平成19年度 修士論文

SAccessor: デスクトップＰＣのための安全なファイルアクセス制御システム

東京工業大学大学院 情報理工学研究科
数理・計算科学専攻

学籍番号 06M37154

滝澤 裕二

指導教員

千葉 滋 准教授

平成17年2月7日

概要

本論文ではデスクトップ PC のための安全なファイルアクセス制御システム SAccessor について示す。従来システムでは OS による認証とファイルアクセス制御機構によりファイルへの不正アクセスを防いできた。しかし OS にも脆弱性が報告されており、それらの脆弱性を突いた攻撃を受けてしまうと OS によるアクセス制御が無効化される恐れがある。OS への攻撃の影響を受けずにアクセス制御をするために、バーチャルマシンやストレージデバイス等、攻撃対象の OS とは分離された箇所においてアクセス制御を行うことが考えられるが難しい。OS が攻撃されている可能性があるため OS の情報は信用できず、バーチャルマシンやストレージデバイス単体ではユーザの情報が不足しているためアクセスが不正かどうか判断することができないからである。

SAccessor はファイルアクセス制御を攻撃対象の OS から分離し、上記の問題を解決して実用的に使えるようにした。SAccessor では VMM を用いて一台のマシン上にユーザがログインして作業する作業 OS と、認証とファイルアクセス制御を行う認証 OS の 2 つの OS を動作させる。ディスクは認証 OS が保持していて作業 OS は直接ファイルを操作することはできず NFS を通してファイルへアクセスする。このようにすることで作業 OS がクラックされても認証 OS 側でアクセス制御を強制することができる。ログファイルやプロセス ID 等一部のファイルを除いたシステムファイルは読み込みのみ作業 OS へ許可し、システムファイルへの書き換えを防ぐ。システムファイルの書き換えは認証 OS が提供するサービスを通してのみ可能である。認証 OS のサービスの起動時には認証 OS が認証ダイアログを表示し作業 OS を介さずに直接ユーザ認証を行う。この認証は作業 OS を介さないため、作業 OS に侵入した攻撃者からはこの認証を成功させることができない。ユーザのファイルへのアクセスにも上記と同じ認証を行う。ユーザファイルへの認証はポリシに定義されたファイルのグループ毎に認証を行う。認証には有効期間を設定でき、有効期間内では認証は省略することができる。認証の有効期間の切れたファイルへのアクセスを制限するために、作業 OS 上に残ったファイルキャッシュはフラッシュさせる。

我々は VMM として Xen、各 OS として Linux をベースに NFSD を改

造することで SAccessor の実装を行った。実験の結果、SAccessor を使用した場合とローカルの ext3 ファイルシステムを使用した場合と比較すると最悪で約 34 %のオーバーヘッドになり、バックエンドに仮想計算機と NFS を用いていることが主な原因になることを確認した。100MB ファイルキャッシュのフラッシュのためには約 3,7 秒かかりその間作業 OS は停止するが、その間に発行されたマウスイベントやキーストロークは取りこぼし無く作業 OS にわたることを確認した。

謝辞

本研究を支えてくれた多くの方々への感謝の意をここでは表したいと思います。指導教官の東京工業大学 千葉滋 准教授には大学四年生から大学院修士課程修了までの三年間、研究の方向性や進め方、プレゼンテーションの仕方や資料作成法等、多方面に渡りご指導頂きました。深く感謝いたします。

東京工業大学　光来健一助教には、本研究の当初から包括的なご指導を頂きました。本研究を進めるにあたっての基礎知識、関連技術、論文の書き方や学会発表の技術について様々なご意見を頂きました。心より感謝いたします。千葉研究室の西澤無我氏には、関連研究や最新技術の動向などの知識をご指導いただきました。柳澤佳里氏には、多くの疑問を聞いていただき、研究室生活全般にわたり助言を頂きました。田所秀和氏には、仮想計算機や OS に関する高度な技術についてご指導いただきました。心より感謝いたします。

同研究室の Romain LENGLET 氏、石川零氏、薄井義行氏、日比野秀章氏、青木康博氏、熊原奈津子氏、竹内秀行氏、堀江倫大氏、栗田洋輔氏、今吉竜之介氏、内河綾氏、戸部敦氏、赤井駿平氏、安積武志氏、森田悟史氏のおかげで充実した研究室生活を送ることができました。最後に小生を精神的に支えてくださった東京工業大学 サイクリング部のみなさまに心より感謝しております。

目 次

図 目 次

表 目 次

第1章　はじめに

ビジネスや個人利用でデスクトップPCを使用するようになり、重要なファイルがデスクトップPCに保存されていることが多くなっている。しかし、コンピュータにはセキュリティホールが数多く報告されており、デスクトップPCは常にウィルスメールや、トロイの木馬などの攻撃を受ける危険に晒されている。従来システムではOSによるファイルアクセス制御機構により、これらの攻撃は限定されてきた。例えば、一般ユーザ権限で動く攻撃プログラムは管理者権限を必要とするファイルにはアクセスすることができない。

しかし、OSによるファイルアクセス制御はOSカーネルの脆弱性を利用した攻撃を受けると機能しなくなる可能性がある。近年、OSカーネルの脆弱性は数多く見つかっており、修正パッチが公開される前に行われるゼロデイアタックやデスクトップPCにパッチを未適用であることが問題となっている。OSカーネルの脆弱性を攻撃されると、ユーザ認証をバイパスして管理者権限を奪われたり、アクセス制御そのものを無効化される危険性がある。

既存技術であるセキュアOSでは、従来の管理者権限に依存した管理や単純なファイルオーナーによる権限制御を、より厳格なアクセス制御機構によって補完している。例えば、アクセス制御を強化して絶対的な管理者権限をなくし、クラッカーの侵入を実質的に無効化する等である。セキュアOSは強制アクセス制御によってセキュリティ管理者以外はセキュリティ設定が不可能である。ユーザ別のアクセス制御によって管理者権限を分割し特権の集中を防ぐ、プロセスごとのアクセス制御が可能といった機能がある。しかし、セキュアOSでもOS自体の脆弱性が攻撃された場合には、権限の分割が無効化され全てのファイルにアクセスできるOSレベルの権限をクラッカーが取得できる危険がある。

これはセキュアOSや既存のセキュリティ機構の大部分がOSによって実現されているために起こる問題である。この問題を解決するために、ストレージデバイスや仮想計算機（VM）など攻撃対象のOSの外部でアクセス制御を行うことが考えられるが、これは簡単ではない。例えば、ストレージデバイスでアクセス制御をする場合を考える。完全にデバイス単体でアクセス制御を行おうとすると、「どのユーザがどのファイルに」アク

セスしようとしているのかが判断できず適切にアクセス制御をすることができない。そのため、ストレージデバイスへのアクセスに対しては常に許可をするか常に拒否するかの制御しかできない。OS からユーザの情報を受け取る場合には、OS が攻撃者によってクラックされている可能性があるため OS からの情報を元にアクセス制御をすることはできない。ファイルサーバを利用すればアクセス制御を OS から分離することができるが、クライアントを乗っ取られると正規ユーザになりすまされてファイルへの不正アクセスを許してしまうなどの問題がある。このように、単純に OS の外部にアクセス制御機構を用意してやるだけでは実用的なシステムを作るのは難しい。

オープンネットワーク 認証システムである kerberos では、個々の PC が管理するパスワードで認証を行うのではなく一括管理を行っているサーバで認証を行う。クライアントは認証サーバで認証を行い期限付きのチケット認可チケットをサーバから受け取り、そのチケット認可チケットをチケット認可サーバに提出してサービスを利用するのに必要なチケットを獲得する。このチケットにはサービスを利用するときに使う鍵を含まれる。期限付きのチケット認可チケットを使うことで、チケットの再利用性を制限しチケット漏洩に対する危険を最小限にしている。しかし、Kerberos 認証ではクライアントマシンがクラックされ秘密鍵やパスワードが漏洩した場合には、クラッカーがユーザになりすまして、認証サーバの認証を成功させてしまう危険がある。

これらの問題を解決するために、本論文では認証とファイルアクセス制御とを攻撃対象の OS から分離し、実用的にアクセス制御を行えるようにしたシステム SAccessor を提案する。SAccessor は仮想計算機モニタ（VMM）を用いて 1 台のマシン上でユーザがログインする作業 OS と、ファイルアクセス制御を行う認証 OS を動作させる。作業 OS は認証 OS と通信してファイルアクセスを行い、認証 OS はポリシに従って安全にユーザ認証を行う。作業 OS と認証 OS は VM のレベルで安全に分離されているため、この認証とアクセス制御は作業 OS がクラックされた場合にもバイパスすることはできない。我々は安全なユーザ認証や実用的なアクセス制御を行うために、認証ダイアログ、ファイルキャッシュのフラッシュ、Setuid されたプログラムを安全に実行する方法の開発を行った。以下、2 章で従来システムについて述べ、3 章で提案するシステム SAccessor の設計について述べる。4 章で SAccessor の実装、5 章で評価のために行った実験について述べ、6 章で本稿をまとめる。

第2章　コンピュータのセキュリティ問題と

近年、デスクトップPCに対する攻撃が増えてきている。例えば攻撃者はメールに悪意のあるプログラムを添付し、メーラの脆弱性やユーザの不注意を利用してプログラムを実行させる。P2Pソフトウェアの場合にはユーザの関心を引くような名前のファイルを共有し、ユーザにダウンロードさせて実行させる攻撃などがある。このような攻撃を受けてしまうと、機密情報が流出する恐れがある。またウィルスがシステムファイルを書き換えて感染することで、システム起動時にバックドアが起動するようになってしまう。さらに攻撃者がPCへの侵入に成功した場合にはログファイルが削除されてしまい、侵入の発見が困難になる。

