network Intrusion Detection with Integrated Sampling and Filtering, *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection*, pp. 272–289 (2006).
10) Ning,Z. and Gong,J.: A Sampling Method for Intrusion Detection System, *Proceeding of the 11th Asia-Pacific Symposium on Network Operations and Management*, pp. 419–428 (2008).