特集●ソフトウェアシステム

安全なソフトウェア実行システム SoftwarePotの設計と実装

大山 恵弘 神田 勝規 加藤 和彦

ソフトウェアを安全に実行するためのシステムSoftwarePotを開発した。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た状態で実行する。SoftwarePot用のソフトウェアパッケージの作者は、ソフトウェアを含む仮想的なファイルシステムの情報をアーカイブしたファイルを流通させる。SoftwarePotはファイル以外の資源（例えばネットワーク）へのアクセスを制御することもできる。仮想的なファイルシステムと資源へのアクセスの制御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。SoftwarePotはSolarisとLinux上に実装されている。広く使われているアプリケーションをSolaris上で実行する実験では、SoftwarePotの使用による実行時間の増加は最小21%、最大95%だった。

1 はじめに

今日インターネットを通じて悪意のあるソフトウェアが多数流通している。悪意のあるソフトウェアによる計算機システムへの被害を防止するために、資源へのアクセスが制限された実行環境（サンドボックス）を作るための技術[1] [3] [4] [6] [7] [9] [11] [15] [22] [27] [33]が盛んに研究されている。

本論文ではソフトウェアをサンドボックス内で安全に実行するシステムSoftwarePotについて述べる。本論文で使う「安全」の意味は、ファイルやネットワークなどの資源がユーザの意図に反したやり方でアクセスされないことである。ユーザの意図に反したやり方でのアクセスには、パスワードファイルの送信・改竄などのアクセス制御に関するものと、DoS攻撃などの資源消費に関するものがあるが、現在のSoftwarePotは前者の防止を主目的とする。SoftwarePotは仮想的なファイルシステムを含むサンドボックス（以下、**ポット空間**）を作る。ポット空間内の資源とポット空間外の資源は隔離される。すなわち、ポット空間内で走るソフトウェアはポット空間外にあるファイルを明示的な指示なしには参照できず、ネットワークへのアクセスもユーザの指示に従って制御される。SoftwarePot用のソフトウェアパッケージは、ソフトウェアが走る仮想的なファイルシステムの情報を含んでいる。仮想的なファイルシステムの実現や資源へのアクセスの制御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。

SoftwarePotはアプリケーションとOSの間に位置するユーザレベルのミドルウェアである。実行にスーパーユーザ権限は必要とせず、カーネルパッチも必要としない[†1]。SoftwarePotはプログラミング言語を規定しておらず、ソースが非公開のCOTS (Commer-

Design and Implementation of Secure Software Execution System SoftwarePot
Yoshihiro OYAMA, 科学技術振興事業団 Japan Science and Technology Corporation
Katsunori KANDA, 筑波大学 University of Tsukuba
Kazuhiko KATO, 筑波大学 University of Tsukuba, 科学技術振興事業団 Japan Science and Technology Corporation
コンピュータソフトウェア, Vol.0, No.0(2002), pp.00–00.
2002年0月0日受付.

†1 Linux上での実装では、SoftwarePotが使用するカーネルモジュールのインストール時にスーパーユーザ権限を必要とする。

cial Off-The-Shelf) ソフトウェア、レガシーバイナリ、スクリプトプログラムなども問題なく扱える。

本研究で我々が取り組んだ技術的課題は、ファイルシステムを含むサンドボックスの設計、そのサンドボックス内でソフトウェアを安全に実行するための仕組みの設計、それらの設計を具現化する実装方式の考案である。

本研究の貢献を以下に述べる。

- 流通するソフトウェアの安全な実行のための有用な枠組みを提案した。
- 上の枠組みを実現したシステム SoftwarePot の設計と実装を行った。

本論文は次のように構成される。第2章で SoftwarePot の設計を述べる。第3章で SoftwarePot の実装を解説する。第4章で SoftwarePot の性能を測定した実験の結果を報告する。第5章で議論する。第6章で関連研究との比較を通じて SoftwarePot の新規性を明確にする。第7章でまとめと今後の課題を述べる。

2 設計

2.1 概要

SoftwarePot はソフトウェアパッケージの作成と安全な実行を支援する。SoftwarePot が使われる様子を図1に示す。SoftwarePot は *encapsulate* **操作**と *execute* **操作**を提供する。

Encapsulate 操作は、ファイルアーカイブと、そのアーカイブを元に仮想的なファイルシステムを構築するための情報の組を格納したファイルを作成する。そのファイルを**ポットファイル**と呼ぶ。ポットファイルは RPM ファイルや Zip ファイルのようにソフトウェアパッケージとして流通する。Encapsulate 操作は、広く流通させるソフトウェアパッケージを作るために実行されることもあれば、ソフトウェアを走らせるための環境を個人的に作るために実行されることもある。

Execute 操作により、ポットファイルの利用者はポットファイルが表現する仮想的なファイルシステム上でソフトウェアを実行する。その際、セキュリティポリシーを与えて実行を制御する。セキュリティポリシーは、仮想的なファイルシステム上のファイル（以下、**仮想ファイル**）を実際のファイルシステム上のファイル（以下、**実ファイル**）にマップする規則、実ファイルへの操作を制限する規則、通信を制御するための規則を含む。セキュリティポリシーを記述したファイルを**ポリシーファイル**と呼ぶ。