UNIXはOSの持つAccess Control Listによってユーザのファイルに対するアクセス権限を管理しており、SELinux[12]はカーネルの拡張モジュールとして実装されるように、従来システムではこのような攻撃は、OSによる認証とファイルアクセス制御によって防いできた。例えば一般ユーザがウィルスメールを開いてしまった場合、そのウィルスは一般ユーザ権限で動作するため、管理者のファイルであるシステムファイルを書き換えるような攻撃はできない等である。。

しかし、OSにも脆弱性が見つかっている。例えば、ローカルユーザが管理者権限を取得できる脆弱性[4]、OSカーネルをバッファオーバーフローさせて任意のコードを実行させる脆弱性[3]も報告されている。このような脆弱性に対しては修正パッチが公開されるのが普通だが、ユーザのセキュリティ意識が低い、パッチを当てると動かなくなるアプリケーションがあるなどの理由によりパッチがあたっていない場合も多い。また、修正パッチが公開される前に攻撃を受けてしまうゼロデイアタック、修正パッチにより新たな脆弱性などの危険もある。

このようなOSの脆弱性を攻撃されると、OSが提供しているファイルアクセス制御そのものが無効化されてしまう。たとえSELinux[12]などのセキュアOSで管理者権限を制限していても任意のファイルへのアクセスを許してしまう。任意のファイルへのアクセスを許してしまうと、ログファイルを書き換えて侵入の痕跡を消去したり、システムの設定変更、バックドアやキーロガーの設置などの行為が行われ対応が困難である。

OS の脆弱性への攻撃の影響を避けるために、VMM やストレージデバイスなど、OS の外部にアクセス制御機構を用意することでこの問題を解決できると思われるがこれは簡単ではない。例えばストレージデバイスでアクセス制御をする場合を考える。まず、ストレージデバイスが単独でアクセス制御を行う場合、デバイス単体ではユーザやファイルという情報がわからないため、どのアクセスを許可していいか、あるいは拒否するべきかの判断ができない。そのため、常に許可をだすか、常に拒否をするかという制御しか行うことができない。次に OS からユーザ等の情報を受け取って制御する場合、OS がクラックされているかもしれないために OS から渡される情報を使ってしまうと適切な制御ができないという問題がある。

以下、本章ではセキュリティに関連する基本技術であるセキュア OS[12][7]、Kerberos[13][6]、仮想計算機の既存技術の Xen[1] について述べる。また、先行して提案されているセキュリティ技術 Proxos[15]、SVFS[19]、ストレージベースの侵入検知システム [8][14]、ファイル完全性チェックツールである XenRIM[9][10] について述べ、最後にこれらのシステムでは、コンピュータのファイルアクセスに対する安全性や利便性が不十分であることを述べる。

2.1　セキュア OS

2.1.1　セキュア OS とは

いままでの OS で十分に行われていなかったアクセス制御を強化し、より強いセキュリティ機構を持つ OS がセキュア OS である。セキュア OS と従来 OS の違いは、従来 OS のセキュリティ技術が「どのように侵入を阻止するか」に注目していたのに対し、セキュア OS では「コンピュータに侵入された後のこと」に着目している。セキュア OS ではアクセス制御を強化して、侵入を実質的に無効化する。これまでの OS が抱えている問題には以下のようなものがある。

- ファイル所有者がアクセス権を勝手に変更できる（これは任意アクセス制御と呼ばれている）。
- root アカウントは、アクセス制御を無視して全てのファイルにアクセスできる。
- プロセスに特権を与える際に余計な特権までも与えてしまうため、プロセスが乗っ取られた場合、システムに大きな被害をもたらす

セキュア OS では上記の問題を解決するために「強制アクセス制御（MAC: Mandatory Access Control)」と呼ばれるアクセス制御機構を備えている これは、ファイルの所有者が勝手にアクセス権を変更できないようにし、システム全体を管理する者の意図通りのアクセス権を強制するものである。MAC は root アカウントにも強制力を持つ。root アカウントがファイルなどのアクセス権を変更するには、特定の手順を用いてシステムの状態を変更してからアクセス権を変っこうする必要がある。

セキュア OS の別の特徴は「最小特権」である。セキュア OS ではプロセスに特権を与える際に、全ての特権を一度に与えるのではなく、細かく分割された特権を少しずつ与えることができる。これにより、プロセスに不必要な特権を与えてしまう可能性が少なくなり、仮にプロセスが乗っ取られたとしてもシステムに及ぼす影響を最小限にできる。

MAC と最小特権という 2 つの機能を満たしているものとされている。Linux 用の代表的なセキュア OS モジュールには以下の 3 つがあげられる。

- SELinux[12]
- LIDS[7]
- RSBAC[11]

セキュア OS は攻撃者が PC に侵入に成功した後のことを対象にしているが、攻撃者が OS の脆弱性をついて OS レベルの特権を取得した場合には無効化されてしまう危険がある。本研究で提案する SAccessor では OS レベルの特権を攻撃者が取得したことを想定したセキュリティ機構になっている。

2.1.2　LSM

Linux カーネル 2.6 から新機能として「LSM(Linux Security Module)」機能が追加された。LSM とは、カーネル内のセキュリティチェック気候へのフック関数郡を定義するフレームワークを提供する機能（運用環境固有のセキュリティカーネルを実装するための）である。LSM を用いることで、カーネルのセキュリティチェック機能をユーザが独自に拡張することが可能である。

LSM を有効にすると、I/O ポートアクセスの許可などのセキュリティチェックポイントで、ユーザの登録した LSM のコールバック関数が呼び出され、操作の正当性チェックが行われる。LSM で定義可能なセキュリティチェックポイントは 150 項目以上に及ぶ。主な主要項目として、以下のような操作に対するセキュリティ機構が実装可能である。

- I/O ポートへのアクセス
- ホスト/ドメイン名の設定
- システムのシャットダウン
- プロセスの生成/終了
- 各種シグナル操作
- 各種ファイルシステム操作
- 各種ソケットの操作

2.1.3　SELinux

SELinux(Security-Enhanced Linux) とは、LSM に対応した Linux カーネルのセキュリティ拡張モジュールである。SELinux は Linux カーネルに「セキュア OS」の機能を付加する。

アプリケーションにセキュリティホールが発見されてからパッチが供給され、そのパッチが適用されるまでの間は、サービスを停止するか、攻撃を受けないことを期待しながらセキュリティホールのある状態で運用を継続するしかない。どちらにしても、いち早くパッチを適用することが最優先となる。しかし、パッチの適用にはそれが稼働中のほかのアプリケーションに及ぼす影響も検証しなければならない。検証のために十分な時間をとろうとするとパッチの適用が遅れ、クラッカーから攻撃を受けてしまうかもしれない。これは Linux が root アカウントという絶対的な管理アカウントに依存していることが問題である。root に全権限を与えてしまっているために、攻撃者に侵入を許し root 権限を奪われてしまったときの被害が大きくなってしまう。

SELinux を導入することで、この問題はある程度解決できる。SELinux では従来の root に依存した管理や、単純なファイルオーナーによる権限制御を、より厳格な 3 種類のアクセス制御機構によって補完する。3 種類のアクセス制御機構とは以下のものである。

- TE(Tyep Enforcement)
  従来の Linux では、動作するプロセスはそれぞれを実行するユーザ権限でファイルなどのリソースにアクセスする。つまり、各プロセスについてオーナ、グループとパーミッションに基づいた権限の制御しかできなかった。SELinux では TE という機構が用意されている。TE ではプロセスに対し「ドメイン」、ファイルに対し「タイプ」というラベルが付加される。各リソースはファイルやソケット

などの「オブジェクトタイプ」毎に「アクセスベクタ」が割り当てられる。アクセスベクタとはリソースに対して行える操作の種類である。例えばファイルに対するオブジェクトタイプ「file」には read、write、lock、append など約２０種類のアクセスベクタが存在する。SELinux ではプロセス毎にタイプに対して許可されるアクセスベクタを設定できる。つまり、プロセス A はファイルの読み書きができるが、別のプロセス B はファイルを読むことしかできないといったように、プロセス毎に固有のアクセス制御を詳細に行うことができる。

- RBAC(Role Based Access Control)
  RBAC はユーザのリソースへのアクセス制御機構である。ユーザにはそれぞれ管理者、一般ユーザといったロールを割り当て、各ロールには最低限必要な操作が可能なポリシを設定し、リソースへのアクセスを許可する。RBAC を使うと root アカウントであっても他のユーザと同じように限定された権限しか許可しないようにできる。これにより攻撃者が root 権限を取得したとしても、割り当てられたロールの範囲でしか攻撃を行えないため、大幅にリスクを減らすことができる。

- MAC(Mandatory Access Contol)
  MAC ではポリシファイルに記述された設定に基づいて TE、RBAC などのアクセス制御を全てのユーザとプロセスに例外なく適用する。ポリシファイルの設定を行えるのは限定されたシステムの管理者だけである。

しかし、SELinux では限定された権限内での行為は対応できない、また OS そのものにセキュリティホールがある場合にも攻撃を防ぐことはできない。

2.2　Kerberos

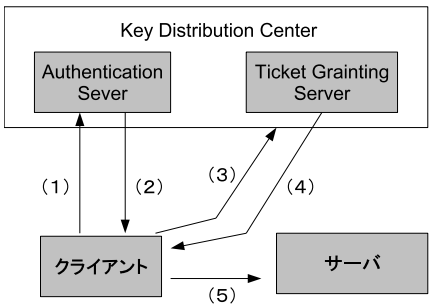
Kerberos はオープンネットワーク 認証システムである。PC にログインするときには一般的にはログインする PC で管理されているパスワードでの認証が行われるが、Kerberos ではパスワードを一括管理する認証局のようなものを持つ。Kerberos 認証システムでは一度認証してしまえば、その認証局の管理する領域内ではサービスを提供する PC が複数台あってもすべてのサービスを受けられる。この仕組みのことをシングル・サインオンという。

