システムソフトウェアと
オペレーティング・システム 90−11
（２００２．６．２７）

カーネルモジュールを用いたSoftwarePotの効率的な実装方式

神田 勝規[†]　大山 恵弘[††]　加藤 和彦[††,†††]

[†] 筑波大学大学院博士課程システム情報工学研究科

[††] 科学技術振興事業団さきがけ研究21

[†††] 筑波大学 電子・情報工学系

概 要

インターネットに代表されるオープンなネットワーク環境で流通するソフトウェアを安全に実行するためのシステムSoftwarePotの研究開発を進めている。SoftwarePotはソフトウェアを仮想的なファイルシステムに閉じこめた状態で実行する。仮想的なファイルシステムはシステムコールの引数の検査と書き換えによって実装されている。本論文では、仮想的なファイルシステムの実装におけるオーバーヘッドを軽減するために、カーネルモジュールを用いた実装方法を提案すると共に、その実装を用いて行なった実験結果を示す。

Efficient Implementation for SoftwarePot using Kernel Modules

Katsunori KANDA[†]　Yoshihiro OYAMA[††]　Kazuhiko KATO[††,†††]

[†] University of Tsukuba Graduate School, Doctoral Program System & Information Engineering

[††] PRESTO, Japan Science and Technology Corporation

[†††] Institute of Information Sciences and Electronics, University of Tsukuba

Abstract

We have been developing the SoftwarePot system, which securely executes the software circulated in an open-network. Software executed in SoftwarePot is enclosed by a virtual file system, which is achieved by intercepting and rewriting arguments of system calls. This paper proposes a scheme to efficiently construct a virtual file system for SoftwarePot and shows the experiment results.

1　はじめに

HTTP、FTP、SMTP等を利用し、多くのソフトウェアが流通するようになった。しかし、電子メールに添付されるウイルスのように、ソフトウェアの中には利用者の意図とは異なる動作をするものがある。こういったソフトウェアの中には、ファイルを不正に改竄したり、機密データを漏洩させるものがある。悪意を持った作者によって作られたこのようなソフトウェアから計算機資源を守るために、資源へのアクセスが制限された実行環境を作る研究がこれまでに行なわれてきた。

我々は、ソフトウェアを流通させる用途に適したサンドボックスシステムSoftwarePot [4, 5, 8] を提案している。SoftwarePotはソフトウェアを仮想的なファイルシステムに閉じ込めて実行することができる。ソフトウェアは明示的な指示が与えられない限り、仮想的なファイルシステム外のファイルにアクセスすることはできない。ソフトウェアパッケージ開発者が作成したパッケージは仮想的なファイルシステムの構築に必要な情報を含むアーカイブファイルとして配布される。パッケージを入手したユーザーは、ファイルへのアクセスや通信といった計算機資源へのアクセスを制限することができる。

文献 [4, 8] で提案されているSoftwarePotは計算機資源へのアクセス制御、仮想的なファイルシステムの構築をシステムコール引数の検査と書き換えによって実現している。システムコールの検査と書き換えは、ユーザーレベルで行なわれる。ユーザーレベルでのシステムコールの検査はシステムコールを1回呼び出すごとに2回のコンテキスト

スイッチが発生するためオーバーヘッドが大きいことが知られている [6]。また、ファイルパス引数の検査と書き換えによるオーバーヘッドがかかることが実験 [8] によってわかっている。

本論文では、カーネルモジュールを用いることにより SoftwarePot によるオーバーヘッドを削減し、ソフトウェアを高速に実行するための方式を提案する。

本論文は以下のように構成される。まず、2 章において SoftwarePot の実装方式と問題点を述べる。3 章において提案方式を述べる。4 章で、提案方式の性能を測定した実験結果を報告する。5 章で関連研究との比較を述べる。6 章でまとめと今後の課題を述べる。

2 SoftwarePot

2.1 概要

SoftwarePot 上で実行されるソフトウェアは、仮想的なファイルシステム (**ポット空間**) の構築に必要なメタデータを含むアーカイブファイル (**ポットファイル**) として配布される。ポットファイルを入手したユーザーはセキュリティポリシーファイルを作成もしくは入手し、ソフトウェアを実行する。セキュリティポリシーファイルには、ソフトウェアが使用できるシステムコールの種類、使用できるネットワークリソース、ファイルマッピングのポリシーが記述される。ファイルマッピングとは、ローカルのファイルシステムの一部をポット空間内へリンクする操作で、ファイルの実体をコピーすることなくポット空間内でローカルのファイルを使用することができる。ファイルマッピングが行なわれる典型的なファイルは共有ライブラリや実行結果を保存するためのディレクトリである。