2.2 仮想的なファイルシステム

SoftwarePot が作る仮想的なファイルシステムは、ファイル階層の構成がその OS 上の実際のファイルシステムと異なること以外は、通常のファイルシステムと同じように扱える。

仮想ファイルの実体は様々な場所に存在する。仮想ファイルは実体が存在する場所に従って以下のように分類される。

Static ファイル ポットファイルにファイルの内容が含まれるファイル。Encapsulate 操作時に実ファイルのコピーがポットファイルに格納される。

Map ファイル 実ファイルにマップされるファイル。ポットファイルの中には、マップ先のファイルを指し示す参照が格納される。マップ先のファイルは execute 操作を行うローカル計算機上にあっても遠隔計算機上にあってもよい。マップ先のファイルがローカル計算機上にある場合には実ファイルのパスが参照として格納され、遠隔計算機上にある場合には URL などの遠隔ファイルにアクセスするための情報が参照として格納される。ローカルファイルへのマップの用途の例は、デバイスファイルのマップや、ポット空間内で走るソフトウェアに編集させたい実際のファイルシステム上のデータファイルのマップである。遠隔ファイルへのマップの用途の例は、ポットファイルのサイズの削減や、最新版のファイルのネットワークを通じた提供である。

Indirect ファイル プロセスの標準入力または標準出力にマップされるファイル。ポットファイルの中にはコマンドの情報が格納される。Indirect ファイルがアクセスされるたびにそのコマンドを実行するプロセスが生成される。Indirect ファイルの用途には、実ファイルの内容をフィルタプ

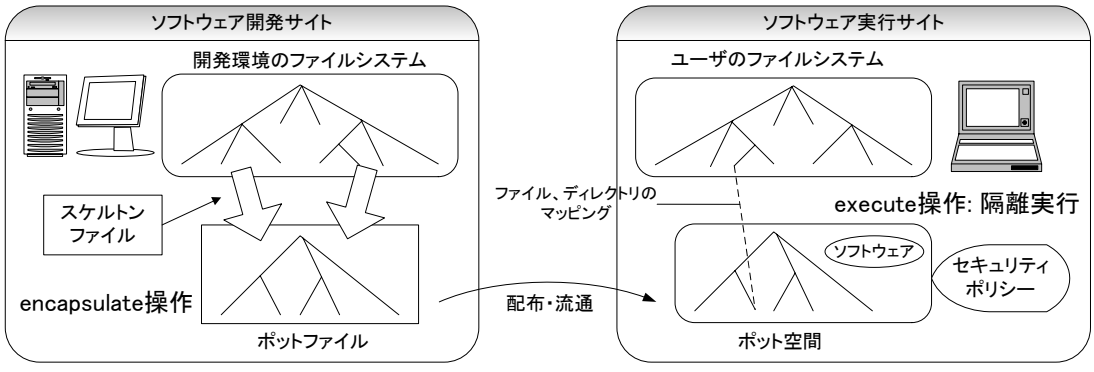

図 1 SoftwarePot が使われる様子

ログラムによって加工することなどがある。

ポットファイルが static ファイルの情報を与え、ポリシーファイルが map ファイルと indirect ファイルの情報を与える。

Encapsulate 操作では、どのパスにどのようなファイルを作るかを**スケルトンファイル**によって指示する。以下にスケルトンファイルの簡略化した例を示す。

```
static:
# virtual paths      # real paths
/mybin/viewer        $HOME/bin/viewer
/usr/lib/libc.so.1   /usr/lib/libc.so.1
/usr/lib/libdl.so.1  /usr/lib/libdl.so.1
/mypics/pic1.jpg     picA.jpg
...

entry: /mybin/viewer  -v  -X

required:
# virtual paths   # recommended map targets
/dev/null         /dev/null
/dev/zero         /dev/zero
/etc/passwd       /etc/passwd
/workdir          # no recommendation
/mybin/plugin     http://foo.com/tools/plugin
```

`static:`の部分には static ファイルを作るための指示を書く。第一列に仮想パスを与え、第二列にその仮想パスに作りたい内容をもつ実ファイルのパスを与える。`entry:`の部分には execute 操作で最初に起動されるコマンドを書く。`required:`の部分にはセキュリティポリシーを通じて作る必要がある map ファイルの情報を書く。第一列に仮想パスを与え、第二列に推奨するマップ先を（もし与えたければ）与える。`required:`の部分で指示された仮想パスの map ファイルがセキュリティポリシーを通じて作られていないときには、ソフトウェアの実行時にそのファイルを作るようユーザに警告が出される。その際、推奨するマップ先がユーザに提示される。`required:`の部分に書かれた情報は専用のコマンドでポットファイルから抽出して見ることができ、セキュリティポリシーの記述の際に参考にされる。

2.3 セキュリティポリシーによる実行制御

ソフトウェアパッケージの利用者はセキュリティポリシーによってポット空間内のソフトウェアが行える操作を制限する。以下にセキュリティポリシーの簡略化した例を示す。

```
map:
/dev/null       /dev/null
/dev/zero       /dev/zero
/workdir        $PWD/tmp
/mybin/plugin   http://foo.com/tools/plugin

indirect:
/etc/passwd  r  $HOME/bin/my_filter  /etc/passwd

socket:
deny  all
allow  inet  connect  *.is.tsukuba.ac.jp  80

redirect:
connect 202.226.93.133 23 -> 130.158.85.97 10023

path:
allow  all
deny  w  $PWD/tmp
```