一度認証を行ったクライアントはチケット認可チケット (TGT) という期限付きのチケットを認証局から受け取る。この期限付きのチケットは、

期限内であれば何回でも使用できる。目的のサービスを利用する際にはパスワードの代わりに TGT を使いサービス毎にサービスチケットを手に入れる。サービスチケットにはサービスを利用するために必要な鍵が含まれていて、実際にサービスを利用するときに使う。このサービスの有効期限も TGT の有効期限と同じになる。kerberos のチケットはディスクに保存されるが、チケットに有効期限を設定することでチケットを盗まれたときのなりすましの可能性を最小限にしている。

Kerberos では「信頼できる第三者機関」による認証モデルを基本としている。その第三者機関が認証局の働きをするため鍵発行局（Key Distribution Center）と読んでいる。KDC は実質上認証サーバ（Authentication Server）、認可サーバ (Ticket Granting Server) からなる。図 2.1 は Kerberos による認証、サービス利用までの流れである。クライアントがあるサービスを利用した後に、続けて別のサービスを利用したいとすると、ユーザは TGT を TGS に再び提出してチケットを入手することができる。TGT はクライアントの秘密鍵から作られるが、作るたびに異なり秘密鍵そのものではないためキャッシュして再利用しても比較的安全になる。

しかし、kerberos 認証ではクライアントマシンがクラックされ秘密鍵が盗まれ、キーロガーなどのツールによりパスワードが漏洩した場合には攻撃者がユーザになりすまして認証を成功できる危険がある。



1. TGT の要求
2. TGT の入手
3. TGT を提出してサービスチケットを要求
4. サービスチケット入手
5. サービスチケットを提出しサービスを要求

図 2.1: ケルベロスによる認証

2.3　計算機の仮想化技術

仮想化技術には様々な種類があるが、計算機の仮想化技術について 3 つ紹介する。

- Full-Virtualization
  ソフトウェア的に 1 台の PC を丸ごとエミュレートする手法。通常の OS をそのままインストールして実行できる。代表的な Full-Virtualization の仮想計算機として VMware[17]、VirtualPC[16] などがある。

- Para-Virtualization（準仮想化）
  完全な PC を仮想マシンとして実現するのではなく、擬似アーキテクチャの仮想マシンを構築する手法。ベースとなるホスト OS が必要ない。Full-Virtualization と比べ実行速度が速い。しかし、OS カーネルを擬似アーキテクチャに合わせて修正する必要がある。代表的なものは Xen[1] である。

- VMM(Virtual Machine Monitor)
  VMM は Para-Virtualization を発展させて、OS に修正を加える必要をなくした手法。Full-virtualization と比べ実行オーバーヘッドが少なく、ホスト OS も必要ない。この手法は Xen3.0 から Intel VT[5] 対応の CPU と組み合わせることで実現可能になった。

2.3.1　Xen 概要

Xen は para-virtualization を用いた仮想化技術である。以下では Xen 上で動作する OS のことをゲスト OS、ゲスト OS が動作する仮想計算機のことをドメインと呼ぶ。また Xen そのもののことをハイパーバイザーと呼んでいる。これは、Xen がゲスト OS のスーパーバイザーコードよりも高い特権レベルで動作するからである。Xen 上で動作させる OS には、Xen の提供する擬似アーキテクチャ向けにゲスト OS を修正しなければならないが、Application Binary Interface(ABI) は変更しないためゲスト OS 上で動くアプリケーションは修正不要である。Xen は以下のような設計の原則を持っている。

- ゲスト OS 上のアプリケーションの修正が不要であること。従って、OS の既存の標準的な ABIs を維持する。

- 複雑なサーバの設定を一つのゲスト OS のインスタンス上でできるようにするため、ゲスト OS 上では複数のアプリケーションを動作させられること。

- 高いパフォーマンスを実現するために para-virtualization が不可欠である。
- 複数のアーキテクチャが共存している場合であっても、資源の仮想化の影響をゲスト OS から隠すこと。

図 2.2 は Xen のアーキテクチャである。Xen(VMM) はマシン上で直接動作するソフトウェアのレイヤーであり、Xen の上には 1 つ以上の仮想計算機が動作する。それぞれの仮想計算機上にはそれぞれ異なる OS を動作させることができ、仮想計算機上で動作する OS のことをゲスト OS と呼ぶ。それぞれのゲスト OS の資源は Xen によって管理され、お互いの資源には直接アクセスできないようになっている。

Xen では仮想計算機をドメインと呼ぶ。ドメインにはハードウェアに直接アクセスできる特権的な仮想計算機であるドメイン 0 と特権を持たないドメイン U がある。ドメイン U はドメイン 0 を介してハードウェアにアクセスを行う。また、ドメイン 0 は Xen の管理ツールが動作し、仮想計算機の作成や削除等の管理を行う。

Xen はそれぞれの仮想計算機に対して VMM の動作するハードウェアと似た仮想的なデバイスを提供する。ゲスト OS はこの仮想的なデバイスに対して I/O アクセス要求を行う。この仮想デバイスへの I/O 要求はドメイン 0 に転送されドメイン 0 のデバイスドライバが実デバイスを操作する。

2.3.2　仮想計算機インターフェイス

表 2.1 は準仮想化された x86 のインターフェイスの概要である。MemoryManagement,CPU,I/O について順に説明をする。

Memory Management

x86 アーキテクチャはソフトウェア管理の TLB を持っていないため、メモリの仮想化はアーキテクチャを準仮想化する中で一番難しい部分である。x86 では TLB がミスすると自動的にプロセッサーがハードウェアのページテーブルを調べに行く。さらに TLB はタグ付けされていないため、アドレス空間の変更するときには TLB をフラッシュさせる必要がある。これらの制限のため Xen では二つの決断をした。

- ゲスト OS がハードウェアのページテーブルの割り当てと管理する。
- 全てのアドレス空間のトップに Xen のために 64MB のメモリを予約する。

Memory Management	
Segmentation	完全に特権的なセグメントのディスクリプタをインストールすることはできない、また論理アドレスの一番上をオーバーラップすることはできない。
Paging	ゲスト OS はハードウェアのページテーブルに直接アクセスできる。しかし、更新はハイパーバイザが行う。各ドメインには不連続なマシンページを割り当てることができる。
CPU	
protection	ゲスト OS は Xen よりも低いレベルで動作する
Exceptions	ゲスト OS は例外処理のハンドラを Xen に登録しなければならない。
System Calls	ゲスト OS はシステムコールに対して fast なハンドラを登録できる、これは Xen を介する間接的な呼び出しを避け、アプリケーションがシステムコールを呼べるようにするためである。
Interrupts	ハードウェアによる割り込みは軽いイベントシステムに置き換えられる。
Time	それぞれのゲスト OS はタイマーインターフェイスを有し、real と virtual の両方の時間がわかる。
Device I/O	
Network,Disk,etc	データは非同期な I/O リングを通して転送される。イベントのメカニズムがハードウェアの通知のための割り込みの代わりになる。

表 2.1: para-virtualized x86 インターフェイス

新しいプロセスを生成するなどゲスト OS が新しいページテーブルを要求する場合には、ゲスト OS は自分のメモリ空間にページを確保、初期化し Xen に登録する。そのため、OS はメモリのページテーブルへ直接アクセスする権限を持たず、ページテーブルの更新や有効化は Xen によって行われる。そのためページテーブルを更新するためにはハイパーバイザーコールを呼び出すオーバーヘッドが生じる。またアドレス空間の上位 64MB は Xen 用に予約され、ゲスト OS からは操作することができないが通常の x86ABIs ではこのアドレスは使われていないため、アプリケーションの互換性は失われない。

CPU

ハイパーバイザを OS よりも高い特権レベルで動かすことは一般的な OS が最も特権レベルの高いという想定を違反するが、OS のバグからハイパーバイザーを守るために必要になる。多くのプロセッサには二つしか特権レベルが無いものがある。この場合には、ゲスト OS はアプリケーションと同じ特権レベルの低いほうのモードで動作させる。ゲスト OS はアプリケーションと別のアドレス空間で動作してアプリケーションからの干渉を防ぎ、ハイパーバイザーの仮想的な特権レベルを利用してアプリケーション、ハイパーバイザー間のアドレス空間の変更を行う。

x86 アーキテクチャでは４つのレベルが存在する。特権レベルはリングで表現され、リング０からリング３までである。リング０が最もレベルが高く、リング３が一番特権レベルが低い。Xen ではリング０でハイパーバイザーが、リング１でゲスト OS、リング３でアプリケーションのコードが実行される。

メモリフォールトやソフトウェアトラップを含む例外処理はそれぞれの例外処理に対してハンドラを Xen に登録しておく。ハンドラを指定するのは実際の x86 アーキテクチャの場合と変わらない、なぜなら例外のスタックフレームは Xen の準仮想化アーキテクチャでは変更していないからである。唯一変更してあるのはページフォールトのハンドラである。リング０以外でページフォールトを起こすと Xen のハンドラは例外スタックフレームのコピーをゲスト OS のスタック上に作ったあと、適切なハンドラに制御を渡す。