2.2 現在の実装

図 1 に現在のシステムの全体像を示す。SoftwarePot は、ソフトウェアの発行するシステムコールをソフトウェアとは異なるプロセス (モニタープロセス) で捕捉し、システムコールの種類や引数を検査及び引数を書き換えることによりセキュリティポリシーをソフトウェアに対し強制する。以下において、システムコールを捕捉した時にモニタープロセスが行なう処理について述べる。

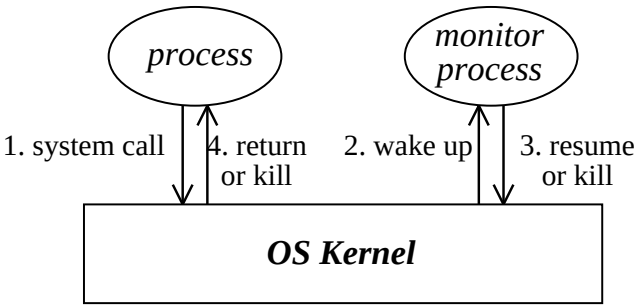

図 1: 現在のシステムの全体像

モニタープロセスはシステムコールを捕捉すると、実行が許可されているかどうかをはじめに検査する。許可されていない場合、エラーコードを返しシステムコールの実行を中断する。システムコールの実行が許可されている場合、実際のシステムコールの処理を行なう。ただし、ファイルパス引数を持つシステムコールの場合、実際の処理を行なう前にファイルパス引数の書き換えを行なう。

ファイルパス引数の書き換えとは、ポット空間内のファイルパスを実際のファイルシステム上のファイルパスに変換する操作であり、この操作によってソフトウェアはポット空間上のファイル以外へアクセスすることができなくなる。

ポット空間内のファイルは、ポットファイルから取り出されたファイルに関連付けられている場合と、実際のファイルシステム上にもともと存在していたファイルに関連付けられている場合がある。ポットファイル内のファイルへのアクセスが発生した場合、アクセスされたファイルの実体がポットファイル内から一時ディレクトリへ取り出される [1]。実際のファイルシステム上にもともと存在していたファイルへのアクセスはファイルパスの書き換えの処理のみで実現できる。

3 提案方式

提案システムの全体像を図 2 に示す。提案システムは以下のモジュールから構成されている。

カーネルレベルリファレンスモニター システムコールをカーネル空間内で捕捉する。捕捉されたシステムコールはその実行が許可されているかどうかを検査されるだけで、現在の方式のようにファイルパス引数の書き換えは行なわ

[1] 事前にすべてファイルを取り出して実行時のオーバーヘッドを減らすこともできる。

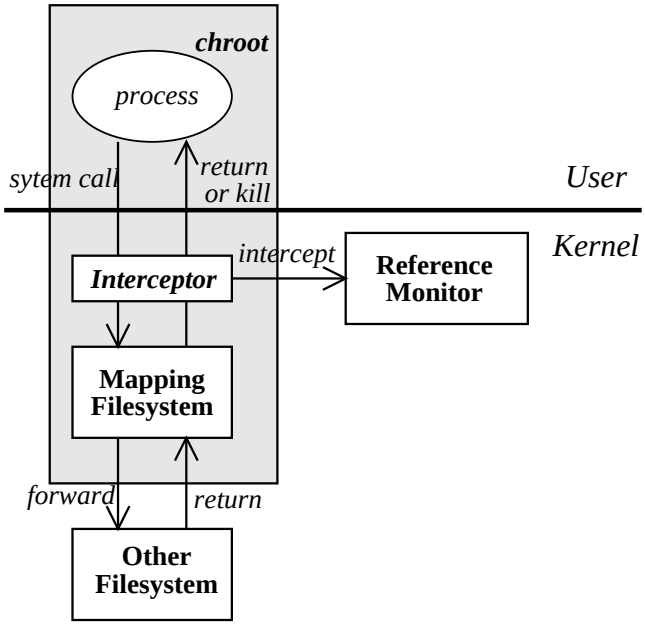

図 2: 提案システムの全体像

ない。

マッピングファイルシステム 複数デバイスをまたがったハードリンク (**クロスデバイスリンク**) を作成するためのファイルシステム。マッピングファイルシステムは、他のファイルシステムと組み合わせて使う必要があり、自身でファイルを保持することはできない。