`map:`の部分には map ファイルを作るための指示を書く。第一列の仮想パスは第二列のマップ先にマップされる。上の例では、ポット空間内で/workdir/a.txt を操作すると、実ファイルの$PWD/tmp/a.txt が操作される。`required:`の部分で与えられた推奨マップ先と異なるファイルにマップする指示を与えてもよい。

`indirect:`の部分には indirect ファイルを作るための指示を書く。第一列には仮想パスを与え、第二列にはそのファイルを読み出すかそのファイルに書き込むかの種別を r と w で与える。第一列の仮想パスのファイルは、第三列以降の列をコマンドとして起動したプロセスの標準入力または標準出力に結び付けられる。上の例では、ポット空間内で/etc/passwd を open すると、my_filter というプログラムが/etc/passwd という引数で起動される。my_filter は本物の/etc/passwd を加工したデータを標準出力に出力し、/etc/passwd を open したプロセスは、そのデータを/etc/passwd の内容として読み出す。

`socket:`の部分はソケットによる通信の制御を指示する。通信の許可と禁止の種別、アドレスファミリ、通信操作の種別、IP アドレスまたはドメイン名、ポート番号を記述する。

`redirect:`で始まる部分は通信のリダイレクト処理を指示する。上の例では、IP アドレスが 202.226.93.133 の計算機の 23 番ポートへの connect を、IP アドレスが 130.158.85.97 の計算機の 10023 番ポートへの connect に変更している。

`path:`の部分は、map ファイルを通じポット空間内からアクセス可能になっている実ファイルのアクセス制御を指示する。読み書きの種別を r と w の文字で与え、さらに、アクセス制御したい実パスのプレフィクスを与える。上の例では$PWD/tmp 以下のファイルへの書き込みを禁じている。

一つのポットファイルに対して様々な人が様々なポリシーファイルを記述する可能性がある [12]。典型的には、ポットファイルの作者がそのポットファイル用のポリシーファイルを記述し、ポットファイルと独立に配布することもありうる。計算機の知識を持つポットファイルの利用者が自分でポリシーファイルを記述することもありうる。ポットファイルの作者は必要に応じ、第三者機関や利用者によるセキュリティポリシーの記述を助けるための知識をポットファイルに添付する。計算機の環境に依存したポリシーファイルが必要な場合には、ポットファイルの作者は、一部の情報が抜けているポリシーファイルを配布し、ドキュメントなどを通じてその情報を各ユーザが埋めるよう伝える。

2.4 仮想的なファイルシステムの重ね合わせ

SoftwarePot では複数の仮想的なファイルシステムを重ねて使うことができる。Execute 操作で複数のポットファイルを与えると、それらが表現するファイルシステムを融合したファイルシステムが作られる。ユーザは複数のポットファイルの間に優先順位をつける。異なるポットファイルが同じ仮想パスにファイルを持つ場合には、優先順位の高いポットファイルの仮想ファイルが、融合したファイルシステム上に作られる。この機構は 3D ファイルシステム [14] および多重名前空間 [32] における viewpath 機構、Plan 9 [18] における union directory と同じものである。

ファイルシステムを重ねる機能は複数のソフトウェアパッケージを組み合わせて使うのに役立つ。例えば、あるアプリケーションを一つのポットファイルで提供し、そのアプリケーションを呼び出す別のアプリケーションを別のポットファイルで提供する。二つのポットファイルを組み合わせると、両方のアプリケーションを組み合わせて使える。

現在は仮想的なファイルシステム全体を単位としてのみ優先順位をつけられる。ポットファイル中の一部の仮想ファイルだけに高い優先順位を与え、残りの仮想ファイルに低い優先順位を与えることはできない。SoftwarePot の設計を変更し、[32] のようにファイルごとにポットファイルの優先順位を変えられる機能を導入することは可能である。

2.5 使用例

SoftwarePot の使用例を以下に示す。ポットファイ

ルは以下のように作成される。

```
> makepot  viewer.skl  viewer.pot
```

makepot は encapsulate 操作を行うコマンドである。スケルトンファイル viewer.skl の指示に従ってポットファイル viewer.pot が作られる。ポットファイル viewer.pot は配布され、以下のように実行される。

```
> execpot  viewer.plc  viewer.pot
(a window of the viewer shown)
...
```

execpot は execute 操作を行うコマンドである。viewer.plc は viewer.pot のために書かれたポリシーファイルである。viewer.pot がプラグインプログラムを呼び出す必要があるときなどは、

```
> execpot plugins.plc plugin1.pot plugin2.pot \
          viewer.plc viewer.pot
```

のように、複数のポットファイルと複数のポリシーファイルが同時に与えられることがある。このとき、後ろのコマンドライン引数のファイルには、前のコマンドライン引数のファイルによりも高い優先順位が与えられる。