I/O

既存のハードウェアデバイスをエミュレートするために、Xen は簡単なデバイス抽象を提供している。これは典型的な Full-Virtualization の方法

と同じような手法である。I/O のためのデータはそれぞれのドメイン間で共有メモリを通して受け渡しされる。

ハードウェア割り込みと同様に Xen は軽量なイベント配送のメカニズムをサポートしている。このメカニズムはドメインに非同期に通知するのに使われる。通知はゲスト OS の登録したイベントハンドラの呼び出し時に発行される。

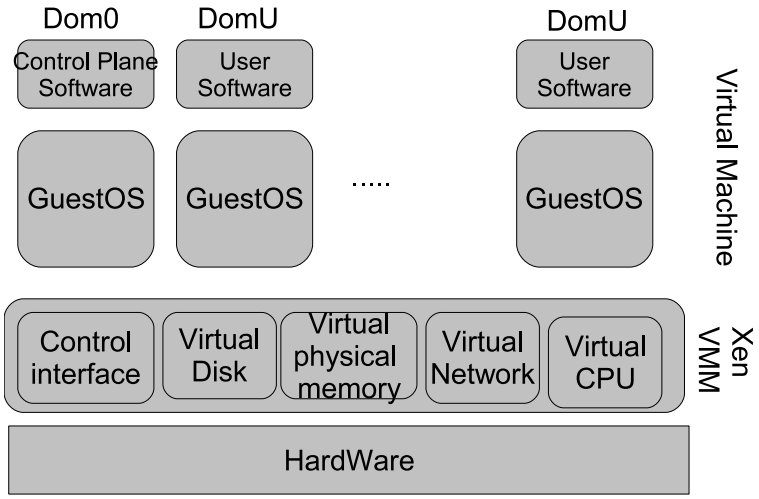


図 2.2: Xen のアーキテクチャ

2.4 Proxos

Proxos はアプリケーションから発行されたシステムコールを、隔離された VM と常用の VM に分配することで、アプリケーションの OS への信頼度を制限することができるシステムである。システムコールの分配の規則はアプリケーションの開発者が決めなければならない。

2.4.1 アーキテクチャ

図 2.3 は Proxos のアーキテクチャである。一台のマシン上に、Commodity OS VM と、プライベート VM、（図には書かれていないが）管理 VM が動作している。管理 VM は Xen でいうところのドメイン 0 であり、他の VM を管理するツールを持っている。Commodity OS VM は標準的機能（ファイルシステム、ネットワーク etc...）を備えた OS が動いて、プライベート VM で Commodity OS から分離して実行したいアプリケーションを動かす。プライベート VM 上で実行するアプリをプライベートア

プリと呼ぶ。プライベート VM 上の Methods は OS の機能を実装したもので、隔離されたプライベート VM 上で実行したいシステムコールは全て Methods が行う。Proxos はアプリケーションから発行されたシステムコールを扱うライブラリで、アプリケーションとは静的にリンクされる。そのため、アプリケーションから発行されたシステムコールは全て Proxos のサブルーチンに変換される。Proxos のサブルーチンによって、セキュリティを考慮しなくていいシステムコールは RemoteProcedureCall を使って常用 OS へ、セキュリティを考慮すべきシステムコールはプライベート VM の Methods へそれぞれ割り振られる。

2.4.2　プライベートアプリ

アプリケーションを Proxos のライブラリと静的にリンクさせるために開発者はアプリケーションのソースコードを変更し、再コンパイルする必要がある。しかし、Proxos は Commodity OS のシステムコールと同じインターフェイスを提供しているので、この修正は一般的に少量で済む。例えば、Glibc ライブラリ (ver2.3.3) は 218 行の変更で Proxos に対応させることができている。さらに、プライベートアプリ、Methods、ルーティング規則、Proxos は一つのバイナリファイルとしてコンパイルされなければならない。これは、空の VM 上にロードして実行できるようにするためである。開発者はこのバイナリイメージを VMM の管理者に渡し、管理者が VMM に登録する必要がある。VMM に直接保存することで、Commodity OS がクラックされてもプライベートアプリは守られる。

プライベートアプリを起動するために、Commodity OS のユーザはホストプロセスを起動する。ホストプロセスは VMM にプライベートアプリを起動するための新しい VM をインスタンス化を依頼する。ホストプロセスはプライベートアプリと一対一対応するものであり、プライベート VM から常用 OS に転送されたシステムコールはホストプロセスの UID で実行される。このようにすることで、プライベートアプリと常用 OS 上で動く別のアプリケーションを区別することなく扱うことができる。ホストプロセスを通して、プライベートアプリは Commodity OS 上で動く他のアプリとインタラクティブに処理を行うことができる。例えば、mknod と open、read、write システムコールを常用 OS 側で実行するような規則を書けば、プライベートアプリは名前付きパイプを使い他のアプリケーションと通信することができる。

2.4.3　Proxos の安全性

Commodity OS がクラックされても VMM へ攻撃はできない、VM の独立性は壊れない、アプリ開発者が適切な規則を書くという仮定のもとでは、Proxos はプライベートアプリが持つデータの機密性と一貫性を保つことができる。まず VM の独立性から、Commodity OS がクラックされた場合にも攻撃者がプライベートアプリのデータに直接影響を与えることはできない。攻撃者が干渉できるのは、Commodity OS にルーティングされたシステムコールだけである。しかし、Commodity OS にルーティングされるシステムコールは開発者が non-security-sensitive と考えるものであるので、クラックされた OS がプライベートアプリに重大な影響を与えることはできない。

さらに、プライベートアプリから発行されるシステムコールはホストプロセスの UID で処理されるため、アプリを起動したユーザの権限しかもたない。そのため既存のセキュリティ機構を弱めるようなことは無い。

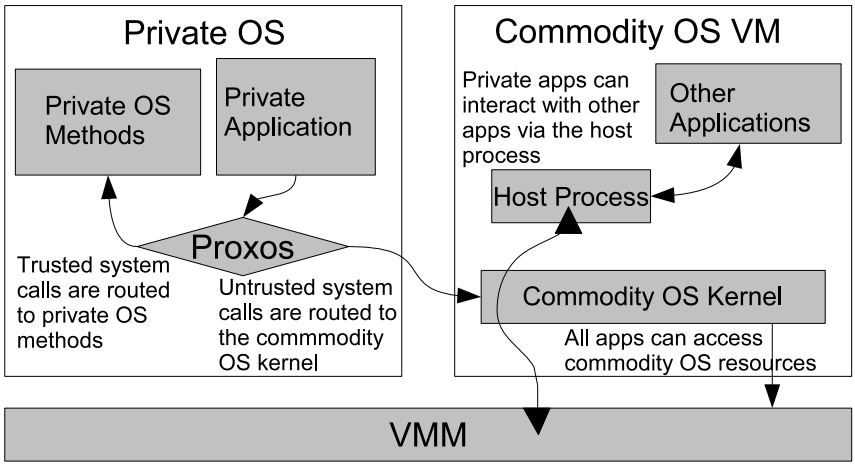


図 2.3: Proxos のアーキテクチャ

2.4.4　Proxos のルーティング記述言語

Proxos は開発者の規定した規則に従ってシステムコールのルーティングを行う。例えば read システムコールでは、読み込むファイルによって異なるルーティングを行うことができる。

Linux のシステムコールは 200 種類以上あり、それら全てに対してルーティング規則を記述するのは複雑であり時間が掛かってしまう。Proxos の

開発者らはシステムコールをアクセスする OS 資源別に分類し、それらの資源に対してルーティング規則を記述できるようにした。OS 資源を以下の 6 つに分類した。

- ストレージ（disk）
- ユーザインターフェイス
- ネットワーク
- 乱数
- システムタイム
- メモリ

その他の周辺機器 (printer,USB,etc) は OS によってファイルという形に抽象化されるのでストレージに分類する。

アプリケーションがシステムコールを Commodity OS で行わなければならない場合は次の 2 通りの理由がある場合である。一つめは他のアプリケーションとの通信するために OS の資源を利用する場合、二つ目はアクセスする OS 資源を信用しなくて良い場合または Commodity OS からしかアクセスできない資源を利用する場合である。従って、ストレージとユーザインターフェイス、ネットワークに関する処理はプロセス間の通信に使われる場合には Commodity OS で実行する必要がある。一方、システムタイムと乱数に関するシステムコールはプライベート VM 上で実行する。これは、プロセス間の通信には使われないため Commodity OS 上で実行する必要はなく VMM によって提供されているためである。メモリに関するシステムコール（brk [1], mprotect [2]）は間接的にアプリケーションのページテーブルのエントリを操作するため、プライベート VM 上で実行しなければならない。

図 2.4 はルーティング規則の記述例である。#はコメントを表す。Rules セクションにはルーティングする OS 資源と、操作関数へのポインタを記述する。DISK:(”/etc/secrets”,priv_fs) は/etc/secrets ファイルへのアクセスは priv_fs に登録された関数で行うという意味である。

Methods セクションでは、プライベート VM でシステムコールを扱う関数を定める。図 2.4 の例では、/etc/secrets ファイルをオープンする場合には priv_open 関数が呼ばれる。図には書かれていないが、priv_tcp、priv_unix も priv_fs と同様に定義する。ルーティング規則が指定されていない操作は全て Commodity OS で実行される。

[1]データ・セグメントのサイズの変更する
[2]メモリ領域の保護を設定する

```
#Rules Section
#route accesses to /etc/secrets to private OS
DISK:("/etc/secrets",priv_fs)

#route accesses to UNIX domain socket bound
#to /tmp/socket and TCP socket bound to peer
#192.100.0.4 port 1337 to private OS
NETWORK:("unix:/tmp/socket",priv_unix),
        ("tcp:192.100.0.4:1337",priv_tcp)

#route all accessedcs to stdin ,stdout
#and stderr to private OS
UI: (*,priv_ui)

#Methods Section
#individual methods in the private OS
#that are bound to system calls

priv_fs = {
    .open = priv_open,
    .close = priv_close,
    .read = priv_read,
    .write = priv_write,
    .lseek = priv_lseek
}
```