提案システムにおいて、ポット空間は上記のモジュールと chroot システムコールによって構築される。chroot システムコールとは、プロセスごとにルートディレクトリを設定し、ルートディレクトリ以下のファイル以外にアクセスできないようにするためのシステムコールである。プロセスがファイルパスを引数にとるシステムコールを発行した場合、設定されたルートディレクトリを起点としてファイルの名前解決を行なう。例えば、“/tmpXXX” をルートに設定されたプロセスが “/bin” にアクセスした場合、実際のファイルシステム上のファイルパスでは “/tmpXXX/bin” と表わされるファイルにアクセスすることになる。実際のファイルシステム上の “/bin” にアクセスすることはできない。

ポットファイルは、一時ディレクトリに展開されマッピングファイルシステム上にリンクが張られる。また、セキュリティポリシーファイルによって指定された実際のファイルシステム上のファイルもマッピングファイルシステム上にリンクが張られる。プロセスは chroot システムコールによってルートディレクトリをマッピングファイルシステムのマウントポイントに設定される。この設定により、プロセスはマッピングファイルシステム上のファイル以外にアクセスすることができなくなる。このとき、ソフトウェアはカーネルレベルリファレンスモニター上で実行される。

ポット空間を構築する方法としてカーネルレベルリファレンスモニター内に、ファイルパス引数の書き換えを行なうルーチンを組み込む方法がある。提案システムにおいて、この方式を用いなかった理由は、ファイルパスの書き換えを行なう際のファイルパス解析にかかるオーバーヘッドが生じ、これらのオーバーヘッドは非常に大きいからである。

3.1 コンテキストスイッチによるオーバーヘッドの削減

コンテキストスイッチを発生させずにシステムコールを捕捉する方法として、次の 3 つの方式が考えられる。

1. システムコール引数の検査ルーチンをシステムコールのラッパーライブラリ (いわゆる libc) 上に作成する
2. カーネルを拡張して、複数のプロセスで 1 つのアドレス空間を共有できるようにする
3. OS カーネルに動的ロード可能なカーネルモジュールとしてシステムコール検査ルーチンを作成する

提案方式においては、3 の方式を採用する。

1 の方法は、ラッパーのルーチンを通らずにシステムコールを呼び出すことができるため、サンドボックスの作成には向かない。2 の方法は、既存の OS を修正する必要があるため SoftwarePot のように多くのユーザーに使ってもらうことを目指している場合、ユーザーがシステムを導入するのに障壁を高くしてしまう。3 の方法は、インストールには管理者権限が必要であるがスクリプト等を利用することにより比較的楽にインストールを行なえる。マッピングファイルシステムもまた、カーネルモジュールとして実装されているため実行時に、コンテキストスイッチによるオーバーヘッドが生じることはない。

3.2 クロスデバイスリンク

通常のハードリンクは同一デバイス上のファイルをリンクすることしかできず、デバイスをまたがったリンクを作成したい場合は、シンボリックリンクを使う。chroot によって作られた仮想的なファイルシステム内から外部のファイルへアクセスするためには、ハードリンクのような inode レベルでリンクされている必要がある。シンボリックリンクのように名前レベルでのリンクでは、chroot によって作られた仮想的なファイルシステム内からファイルの実体に到達することができない。例えば、“/mnt/map/myetc” が “/etc” と、リンクされていた時に、chroot システムコールによって “/mnt/map” をプロセスのルートディレクトリに設定する。このとき、chroot されたプロセスが “/myetc” にアクセスしようとすると、chroot によって構築された新しいファイルシステ上には “/etc” というファイルが存在しないため “/etc” にリンクされている “/myetc” へのアクセスは失敗する。

クロスデバイスリンクは、デバイスをまたがって作成できるハードリンクである。したがって、chroot によって作られた仮想的なファイルシステム内からクロスデバイスリンクによってリンクされていファイルの実体にたどりつくことができる。

クロスデバイスリンクの構造を図 3 に示す。クロスデバイスリンクの作成要求が発行されるとディレクトリへのリンクの場合、リンク元のディレクトリを open[2]して inode へのポインタを取得する。また、ディレクトリに含まれるファイルに対してもリンクを作成する。ただし、実行効率を考慮して子ディレクトリ内のファイルに対してリンクを作成するのは、子ディレクトリ以下のファイルへアクセスした時である。open によって取得したリンク元の inode へのポインタ (図 3 中の real inode) は、mapent 構造体に格納される。

クロスデバイスリンクが open されると、file 構造体と real inode が関連付けられる。本来ならばマッピングファイルシステムの inode と関連付けられなければならないのに、このような処理を行なうのは execve システムコールの処理を成功させるためである。execve システムコールは、マッピングファイルシステムのようなブロックデバイスを持たないファイルシステム上のファイルが実行されることを想定して実装されておらず、実行ファイルを open した後に execve 独自の read ルーチンを呼び出す[3]。