ユーザは viewer.pot を実行する前に viewer.plc を読み、それが安全かどうか確認することもある。その結果、viewer.plc が過度に緩いセキュリティポリシーであることに気づくかもしれない。例えば、

```
map:
/etc  /etc
```

という指示があり、ポット空間内で走るソフトウェアが/etc/passwd を盗める状態になっているかもしれない。このような場合、ユーザはドキュメントなどを参考にソフトウェアが本当に必要とするファイル集合を見定め、viewer.plc を

```
map:
/etc/netconfig            /etc/netconfig
/etc/.name_service_door   /etc/.name_service_door
```

などのより厳しいものに書き換える。

3 実装

SoftwarePot は Solaris と Linux 上に実装されている。SoftwarePot は約 16000 行の C 言語のコードで記述されている。

Encapsulate 操作は本質的には単なるアーカイブ操作であり、実装は自明である。

Execute 操作ではプロセスが fork される。子プロセスがソフトウェアを実行し、親プロセスはその子プロセスを監視する†2。以後、ソフトウェアを実行するプロセスを**ポットプロセス**と呼び、ポットプロセスを監視するプロセスを**モニタプロセス**と呼ぶ。モニタプロセスは実際のファイルシステム上に一時ディレクトリを作り、その下のファイル階層に仮想ファイルの実体を保持する。モニタプロセスはポットプロセスが発行するシステムコール引数の検査と書き換えによってポット空間を実現する。モニタプロセスはファイルパスを引数に持つシステムコールと資源にアクセスするシステムコールにフックを挿入し、ポットプロセスをそれらのシステムコールの実行前で停止させる。それら以外のシステムコールではポットプロセスは停止しない。

仮想的なファイルシステムを実現するための処理の流れを以下に述べる。ポットプロセスがシステムコールフックで停止したら、モニタプロセスはシステムコール引数の仮想パスを読む。そしてそれを実パスに変換し、変換後のパスをポットプロセスが参照できるメモリ空間に書き込む。Solaris 上の実装では環境変数の領域に書き込み†3、Linux 上の実装ではカーネルモジュールが確保したメモリ領域に書き込む。

もしその仮想ファイルが遠隔ファイルにマップされているならば、モニタプロセスは必要に応じて遠隔計算機に接続し、ファイルの実体を得るなどの処理を行う。得られたファイルの実体はキャッシュされて以降の実行で再利用される場合もあれば、キャッシュされない場合もある [29]。もしその仮想パスのファイルが indirect ファイルならば、プロセスを生成し、そのプロセスの標準入力（出力）とポットプロセスをパイプで結合し、そのプロセスに所定のコマンドを実行させる。

最後に、システムコールハンドラに変換後のパスが

†2 文献 [23] の細粒度保護ドメインを用いると、ソフトウェアの実行とその実行の監視を同じプロセス内で行える可能性がある。その場合、現在の実装よりもオーバヘッドが小さくなる可能性がある。

†3 Solaris 上の実装では、SoftwarePot は非常に長い文字列の値を持つ環境変数の定義をポットプロセスに強制的に加える。モニタプロセスはその文字列の領域をパスを格納する領域として使う。

渡されるようにレジスタの値をセットしてポットプロセスの実行を再開する。システムコールの実行が終わったら、変換後のパスの格納の際に行われたメモリの変更を元に戻す。

ポットプロセスがマルチスレッドである場合、あるスレッドが発行したシステムコールの引数を、検査・書き換え後に別スレッドが書き換える可能性があるが、現時点ではそれに対しては特に対処していない。

ポットプロセスがセキュリティポリシーに違反する操作をしたら、モニタプロセスがそのポットプロセスを kill する。ポットプロセスが fork したら、モニタプロセスも自らを fork し、常に一つのモニタプロセスが一つのポットプロセスを監視する。一方、ポットプロセスによる exec の実行に対しては、モニタプロセスは特別な処理を行わない。

Solaris 上の実装では、モニタプロセスは/proc ファイルシステムを利用してポットプロセスを制御する。システムコールフックの挿入、ポットプロセスのメモリ空間の読み書き、ポットプロセスが実行中か停止中かの検査などはすべて/proc 以下のファイルを読み書きすることによって行われる。Linux 上での実装では、それらの処理はカーネルモジュールによって行われる。

4 実験

SoftwarePot を使って実験を行った。実験に使った OS と計算機は Solaris 2.6 と Sun Ultra 60 Workstation (UltraSPARC II, 主記憶 512MB) である。

まず我々はセキュリティポリシーに違反するアプリケーションの実行が強制終了されることを確認した。例えば、許可されていないホストへの通信を試みたアプリケーションが強制終了されることを確認した。

本章の以降の部分では、セキュリティポリシーに違反しないプログラムの実行にかかるオーバヘッドを測定した結果を報告する。

4.1 マイクロベンチマーク

以下の三つのマイクロベンチマークプログラムを使い、SoftwarePot がシステムコールの実行に加えるオーバヘッドを測定した。

Fib フィボナッチ数を計算する。システムコールをまったく呼び出さない。

getpid-loop ループで getpid システムコールを繰り返し（10 万回）呼び出す。他のシステムコールを呼び出さない。

open-loop 同じファイルを open してすぐに close する操作をループで繰り返す。open と close システムコールを共に 10000 回呼び出す。