図 2.4: Proxos Routing Language

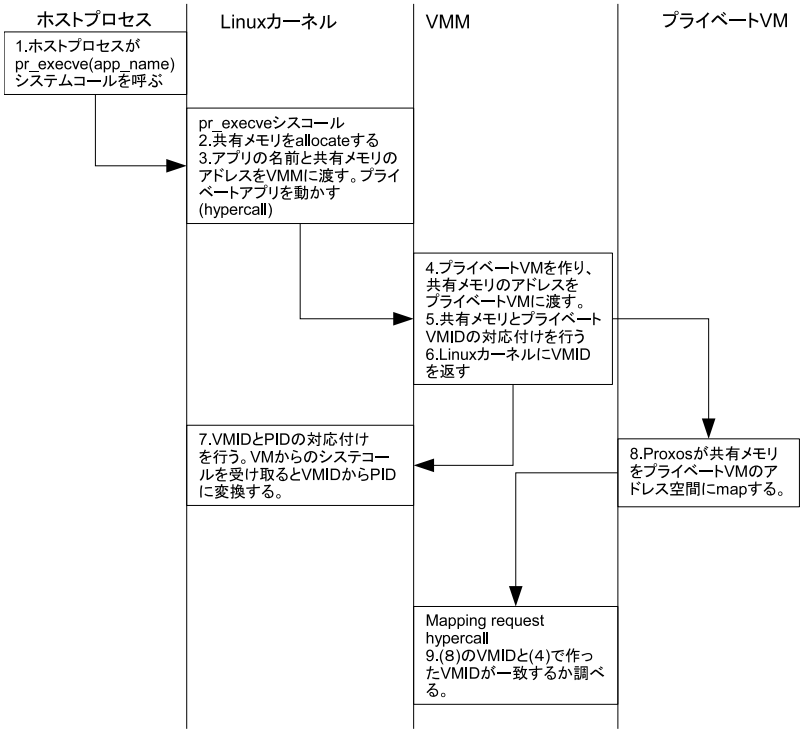


図 2.5: Proxos の実装

2.4.5　Proxos の実装

図 2.5 はプライベートアプリケーションを起動するまでの流れである。Proxos のプロトタイプは VMM として Xen、Commodity OS として Linux を用いている。プライベートアプリはあるプロセスが pr_execve システムコールによっり起動することができる。pr_execve は execve に似た新しく実装したシステムコールである。pr_execve はプライベートアプリの名前を引数に取る。また pr_execve を呼び出したプロセスがプライベートアプリのホストプロセスになる。

pr_execve の中では、共有メモリを確保する。この共有メモリは後にプライベートアプリからのシステムコールの転送に使われる。同時に、共有メモリのアドレスを VMM に渡し、プライベートアプリを動作させるための新しい VM を作るために、新しく追加した hypercall を使って管理 VM へシグナルを送る。管理 VM はプライベートアプリのために VMM の機能を呼び出す。

VMM は新しいプライベート VM を作り、共有メモリのアドレスを渡す。Linux カーネルにはプライベート VM の VMID を知らせる。Linux

カーネルは受け取った VMID を元に、転送されてきたシステムコールを対応するホストプロセスの UID の権限で実行する。

プライベートアプリのシャットダウンは、VM をシャットダウンするハイパーコールを拡張し、Linux カーネルに VM が終了したことを通知するようにした。通知を受け取った Linux カーネルはホストプロセスを終了させる。プライベートアプリが終了してないのに、ホストプロセスを終了させた場合には、プライベートアプリからの OS 資源へのアクセス要求は破棄される。

Proxos ではアプリケーションの開発者がシステムコールレベルのポリシを書かなければならないため、導入が難しいという問題がある。SAccessor のポリシはファイル単位のポリシであるため、導入が簡単である。

2.5　SVFS

2.5.1　SVFS の脅威モデル

SVFS は OS がクラックされた状況での重要ファイルの保護を対象としている。OS がクラックされた状況とは、攻撃者が OS レベルの特権を獲得したと仮定している。多くの攻撃者は特権を取得した後、重要ファイルを修正し正常に動作しているアプリケーションを誤動作させようとする。一方、コンピュータの管理者は攻撃者が重要ファイルを書き換える、侵入の痕跡を消す、システムがリブートするときに自動実行するような悪意のあるコードの追加を防ごうとする。SVFS では以下のファイルとディレクトリを保護する。

- システムの設定ファイル (e.g /etc/rc.d/rc.sysinit)
- システムの実行ファイル
- 共有ライブラリ
- 重要なシステムのディレクトリ（e.g /etc/rcX.d）
  これらのディレクトリ上に悪意のあるコードを設置されるとシステムの起動時に自動実行されてしまう。
- システムのログファイル
  侵入の痕跡を消すために攻撃者がアクセスする
- その他管理者が重要だと思うファイル

SVFS は攻撃者が OS レベルの特権を取得することは想定しているが、攻撃者が物理的にマシンにアクセスできる場合にファイルを守ることはでき

ない。また、SVFS を管理するために使われる通信は安全であると仮定している。

2.5.2　SVFS のアーキテクチャ

SVFS はユーザの作業とファイルアクセス制御を別々の VM 上で行うシステムである。それぞれの VM 毎に異なるアクセス権を与えてファイルを守る。SVFS では 3 種類の VM がある。ユーザが作業を行う Normal VM と重要ファイルへのアクセスを行う Admin VM、それとファイルを保持する DVM である。図 2.6 は SVFS のアーキテクチャである。ファイルアクセスは Normal VM あるいは Admin VM が DVM と通信をして行う。よってファイルアクセスは 2 つのドメインをまたがって行われる。DVM はアクセス要求がどの VM からきたかによってアクセス制御を行う。Normal VM 上で悪意のあるプログラムが動作したとしても、ファイルアクセス制御はファイルシステムを保持する DVM 上で行われるためにアクセス制御をバイパスすることはできない。

Normal VM からの重要ファイルへのアクセスは Read しかできない、重要なファイルへの Write アクセスは Admin VM から行うことになる。

2.5.3　SVFS の実装

VMM には Xen をファイルアクセスのためのドメイン間通信には Virtual Remote Procedure Call を実装している。VRPC は標準的な RPC と同じプロトコルやワーキングフローを持っているが、データ転送にはネットワーク経由ではなく、ドメイン間の共有メモリ空間を使い行っているため RPC と比べてパフォーマンスの向上を行っている。

SVFS では重要ファイルへの Write アクセスを行うためには、特別な VM にログインしなおさなければならないという問題がある。一般ユーザでも重要ファイルを変更する場合があり、これは大変な手間である。一方 SAccessor では 2 つの VM を使用するがユーザは 1 つの VM にログインするだけで、重要ファイルへのアクセスが可能になる。ただし、別 VM にユーザからキー入力を渡す必要があるが、これをシームレスに行う手法を提供している。

2.6　XenRIM

ファイルシステムの完全性チェックツール (FileSystem Integrity Tool) は、ホストベースの侵入検知システムとして不正なファイルの書き換えを

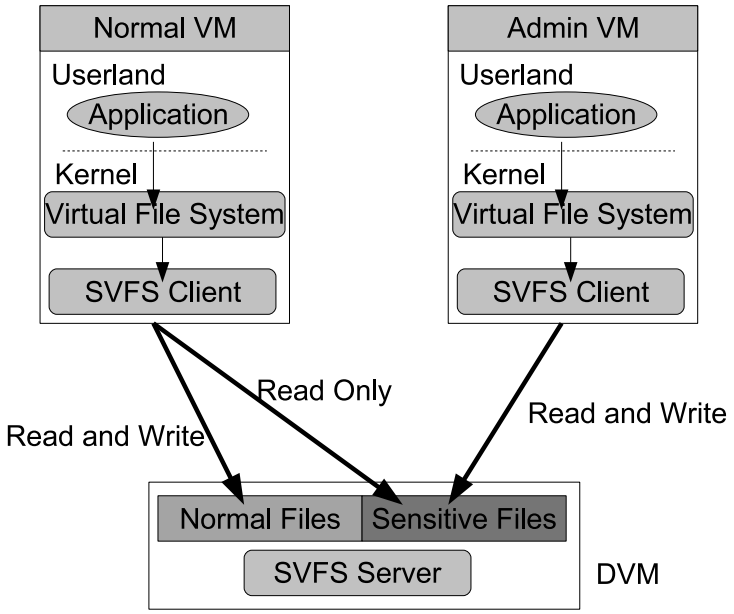


図 2.6: SVFS のアーキテクチャ

発見するのに使われる。一般的な FIT は以下のような動作をする。管理者は監視する必要のある重要ファイルのリストを作成し、FIT がそれらのファイルのデータベースを生成する。FIT は定期的にファイルとそのデータベースを参照し、ファイルの書き換えを検知してその結果をレポートする。しかし、このような検査の仕方はリアルタイムにファイルの書き換えを検知するのには不十分であり、またデータベースや FIT そのものを攻撃者から守るのは困難である。

XenRIM は Xen をベースに作られたのリアルタイム FIT である。XenRIM のアプローチではデータベースを従来の方法で更新する必要がないので導入や維持が容易である。また、Xen を利用することによって FIT 自体を安全に隔離された仮想計算機上に置くことで攻撃者からの干渉を防ぐことができる。また、XenRIM が攻撃対象の VM と別 VM 上にあるため侵入者に XenRIM の存在を気づかれることがない。