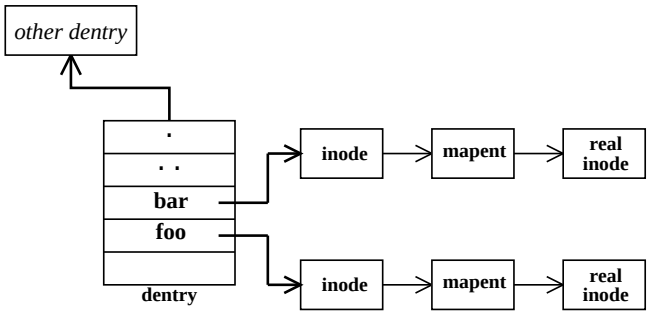

図 3: クロスデバイスリンクの構造

file 構造体と real inode を関連付けると、execve システムコールは正しく動作する。しかし、この方法ではマッピングファイルシステムの inode へアクセスする手段がないため、マッピングファイルシステムからこのファイルへアクセスする方法が失なわれてしまう。このような理由から、本実装では file 構造体の操作関数 (read,write など) を置き換え file 構造体中のプライベートデータ領域に、マッピングファイルシステムの inode を保存した。

file 構造体に登録した操作関数は、real inode の操作関数を呼び出すだけのラッパー関数である。

4 実験

3 章において述べた実現方式に基づいて実装を行ない、実験を行なった。実験の環境は、PentiumIII 550MHz, Linux 2.4.7, Main Memory 256MBytes である。

4.1 リファレンスモニターによるオーバーヘッド

getpid,open の各システムコールを 10000 回呼び出すプログラムを作成し、リファレンスモニターなしの場合、ユーザーモードリファレンスモニター有りの場合、カーネルレベルリファレンスモニター有りの場合の実行時間を計測した。その結果から、システムコール呼び出し 1 回にかかる処理時間を計算した (表 1)。ただし、ユーザーレベルリファレ

[2]厳密には、カーネル内から open システムコールを呼び出すことはできないため、同等の処理を実装した。

[3]ファイルシステムが提供する read ルーチンはユーザー空間へのデータ転送を行なうため、カーネル内で使用することができない。

表 1: システムコール呼び出し 1 回の処理時間 (μs)

	なし	リファレンスモニター	
		ユーザーレベル	カーネルレベル
getpid	0.6	2.7 (+2.1)	1.9 (+1.3)
open	1.5	35.8 (+35.3)	2.8 (+1.3)

表 2: システムコール処理時間の比較 (μs)

	なし	マッピングファイルシステム有り	
		カーネルレベルリファレンスモニター無し	カーネルレベルリファレンスモニター有り
open	1.5	1.8 (+0.3)	3.2 (+1.7)

ンスモニターにおいて、システムコールを捕捉した時に行なう処理はシステムコールの実行継続許可を与える処理だけである。したがって、ユーザーレベルリファレンスモニターでは 2 回のコンテキストスイッチが起きる。

リファレンスモニターによるオーバーヘッドは、リファレンスモニター上でのシステムコールの処理時間からリファレンスモニターなしの場合の処理時間を引いた値である。表中の括弧内の値は、計算によってもとめたオーバーヘッドである。

getpid システムコールをユーザーレベルリファレンスモニター上で呼び出した場合のオーバーヘッドは、2.1μs であった。このオーバーヘッドには、ユーザーレベルリファレンスモニターの処理時間とコンテキストスイッチ 2 回の処理時間が含まれる。getpid システムコールをカーネルレベルリファレンスモニター上で呼び出した場合のオーバーヘッドは 1.3μs であった。このオーバーヘッドには、カーネルレベルリファレンスモニターの処理時間のみが含まれる。以上の結果から、カーネルレベルリファレンスモニター上でシステムコールを呼び出した場合、コンテキストスイッチの処理時間がかからないため、ユーザーレベルリファレンスモニターに比べてオーバーヘッドが小さくなっていることがわかる。

open システムコールをユーザーレベルリファレンスモニター上で呼び出した場合のオーバーヘッドは、35.3μs であった。このオーバーヘッドには、ユーザーレベルリファレンスモニターの処理時間とコンテキストスイッチ 2 回の処理時間に加え、ファイルパス書き換えのための前処理が含まれる。open システムコールをカーネルレベルリファレンスモニター上で呼び出した場合は、オーバーヘッドにはカーネルリファレンスモニターの処理時間のみが含まれる。