SoftwarePot を使った場合と使わない場合の実行時間を比較した。実験結果を表1に示す。Fib の性能は SoftwarePot を使うことによってほとんど変化しなかった。Getpid-loop は SoftwarePot を使うと明らかに遅くなった。getpid はパス関連システムコールではないためフックされないので、この結果は不思議に見えるかもしれない。我々は、いくつかのシステムコールがフックされると、フックされないシステムコールにもオーバヘッドが加わると推測している。Open-loop の実行時間は SoftwarePot の使用により 8246 ミリ秒増加した。Open-loop は open を 10000 回呼び出し、かつ、close のオーバヘッドは極めて小さいので、SoftwarePot は open の 1 回の実行あたり約 0.8 ミリ秒のオーバヘッドを加えると考えられる。

4.2 アプリケーションベンチマーク

以下のアプリケーションを使って SoftwarePot のオーバヘッドを測定した。どのアプリケーションもファイルを頻繁にアクセスする。

Emacs Lisp シェルスクリプトと 216 個の Emacs Lisp ファイル（.el ファイル）を使う。スクリプトは `emacs` をバッチモードで起動する。`emacs` はすべての.el ファイルを.elc ファイルにバイトコンパイルする。コンパイルが終了したらそのスクリプトは.elc ファイルをユーザが指示したディレクトリにコピーする。

LaTeX シェルスクリプト、一つの LaTeX ソースファイル、三つの LaTeX スタイルファイル、17 の EPS ファイルを使う。スクリプトは `latex` と `dvi2ps` を呼び出して LaTeX ソースファイルから PostScript (PS) ファイルを作成する。次に PS ファイルをユーザが指示したディレクトリにコ

	非使用	使用
Fib	13.4	13.5
getpid-loop	10.6	14.7
open-loop	0.192	8.44

表 1 SoftwarePot の非使用時と使用時のマイクロベンチマークプログラムの実行時間（秒）

ピーする。その PS ファイルは 15 ページの英語論文であり、サイズは 1.6MB である。

make & gcc シェルスクリプトと grep 2.0 のソースツリーを使う。スクリプトはまず `make` と `gcc` を呼び出してソースファイルをコンパイルする。次にスクリプトはソースツリーの中のテスト集によって、作られた grep バイナリの動作を確認する。最後に、ユーザが指示したディレクトリにそのバイナリをコピーする。

catman シェルスクリプトと 189 個の nroff 入力ファイル（オンラインマニュアルの第 2 章すべて）を使う。スクリプトは `catman` を起動し、整形されたマニュアルと windex ファイルを生成する。次に windex ファイルをユーザが指示したディレクトリにコピーする。

実験で使用したポットファイルは、アプリケーションが必要とするファイル（例えばライブラリファイル）を全て static ファイルとして含んでいるわけではない。この実験では、ライブラリファイル、/etc 以下の設定ファイル、デバイスファイルに関しては全て実ファイルをマップした。

実験結果を図2に示す。左側の棒が SoftwarePot を使わない場合の実行時間を示し、右側の棒が SoftwarePot を使った場合の実行時間を示している。実行時間は正規化されている。Emacs Lisp の性能低下が約 21%で最小であり、LaTeX の性能低下が約 95%で最大であった。性能低下幅はファイルパス関連システムコールの実行頻度に大きく影響されると我々は考えている。Emacs Lisp, LaTeX, make & gcc, catman の 1 秒あたりのファイルパス関連システムコールの実行回数はそれぞれ 68, 368, 168, 373 であった。

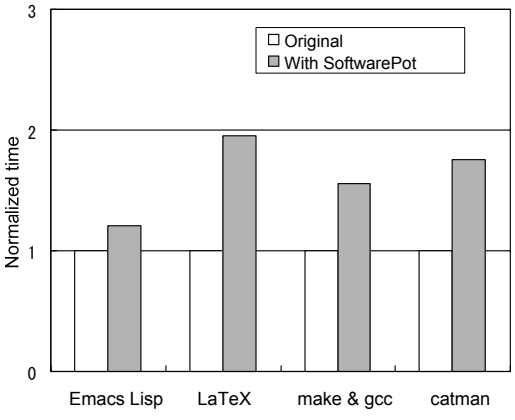

図 2 アプリケーションの実行時間

SoftwarePot のコードには多くの最適化の余地が残されているので、今後 SoftwarePot のオーバヘッドを縮小していくことは可能である [30]。

5 議論

5.1 用途

SoftwarePot は以下の用途で有用である。

- パッケージ管理ツール（“Secure RPM”）。RPM [8] などの既存のパッケージ管理ツールを使ったインストール作業では悪意あるコードを実行させられる可能性がある。例えば、ある RPM パッケージは.login に悪意あるコードを加えるかもしれない。SoftwarePot を使えば、アクセスされうるファイル群を厳しく制限した状態でソフトウェアパッケージを利用できる。
- 信頼できないソフトウェアを実行するサンドボックス。例えば、教師が学生のレポート課題のプログラムを SoftwarePot を使って実行する。プログラムにバグや悪意が含まれていても、教師の計算機資源は守られる。
- ワークフローシステムやモバイルエージェントなどのネットワークアプリケーションのためのミドルウェア [13]。各サイトで encapsulate 操作を実行し、サイト間でポットファイルを転送する。各サイトは自前のセキュリティポリシーを用意して自分のサイトを守る。
- ファイル置換ツール（パスリダイレクトツール）。