XenRIM では従来の FIT の問題点を 7 つ挙げ、それらを解決する手法を提案している。問題点は以下の 7 つである。

- 検知の遅延
  定期実行による検査では侵入者に攻撃をする時間を与えてしまう。
- 導入の困難さ
  ISP やデータセンターのような数百のマシンを管理する場合、全て

のマシンに対してデータベースを作らなくてはならない、さらにそれらのデータベースを安全に管理することが難しい。

- 維持のオーバーヘッド
  システムが変更されたとき、データベースの更新を逐次行わなければならない。ソフトウェアの修正パッチの適用やアップデートは頻繁に行われるためデータベース更新のコストが大きい。

- パフォーマンスへの影響
  FIT は通常、ファイルシステムのシグネチャのハッシュ値の計算を行うが、数億のファイルを持つシステムではこの計算時間が大きくパフォーマンスが低下してしまう。

- FIT 自体が攻撃される危険
  ほとんどの FIT はユーザ空間で動作するアプリケーションであるため、攻撃者が管理者権限を取得した場合には FIT プロセスが無効化されてしまう。

- システムの露出
  攻撃者から容易に、システムが FIT で監視されていることが気づかれてしまう。

XenRIM の設計と実装

XenRIM では前述の問題を解決する手法を提案している。従来の定期実行をする FIT ではなく、XenRIM はファイルシステムのイベントにフックを挿入することで、全てのファイルシステムに関するイベントをリアルタイムに知ることができる。表 2.2 はフックを挿入するイベントのリストである。

図 2.7 は XenRIM のアーキテクチャである。XenRIM は 2 つの主要なコンポーネントからなる。ドメイン U で動作するイベントをキャプチャするツールである XenRIMU とドメイン 0 で動作しているイベントを記録する xenrimd である。XenRIMU は Linux Security Module として実装される。XenRIMU と xenrimd は同一のマシン上で動作するので共有メモリを通じてデータのやりとりを行う。xenrimd と XenRIMU は互いにイベントチャンネルポートを使ってデータを記録したメモリアドレス等の通知を行う。

XenRIM をカーネルモジュールとして実装することで、異なるバージョンのカーネルに対し、高い互換性とツールの維持のコストが改善されている。また XenRIM は従来の FIT と違いユーザ空間で動作するデーモンプ

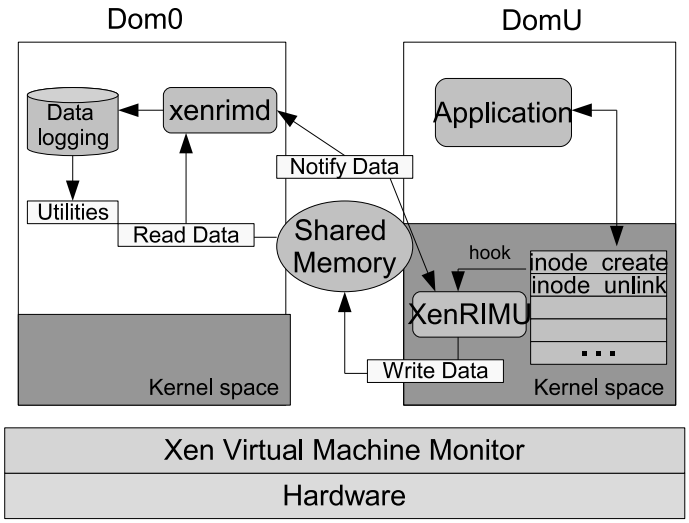


図 2.7: XenRIM のアーキテクチャ

ロセスではないため、攻撃者にシステムを監視していることを気づかれにくく、他の FIT と同様の方法で停止させることはできない。カーネルスペースから外部に情報を受け渡すために、Xen の機能であるドメイン間の共有メモリスペースを利用している。この方法はネットワークを利用しないため、ネットワークのトラフィックから XenRIM の存在を知ることはできない。XenRIM を動作させるための仮想計算機を作るだけでよいので導入が簡単である。データベースを持たないためデータベースの更新や維持のコストが削減されている。read や write のような最も頻繁に使われるイベントに対してフックを挿入しないためパフォーマンスの低下が少ない、またフックの挿入により何時イベントが発生したか、誰によって発行されたイベントなのかという価値のある情報を取ることができる。XenRIM はクライアント・サーバー方式を取っているために、複数のドメインの監視をサーバで一括管理できるという利点がある。

しかし、XenRIM ではファイルの不正アクセス自体は防げず、管理者がファイルアクセスのログを見る、あるいは XenRIM からの警告を見て不正アクセスが行われたか判断しなければならない。SAccessor では攻撃者からの不正アクセスを防ぐ、また攻撃の被害を限定するような仕組みを提供している。

Hook name	**Gathered information**
inode_create	ファイルの作成
inode_mkdir	ディレクトリの作成
inode_rmdir	ディレクトリの消去
inode_link	ハードリンクの作成
inode_sy;mlink	ソフトリンクの作成
inode_unlink	ファイルの削除
inode_rename	ファイル名の変更
inode_setattr	ファイルのパーミッションの変更
inode_mknod	デバイスファイルの作成

表 2.2: XenRIM でフックが挿入されるイベントのリスト

2.7　S4

攻撃者による侵入を検知した後には、システムの管理者はいくつかやるべき仕事がある。攻撃による被害を特定し、システムを元の安全な状態に戻すこと。攻撃には機密情報の漏洩や、バックドア・トロイの木馬の設置、保存されているデータの侵害などがある。これらを特定することは困難である。なぜならば攻撃者はログを消して侵入の痕跡を消したり、侵入検知システムを無効化しようとするからである。システムの回復にはシステムをバックアップから復元したり、あるいは OS の再インストールをする場合もある。これらの作業にはしばしば大きな時間が費やされ、システムの可用性を著しく落としてしまったり、最後のバックアップと侵入までの間のデータを復元できない等の問題がある。また、攻撃された弱点の発見も行わなければならない。弱点を見つけて防がなければシステムが復旧した後、再び同じ方法で侵入を許してしまうためである。

Self-Securing なストレージは、検知不可能な改ざんや、データの永久的な削除を防ぐことでこの問題に対して部分的に解決することができる。Self-Securing なストレージの一番の利点は侵入者が OS を攻撃したとしてもストレージによるバージョニングをバイパスできないことである。

S4 はログストラクチャとジャーナルベースのメタデータを合成することで、パフォーマンスに与える影響を小さく、バージョニングを包括的に行えるようにした Self-Securing なストレージシステムである。S4 はストレージデバイスへのリクエストの監視やファイルのバージョニングを行う。S4 は侵入後の管理者によるシステムの診断や回復の作業をアシストするシステムである。管理者は S4 を使うことで古いバージョンのファイルやストレージデバイスへのリクエストの管理ログを閲覧することがで

き、従来よりもシステムの回復や診断を簡単に行うことができる。

S4 はファイルに変更があるごとに新しいバージョンのファイルを作成する。バージョン化されたファイルは管理者の定める detection window と呼ばれる期間の間保存され、detection window を過ぎると破棄される。バージョン化されたファイルはアクセスログと一緒に history pool と呼ばれる領域に保存される。これらの情報をシステムの回復や診断に利用することで、作業が容易になる。

- 診断
  侵入者がログファイルを改ざんした場合であっても、history pool に保存されたログをたどることで侵入者により改ざんされる前のログを復元して調べることができるため、侵入の検知が容易になる。また S4 のリクエストのログを調べれば侵入者が直接アクセスしたファイルを特定することができる。

- 回復
  S4 により侵入者が直接アクセスしたファイルが特定できるため、個々のファイルを復元することで OS の再インストールせずに攻撃の影響を取り除くことができる。ファイルをバックアップから復元する方法もあるが、これはスタティックなファイルにしか適用できない。一方 S4 では任意のファイルに対して復元が可能である。またバックアップではなく history pool からのファイルの復元により、侵入者によってファイルを改ざんされる直前の状態に戻すことが可能である。

これらの利点は S4 の「侵入者がファイルのバージョニングを回避できない」という性質に基づくものである。しかし、S4 ではファイルの改ざん自体を防ぐことができない。一方 SAccessor はファイルの不正アクセスを防御の対象としている。また S4 ではファイルのバージョニングのための保存領域が必要になる。

2.8　既存技術の問題

この節では、これまでに取り上げた既存技術の問題について述べる。SELinux などのセキュア OS はカーネルの一部として実装される。セキュア OS では管理者の権限を分割し攻撃者に管理者権限を奪われたときの被害を限定することや、プロセス毎に異なるアクセス権を与えて攻撃を無力化することができる。例えばあるプロセスに対してネットワークリソースへのアクセス権を制限することで、そのプロセスの制御が奪われたとしてもネットワークを介しての情報漏えいや、別マシンへのリモート攻撃を

防ぐことができる。しかし OS に脆弱性が存在していると、バッファオーバーフローなどの攻撃により、OS レベルの特権で任意のコードを実行され、任意のリソースにアクセスされてしまう危険がある。

VM を用いて、それぞれのプロセスを別々の VM 上で動かすことで攻撃を受けた場合の影響を VM 内に限定するという使い方も考えられる。例えば、ssh とメールクライアントを別々の VM で動作させることでウィルスメールから秘密鍵を守ることができる。しかし、ユーザは複数の VM にログインしなければならない。また、プロセス間のデータの共有が困難である。

Kerberos 認証を用いれば認証に有効期間をつけることで攻撃の機会を制限することができる。しかし、クライアントが攻撃されキーロガーなどによりパスワード等が盗まれた場合には、攻撃者が認証を成功させ自由にリソースにアクセスされてしまう。