4.2 マッピングファイルシステムによるオーバーヘッド

マッピングファイルシステムによって、ファイル操作のためのシステムコールにどの程度のオーバーヘッドがかかるかを計測した。計測方法は、4.1 節と同様の方法を用いて、open システムコールについて計測を行なった。計測結果を表 2 に示す。

マッピングファイルシステム上のファイルに対し open システムコールを呼び出した場合、1.8μs の処理時間がかかり、オーバーヘッドは 0.3μs であった。このオーバーヘッドには、マッピングファイルシステムの open ルーチンの処理時間が含まれる。カーネルレベルリファレンスモニターを通してマッピングファイルシステム上のファイルをアクセスした時の処理時間は、3.2μs であり、オーバーヘッドは 1.7μs である。open システムコールをカーネルレベルリファレンスモニター上で呼び出した場合のオーバーヘッドと、マッピングファイルシステム上のファイルに対し open システムコールを呼び出した場合にかかるオーバーヘッドを足すと 1.6μs となる。これは実験結果と比べて 0.1μs の差があるが計測方法の精度が低いために生じた誤差と考えられる。したがって、カーネルレベルリファレンスモニターとマッピングファイルシステムを同時に用いた場合に新たなオーバーヘッドは生じないと言える。

5 関連研究

セキュリティポリシーに基づいて、プロセスがアクセスできるファイルを制限するシステムとして Janus [3]、SubDomain [1] がある。これらのシステムはファイルへのアクセスコントロールをシステムコール引数を検査することによって行なっている。しかし、提案方式ではファイルシステムによってアクセスコントロールを行なうため、システムコールの引数を検査する場合と比べて、ファイル

パスの解析によるオーバーヘッドが少ない。

OS 機能の拡張コード呼び出し時のコンテキストスイッチによるオーバーヘッドを削減するための方法として、細粒度保護ドメイン [7] がある。この方法では既存の OS に修正を加えているが、提案方式はカーネルモジュールとして実現されているためユーザーが導入しやすいという利点がある。

クロスデバイスリンクの実現方法として、NFS クライアントを拡張しユーザーレベルでファイルシステムを実装する方法 [2] がある。しかし、この方法ではコンテキストスイッチによるオーバーヘッドが生じるため、提案方式と比べてオーバーヘッドが大きい。

6 結論

提案方式によって、コンテキストスイッチによるオーバーヘッドが削減され、これまでの SoftwarePot の実装と比較してオーバーヘッドを大幅に削減できることがわかった。

クロスデバイスリンクへのアクセスはセキュリティポリシーに基づいて行なわれなければならない。しかし、提案方式においてまだ実装されていない。mapent 構造体に新たな属性を追加し、アクセスが発生した時に許可されている操作かどうかを調べるように実装する予定である。この実装によって、生じるオーバーヘッドは条件分岐命令 1 つ分のオーバーヘッドになる。

また、実際のアプリケーションを用いたオーバーヘッドの測定を行ないたい。また、マッピングファイルシステムの機能拡張を行ないディスク I/O の帯域制御を行ないたい。

参考文献

[1] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Virgil Gligor. SubDomain: Persimonious Server Security. In *Proceedings of the 14th Systems Administration Conference (LISA 2000)*, New Orleans, December 2000.

[2] D. Eres. A toolkit for user-level file systems. In *Proceedings of the 2001 USENIX Technical Conference*, June 2001.

[3] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications. In *Proceedings of the 6th USENIX Security Symposium*, San Jose, Ca., 1996.

[4] K. Kato and Y. Oyama. Softwarepot: An Encapsulated Transferable File System for Secure Software Circulation. Technical Report ISE-TR-02-185, Institute of Information Sciences and Electronics, University of Tsukuba, Jan. 2002.

[5] K. Kato, Y. Oyama, K. Kanda, and K. Matsubara. Software Circulation using Sandboxed File Space—Previous Experience and New Approach. In *8th ECOOP Workshop on Mobile Object Systems*, Malaga, Spain, June 2002.

[6] Douglas P. Shormley, David Petrou, Steven H. Rodrigues, and Thomas E. Anderson. SLIC: An Extensibility System for Commodity Operating Systems. In *USENIX 1998 Annual Technical Conference*, pp. 39–52, June 1998.

[7] M. Takahashi, K. Kono, and T. Masuda. Efficient Kernel Support of Fine-Grained Protection Domains for Mobile Code. In *Proc IEEE Int'l Conf. on Distributed Computing Systems(ICDCS)*, pp. 64–73, 1999.

[8] 大山恵弘, 神田勝規, 加藤和彦. 安全なソフトウェア実行システム SoftwarePot の設計と実装. 第 5 回プログラミングおよび応用のシステムに関するワークショップ SPA'02, 2002 年 3 月.