例えばライブラリファイルや/etc 以下の設定ファイルを自分用のものに置き換えたり、/usr/java を~/tmp/java にリダイレクトさせることができる。この種の処理は、実際のファイルシステムの構成に依存せずにソフトウェアに望みの動作をさせるのに役立つ。別の有用な例には、ポット空間内に/tmp を作り、信頼できないソフトウェアから本物の/tmp を隠す処理がある。これは/tmp に存在する情報の漏洩や/tmp へのアクセス権を利用した攻撃（例えば[5]）を防ぐのに役立つ。

- サーバを走らせるためのサンドボックス。FTP サーバや web サーバをポット空間の中で走らせることによって、サイトの安全性を高められる。現在この用途には chroot が使われることが多いが、SoftwarePot は chroot のより高機能な代替となりうる。

5.2 ポットファイルの作成作業

ポットファイルの作者はソフトウェアの実行に必要な全てのファイルを仮想的なファイルシステム上に準備する必要があるが、どのプログラムバイナリがどのライブラリファイルを必要とするかなどのファイルの依存関係を常に完全に把握できるとは限らない。仮に把握できたとしても、把握には手間がかかる。この問題は、RPM パッケージや動的リンクライブラリの依存関係の管理、chroot におけるファイル木の構築にも存在する。この問題の完全な解決は難しい。部分的な解決策としては、例えば、研究[10][19]のように、依存関係の検出をツールで支援することが考えられる。

5.3 セキュリティポリシーの書きやすさ

ある場合には、SoftwarePot は Janus [7] などのサンドボックスを提供する既存のシステムよりも、セキュリティポリシーの記述の手間や人間による安全性検査の手間が小さい。その場合とは、仮想的なファイルシステムの機能によって、アクセスを制御する必要がある実ファイルの集合を縮小できる場合である。パッケージ作成者がより多くのファイルをポットファイルに含め、実ファイルへのマップをより少なくすれば、セキュリティポリシーの作成者および検査者がアクセス制御について注意を払う必要がある実ファイルはより少なくなる。極端な例では、実ファイルへのマップを全く必要としないパッケージも存在する。そのパッケージ用のセキュリティポリシーはファイルのアクセス制御の指示を含まず、記述と検査は容易である。一方、多くのファイルのマップが必要なポットファイルのためのセキュリティポリシーは、記述と検査の容易さの点では既存のシステムと大きく違わず、論文[7] の 13 ページの Janus のセキュリティポリシーに似たものになる。結局、SoftwarePot でのセキュリティポリシーの記述が既存のシステムでのそれと比べてどれくらい容易であるかは、パッケージ作者がパッケージをどの程度「自己完結」させるかに大きく影響される。

5.4 コード署名アプローチとの安全性の比較

計算機の知識がないユーザは、典型的には誰かが署名つきで配布するセキュリティポリシーを、その誰かへの信頼を安全性のよりどころにして利用する。この状況では、信頼する対象が変わっただけなので、SoftwarePot はコード署名アプローチに比べて高い安全性は提供しない。SoftwarePot の安全性がコード署名アプローチより高くなるのは、計算機の知識を持つ人（セキュリティポリシー利用者自身がそうである場合も含む）がセキュリティポリシーを読んで安全性を検査する場合である。検査という過程を挟むことにより、署名者への信頼とセキュリティポリシーを検査した者への信頼を足したものを安全性のよりどころにすることができる。

6 関連研究

サンドボックスを提供するシステムは多数提案されている。例えば、システムコールフックを利用したサンドボックス[1][7][27][33]、呼び出せるライブラリ API を制限したインタプリタによるサンドボックス[15]、カーネル拡張によるサンドボックス[4][6][9]、Java などの仮想機械を利用したサンドボックスがある。それらのシステムでは各資源に関して、アクセスさせるかアクセスさせないかを選択できる。しかし、

SoftwarePot にあるような、仮想的な資源を作ってアクセスさせるという選択肢は提供されない。

PCC [16] や SFI [26] はソフトウェアの実行がセキュリティポリシーに従うことを静的手法によって強制する。一方、SoftwarePot はシステムコール引数の検査と書き換えという動的手法を用いる。SoftwarePot は静的手法を使うものよりもオーバヘッドが大きい反面、静的に強制できないポリシーを扱える利点を持つ。静的手法と動的手法を組み合わせる試みは [20] に見られる。

ソフトウェアのインストール時にソフトウェアパッケージに含まれる未知のコードが実行されることがある。例えば RPM [8] や shar や自己解凍アーカイブによるパッケージを使う場合にそれが起こりうる。SoftwarePot は実際のファイルシステムからできる限り隔離した仮想的なファイルシステム上でソフトウェアパッケージを実行することを可能にし、未知のコードの実行による重要な資源への被害を回避する。