Proxos を使うことでプロセスを安全に実行することが可能である。しかし、Proxos を使用するためにはアプリケーションのプログラマがシステムコールレベルのポリシを書かなければならず、一般ユーザでは導入が難しい。また、プロセス毎に VM を作成しなければならず利便性が低いと言える。

SVFS は VM とファイルサーバを使い、アクセス制御を攻撃対象の OS から分離する。また VM ごとに異なるアクセス権を与え、システムの重要ファイルの改ざんを防ぐシステムである。VM とファイルサーバを使用することで、OS がクラックされた場合にもアクセス制御を強制することができる。通常ログインする常用 VM では重要ファイルへの書き換えはできないため、常用 VM 上の OS をクラックされても、重要ファイルを改ざんすることはできない。一方で、正規ユーザであっても常用 OS からは重要ファイルの書き換えはできないため、重要ファイルを書き換えるような場合には、重要ファイルへの書き込み権限のある別の VM にログインしなおさなければならないため利便性が低い。

XenRIM は VM を用いて、ファイルの完全性チェック機構を攻撃対象の OS から分離している。そのため、OS がクラックされたとしても無効化されることはない。また、攻撃者が XenRIM の存在に気づくのも難しい。しかし、XenRIM ではファイルアクセスのログをとるのみであり、管理者がログを解析しなければならない。

S4 はストレージベースの侵入検知システムで攻撃と思われるファイルアクセスに対しては、管理者に警告を出す、帯域を制限するなどに留まっており、正規ユーザによるファイルアクセスと攻撃によるファイルアクセスを区別して制御することはできていない。

このように現在の既存技術ではファイルアクセス制御を攻撃対象の OS

から分離し、実用的に使えるものはない。本研究では、VM を用いてファイルアクセス制御を OS から分離した上で、利便性と安全性を両立したシステムである SAccessor の開発した。その設計と実装は 3 章以降で述べる。

第3章　SAccessor

我々はアクセス制御を攻撃対象の OS から分離したときに生じる問題を解決し、実用的にアクセス制御を行える SAccessor を提案する。SAccessor は仮想計算機を用いてファイルアクセス制御をユーザのログインする OS から分離する。SAccessor を使えばユーザのログインする OS がクラックされた状況においても、システムファイルの不正な書き換えやの防御やユーザファイルへの攻撃の被害を限定することができる。SAccessor の特徴は３つで以下である。

- VM とファイルサーバを用いてファイルアクセス制御を分離した
- システムファイルとユーザファイルの扱いの違いに着目して、それぞれに適切なアクセス制御を行う
- ユーザを安全に認証する機構を提供している

3.1　アクセス制御の分離

SAccessor は VM を用いてファイルアクセス制御を隔離して安全に実行する。SAccessor は同一ホスト上に、ユーザがログインする作業 OS とファイルアクセス制御を行う認証 OS を動作させ、作業 OS での従来の OS でのファイルアクセス制御とは別に、認証 OS でも独立してファイルアクセス制御を行う。これにより、作業 OS の制御を奪われてもファイルアクセス制御を機能させることができる。

図 3.1 は SAccessor のアーキテクチャである。VMM を使用し、認証 OS、作業 OS の２つの OS を動作させる。認証 OS がファイルサーバになる。作業 OS はローカルファイルシステムを持たず、認証 OS 上に置かれた作業 OS 用のディレクトリをルートディレクトリに NFS マウントして利用する。ファイルサーバを使うことで、作業 OS から認証 OS のディスクにファイル名等の高い抽象度でのアクセスが可能である。作業 OS はこれ以外の方法ではディスクにアクセスすることはできない。

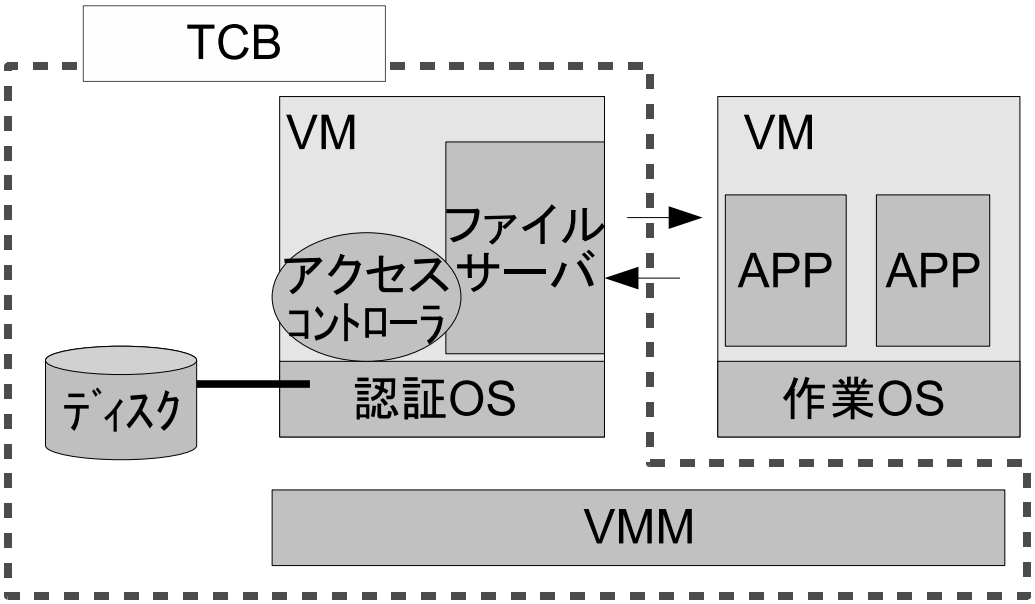


図 3.1: SAccessor のアーキテクチャ

3.2　システムファイルのアクセス制御

3.2.1　認証 OS によるサービス提供

システムファイルの不正な書き換えを防ぐために、作業 OS からシステムファイルへのアクセスは一部の例外を除き読み込みのみを許可する。ここでいうシステムファイルとはインストールされたプログラム、ライブラリ、設定ファイルである。例外的に作業 OS による書き込みを許可するシステムファイルは、/var/run/xxx.pid 等の PID を記録するファイルや、ログファイル、/var/run/utmp など、システムによって自動で書き込みが行われるようなファイルである。ログファイルは追記のみを許可する。追記のみであるため、ログファイルが改ざんされることはなく、侵入の痕跡を消されることはない。pid 等は攻撃者によって書き換えられる恐れがあるが、システムを再起動した後には上書きされるため影響は残らない。

一般ユーザはシステムファイルを setuid プログラムを使って書き換える。認証 OS は setuid プログラム単位で設定ファイルを書き換えるためのサービスを提供し、作業 OS からはそれらのサービスを通してのみシステムファイルの書き換えができる。作業 OS が直接システムファイルを書き換えることはできない。よって、たとえ作業 OS がクラックされても、作業 OS を再起動してやることでクラックの影響を取り除くことができる。

認証 OS の提供するサービスの例として passwd コマンドがある。SAccessor では実行するプログラムが setuid されていると認証 OS と通信し、

ユーザを認証してから認証 OS 上の対応するサービスの起動を行う。passwd コマンドの場合は、認証 OS 上の passwd プログラムを呼び出す。認証 OS 上の passwd プロセスはユーザから受け取ったデータをもとに/etc/passwd ファイルの書き換えを行う。このファイルは作業 OS からは読み込みのみに設定されているので、作業 OS が直接このファイルを書き換えることはできない。

3.2.2　認証ダイアログ

SAccessor では安全に認証を行うために、サービスの起動時に認証 OS の認証プロセスが認証ダイアログを通して直接ユーザとやり取りする。図 3.2 は認証ダイアログのスクリーンショットである。ユーザ名とパスワードを入力させて認証を行う。認証ダイアログ中央には実行する認証 OS のサービスが表示されており、ユーザの意図しないサービスの実行である場合には DENY ボタンをおして認証を失敗させることができる。このダイアログは setuid プログラムの実行時のみ表示される、ダイアログが出る頻度は低い。

この認証ダイアログは作業 OS の画面とは独立した認証 OS のウィンドウとして、作業 OS の画面の上に重ねて表示する。VNC とはネットワーク上の離れたコンピュータを遠隔操作するためのソフトである。VNC はサーバと、クライアントの 2 つに分かれている。操作される側で VNC サーバを起動しておき、遠隔操作する側は VNC クライアントで接続し、遠隔操作を行う。作業 OS 上では VNC サーバが動作しており、認証 OS で VNC クライアントを動作させる。まず認証 OS の画面をディスプレイ全体に表示し、その中の VNC のウィンドウを全画面表示させる。ユーザからは VNC のウィンドウだけが見えている状態になる。認証 OS のウィンドウとして認証ダイアログを VNC のウィンドウの上になるように表示する。このようにすると、ユーザには認証ダイアログがあたかも作業 OS から出されたように見える。そのため、ユーザは自分のログインする作業 OS の裏で認証が動いていることを意識しないシームレスな認証が可能である。

また、認証 OS のウィンドウである認証ダイアログには作業 OS にいる攻撃者はアクセスすることも、ダイアログを盗み見ることもできない。ユーザと認証 OS が認証ダイアログを通して作業 OS を介さずに認証を行うことで安全に認証をすることができる。ダイアログに渡されるキー入力は、キーボードから VMM、認証 OS と渡される。これらは TCB に含まれるので、このキー入力は作業 OS の攻撃者に盗まれることはなく安全に認証 OS 上のサービスに受け渡すことができる。