Ufo [2] と Consh [3] は SoftwarePot と同じくシステムコールフックを利用して仮想的なファイルシステムを作る。smbsh [21] は、システムコールフックでなくライブラリフックを利用して、Prospero [17] は、専用のサーバにファイル名の解決をさせて、仮想的なファイルシステムを作る。Plan 9 [18] の名前空間の機能は、仮想的なファイル名と実ファイルの対応を各プロセスが自分で動的に変更することを可能にする。これらの研究には、仮想的なファイルシステムの機能をソフトウェアパッケージの作成と実行に利用するという着想は見られない。

Chroot を利用してソフトウェアを指定のディレクトリ以下に閉じ込めて実行するシステムがある [22]。これらのシステムでは、NFS などの支援機構なしでは chroot されたディレクトリの外のファイルを参照できない。そのため chroot されるディレクトリ内にライブラリファイルなどの必要なファイルをコピーすることが多いが、それはファイルのバージョン管理を困難にする。一方 SoftwarePot では、map ファイルを用いれば、セキュリティポリシーで制御された形で、実際のファイルシステム上のファイルを仮想的なファイルシステムから参照できる。また、chroot の実行にはスーパユーザ権限を必要とする。そのため、chroot は一般ユーザが日常的にソフトウェアを実行する用途に適さない。

NT 系 Windows や Solaris などの OS が備えるアクセス制御リストを使うと、どのような操作を許可するかをファイル単位、ユーザ単位、ユーザグループ単位で制御できる。この仕組みは SoftwarePot と異なり、ソフトウェアの種類およびユーザの各状況での意図に応じて臨機応変にアクセス制御の仕方を変えるものではなく、また、仮想的な実行環境を作るものでもない。

信頼できる署名が付加されたコードのみを実行するための枠組みがある。例えば Microsoft の ActiveX の Authenticode がある。この枠組みでは、署名者への信頼のみがコードの信頼性を判断するよりどころであるため、無名な署名者によるコードを流通させることが難しい。一方、SoftwarePot では、ソフトウェアの利用者などがセキュリティポリシーを検査したり修正することによって、署名者を信頼できない状況下でもコードを安全に実行できる。

世界 OS [28] もシステムコールのパス引数の書き換えを用いてファイルシステムの上に別のファイルシステムを仮想的に作る。加えて、第2.4節で述べたような、ファイルシステムを重ねる機能も持つ。世界 OS が作る仮想ファイルシステムは、実際のファイルシステムまたは別の仮想ファイルシステムを継承したもの（同じファイル構成を持つもの）である。世界 OS では仮想ファイルシステム上のファイルを実際のファイルにマップできない。仮想ファイルシステム上で行ったファイル変更を実際のファイルに反映させる手段としては、その仮想ファイルシステム上で行った全てのファイル変更を反映させる操作しかない。また、世界 OS は仮想的なファイルシステムを一つのファイルの形で表現して流通させるための機能を持たない。

仮想 OS・仮想機械上で信頼できないコードを実行して計算機システムを守ることがある。例えば User-mode Linux [24] および FreeBSD の jail [11] が提供する仮想 OS や、VMware [25] が提供する仮想機械がその目的に使える。我々の知る限り、これらのシステムでは、仮想的な資源の集合をソフトウェア

パッケージとして流通させるための特別なサポートはない。研究[31] では、VMware の仮想ディスクの配布によってソフトウェアを配布することを試みているが、VMware の仮想ディスクは配布したいファイル以外の情報も含むために非常に大きい。一方、SoftwarePot では、必要最低限のファイルだけをポットファイルに含ませることによってポットファイルを小さく保てる。

7 まとめと今後の課題

SoftwarePot の設計と実装を述べた。SoftwarePot の基本アイデアは、ソフトウェアを必要な実行環境ごとサンドボックスに閉じ込めることと、流通させるソフトウェアパッケージに実行環境を含めることである。現在は実行環境として特にファイルシステムを扱っている。SoftwarePot によって、実際のファイルシステムからできる限り隔離された環境でソフトウェアを実行することができる。我々が行った実験では、SoftwarePot がアプリケーションの実行を 2 倍以上遅くさせることはなかった。

今後の第一の課題は、セキュリティポリシーの仕様の拡張である。現在の仕様では、巨大なメモリを確保したり多数のプロセスを fork するような DoS 攻撃を防ぐのが難しい。また、Java の permission のように、ユーザとコードベースのような実行コンテキストに従ってポリシーを変化させることもできない。第二の課題は、ファイル間の依存関係を調べる機能を SoftwarePot に追加し、ポットファイルとセキュリティポリシーの作成作業を支援することである。

SoftwarePot に関する情報は SoftwarePot ホームページ (http://www.osss.is.tsukuba.ac.jp/pot/) で得られる。

謝辞

本研究について、電気通信大学の河野健二氏、東京大学の関口龍郎氏と住井英二郎氏、筑波大学の新城靖氏と石井孝衛氏、産業技術総合研究所の一杉裕志氏から有益なコメントをいただきました。

参考文献

[1] Anurag Acharya and Mandar Raje. MAPbox: Using Parameterized Behavior Classes to Confine Untrusted Applications. In *Proceedings of the 9th USENIX Security Symposium*, August 2000.

[2] Albert Alexandrov, Maximilian Ibel, Klaus E. Schauser, and Chris J. Scheiman. Ufo: A Personal Global File System Based on User-Level Extensions to the Operating System. *ACM Transactions on Computer Systems*, 16(3):207–233, August 1998.

[3] Albert Alexandrov, Paul Kmiec, and Klaus Schauser. Consh: Confined Execution Environment for Internet Computations. http://www.cs.ucsb.edu/~berto/papers/99-usenix-consh.ps, December 1998.

[4] Andrew Berman, Virgil Bourassa, and Erik Selberg. TRON: Process-Specific File Protection for the UNIX Operating System. In *Proceedings of USENIX Winter 1995 Technical Conference*, pages 165–175, January 1995.

[5] Matt Bishop and Michael Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, 1996.

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

[7] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker. In *Proceedings of the 6th USENIX Security Symposium*, pages 1–13, July 1996.

[8] Red Hat. RPM. http://www.rpm.org/.

[9] Xie Huagang. Build a Secure System with LIDS. http://www.lids.org/, October 2000.

[10] InDependence. http://www.cse.ogi.edu/DISC/projects/independence/.

[11] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the 2nd International System Administration and Networking Conference (SANE 2000)*, May 2000.

[12] Kazuhiko Kato and Yoshihiro 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, January 2002.

[13] Kazuhiko Kato, Yoshihiro Oyama, Katsunori Kanda, and Katsuya Matsubara. Software Circulation using Sandboxed File Space – Previous Experience and New Approach –. In *Proceedings of the 8th ECOOP Workshop on Mobile Object Systems (MOS 2002)*, June 2002.

[14] David G. Korn and Eduardo Krell. A New Dimension for the Unix File System. *Software – Practice and Experience*, 20(S1):19–34, 1990.

[15] Jacob Y. Levy, Laurent Demailly, John K.

Ousterhout, and Brent B. Welch. The Safe-Tcl Security Model. In *Proceedings of the USENIX Annual Technical Conference (NO 98)*, June 1998.

[16] George C. Necula. Proof-Carrying Code. In *Proceedings of the 24th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*, pages 106–119, January 1997.

[17] B. Clifford Neuman. The Prospero File System: A Global File System Based on the Virtual System Model. *Computing Systems*, 5(4):407–432, 1992.

[18] Plan 9. http://cm.bell-labs.com/plan9dist/.

[19] Vessilis Prevelakis and Diomidis Spinellis. Sandboxing Applications. In *Proceedings of 2001 USENIX Annual Technical Conference, FREENIX Track*, June 2001.

[20] R. Sekar, C. R. Ramakrishnan, I. V. Ramakrishnan, and Scott A. Smolka. Model-Carrying Code (MCC): A New Paradigm for Mobile-Code Security. In *Proceedings of New Security Paradigms Workshop (NSPW '01)*, pages 23–30, September 2001.

[21] smbsh. http://www.samba.org/samba/docs/.

[22] Lincoln D. Stein. SBOX: Put CGI Scripts in a Box. In *Proceedings of the USENIX Annual Technical Conference*, June 1999.

[23] Masahiko Takahashi, Kenji Kono, and Takashi Masuda. Efficient Kernel Support of Fine-Grained Protection Domains for Mobile Code. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS '99)*, pages 64–73, May–June 1999.

[24] User-Mode Linux. http://user-mode-linux.sourceforge.net/.

[25] VMware. http://www.vmware.com/.

[26] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *Proceedings of the 14th ACM Symposium on Operating System Principles (SOSP '93)*, pages 203–216, December 1993.

[27] 阿部洋丈, 加藤和彦, 王維. セキュリティポリシーの動的切替機構を持つリファレンスモニタシステム. 情報技術レターズ, 2002 年.

[28] 石井孝衛, 新城靖, 板野肯三. プロセストレース機能を用いた世界 OS の実現. コンピュータシステム・シンポジウム論文集, pages 89–96, 2001 年 11 月.

[29] 大山恵弘, 神田勝規, 加藤和彦. SoftwarePot/iPAQ: PDA の環境に適応するソフトウェアパッケージの作成と実行を支援するミドルウェア. 第 1 回情報科学技術フォーラム (FIT 2002) 講演論文集, 2002 年 9 月.

[30] 神田勝規, 大山恵弘, 加藤和彦. カーネルモジュールを用いた SoftwarePot の効率的な実行方式. システムソフトウェアとオペレーティング・システム研究会研究報告 No. 90, pages 81–86, 2002 年 6 月.

[31] 須崎有康. ネットワークを渡り歩けるコンピュータ. 第 3 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 2000 論文集, 2000 年 3 月.

[32] 東村邦彦, 加藤和彦, 松原克弥. アクティブネットワーク技術を用いた多重名前空間の実現法. 情報処理学会論文誌, 41(6):1665–1676, 2000 年 6 月.

[33] 東村邦彦, 松原克弥, 相河亨, 加藤和彦. モーバイルオブジェクトシステム Planet のプロテクション機構. 第 1 回インターネットテクノロジーワークショップ論文集 (WIT '98), 1998 年 8 月.