認証ダイアログは作業 OS の画面上に重ねて表示するため、作業 OS に侵入した攻撃者に偽のダイアログを出された場合、本物の認証ダイアログと

図 3.2: 認証ダイアログ

区別するのが難しい。攻撃者によって出された偽のダイアログにパスワードを入力してしまうと、攻撃者にパスワードを盗まれてしまう。SAccessor は攻撃者が出す偽の認証ダイアログと区別するために、認証ダイアログのタイトルにはシークレット文字列を表示する。シークレット文字列はあらかじめ認証 OS 側にユーザが登録しておくものであり、作業 OS からはみることができない。そのため、攻撃者はこのシークレット文字列を偽のダイアログの表示することが難しい。ユーザはシークレット文字列を見ることで認証ダイアログを偽のダイアログと区別することができる。

3.2.3　安全な入出力

認証 OS のサービスの入出力を安全に行えるようにするために、認証 OS のターミナルソフトの中でサービスを動かす。このターミナルソフトのウィンドウは認証ダイアログと同様に作業 OS の VNC ウィンドウの上に表示される。標準入出力はこのターミナルを通してユーザと認証 OS が直接キー入力などの受け渡しを行う。これは、作業 OS 経由でキー入力や画面出力を行ってしまうとユーザが起動した setuid プログラムに対して攻撃者がキー入力したり、出力を盗み見される危険があるからである。例えばパスワード変更の操作を行っているとき、攻撃者にキー入力を盗まれるとパスワードを知られてしまう。また、キー入力に割り込まれるとユーザの意図しないパスワードに変更されてしまう。setuid されたプログラムが GUI プログラムの場合には、そのウィンドウを作業 OS の画面の上に表示する。

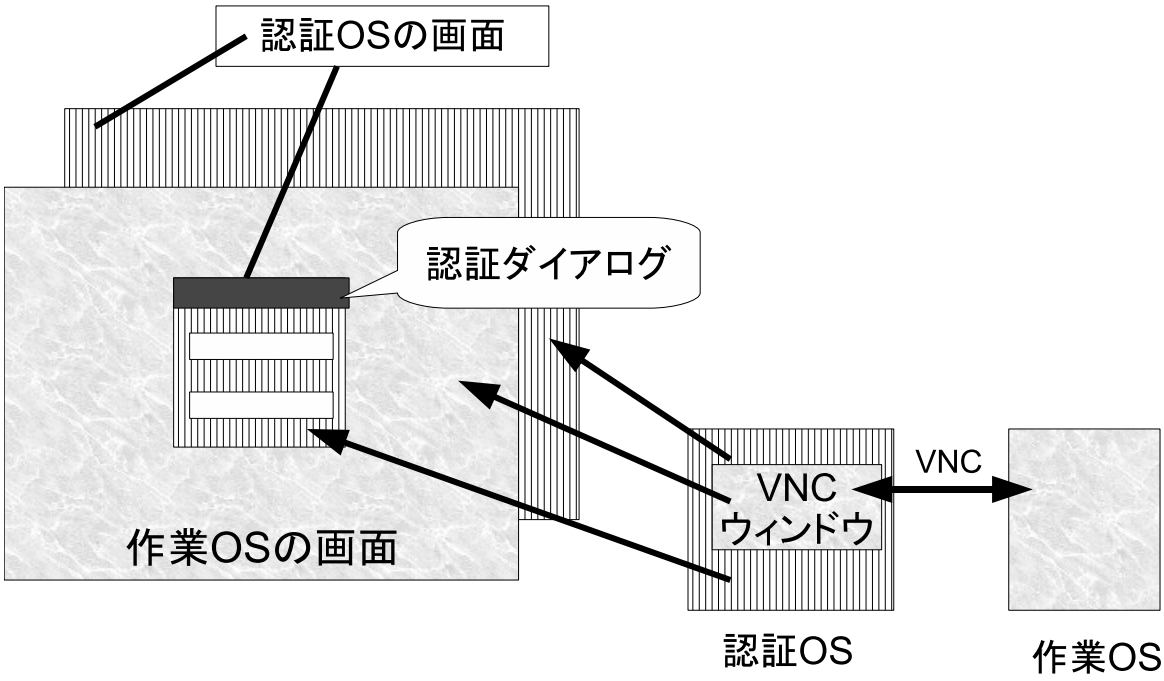


図 3.3: 画面構成

3.3　ユーザファイルのアクセス制御

システムファイルにアクセスする setuid プログラムは最新の注意を払って作成されているため、認証 OS 側で安全に実行できる。しかし、ユーザファイルにアクセスする一般のプログラムを認証 OS で動かすのは危険である。例えばメーラのサービスを認証 OS に用意して認証 OS 上で実行すると、ウィルスメールを開いた場合に悪意のあるコードが認証 OS 上で実行され、任意のファイルにアクセスされてしまうからである。

そこで、ユーザファイルに対するアクセスに対しては、作業 OS で動いているプログラムがファイルにアクセスしたときに認証ダイアログを表示して認証を行う。ダイアログの中央にはプログラム名の代わりに、アクセスしようとしているファイル名とアクセス権が表示される。しかし、ファイルアクセスの毎に認証ダイアログによる認証を用いると実用的にならない。なぜなら、ファイルアクセス毎に認証ダイアログを表示すると著しく利便性が落ちてしまうからである。その一方で認証をログインのときにしか求めないようにするとログイン中に作業 OS の制御を奪われたときに、それ以降は任意のファイルにアクセスされてしまう。

```
⟨Mailer⟩ [3600]
/home/takizawa/.thunderbird/* (rw)
⟨/Mailer⟩

⟨ssh⟩[10]
/home/takizawa/.ssh/id_rsa (r)
⟨/ssh⟩
```

図 3.4: SAccessor のポリシ例

3.3.1　認証の有効範囲の限定

SAccessor では、ファイルやディレクトリをまとめたグループ単位でアクセス制御を行う。ポリシファイルにグループを定義することができ、それぞれのグループに対して許可するアクセス権と認証の有効期間を設定する。一度グループに対して認証を行ったら、認証の有効期間内ではグループのファイルへのアクセスに対する認証は省略され、認証の回数を減らすことができる。また、ログイン時のみに認証する場合と比べると、よりきめ細かいアクセス制御が可能になる。たとえ認証を行ったあとで作業 OS の制御を奪われたとしても、攻撃者は認証が有効になっているグループのファイルにしかアクセスできず、攻撃による被害を限定することができる。このように SAccessor では実用性と安全性のバランスを取ることができる。

図 3.4 は SAccessor のポリシ例である。タグで囲まれた範囲が一つのグループになり、その中で羅列されるファイルのパス名がグループに属するファイルになる。それぞれのパス名の後ろには許可したいアクセス権を記述する。r、w はそれぞれ読み込みと書き込みを表している。a は追記を表す。グループタグの隣に書かれた数字は認証の有効期間（秒）を表している。有効期間を省略すると、そのグループに対しては認証は行わずアクセス制御のみを行う。

図 3.4 のポリシを適用した場合、ユーザがメールを見るためにメーラを起動しようと認証を行うと *Mailer* グループに属する.thunderbird ディレクトリ以下のファイルへのアクセスが可能になる。このときもし、ウィルスを添付されたメールを受信し、ウィルスプログラムがユーザの秘密鍵 *id_rsa* へアクセスしようとすると、*ssh* グループのための認証のために認証ダイアログが表示される。ウィルスはこの認証に成功させることができず、ユーザがこの認証を失敗させてやることで *id_rsa* へのアクセスを防ぐことができる。

3.3.2　ファイルキャッシュの制御

作業 OS が認証 OS のファイルサーバから取得したファイルの内容はファイルキャッシュとして作業 OS 上のメモリに保持され、その後、ファイルキャッシュが有効な間のファイルアクセスはそのファイルキャッシュに対して行われる。作業 OS にファイルキャッシュが残っていると、作業 OS は認証 OS と通信を行わずローカルのキャッシュに対してアクセスするために、認証の有効期間が終わっても認証を行わずにファイルを読み続けることができてしまう。

例えば図 3.4 において、初めて *id_rsa* にアクセスするときには認証を求められる。認証に成功すると実際にファイルを読み込み、作業 OS のメモリにキャッシュする。認証の有効期間である 10 秒が過ぎた後に再び *id_rsa* を読み込もうとした場合、認証 OS と通信して認証を行うべきだが、作業 OS にファイルキャッシュが残っている場合には認証 OS とは通信をせず作業 OS のファイルキャッシュに直接アクセスしてしまう。そのため認証の強制ができず、攻撃者に *id_rsa* ファイルを読み込まれる危険性が大きくなってしまう。SAccessor ではこの問題を避けるために、認証の有効期間の切れたファイルは認証 OS が作業 OS 上のメモリ上から強制的にクリアする。認証 OS は VMM の機能を使って作業 OS のメモリからクリアすべきキャッシュを探す。認証 OS が作業 OS のメモリ操作を行うので、作業 OS の改造は不要である。これにより作業 OS がクラックされる前に認証の有効期限が切れたファイルキャッシュの読み込みを防ぐことができる。作業 OS がクラックされて OS の構造を変えられた場合は、作業 OS 上にあるファイルキャッシュはもともとファイルから読み込めるデータであるので、ファイルキャッシュをフラッシュできないことによる安全性の低下はない。

SAccessor によって、作業 OS に書き込み権限のないファイルの改ざんは防ぐことをできる。そのため、攻撃者はファイルではなく作業 OS 上のファイルキャッシュを書き換えてファイル改ざんと同じ効果の攻撃をせざる得ない。しかし、SAccessor によってファイルの書き換えを防いでいるためファイルキャッシュの改ざんによる攻撃は OS を再起動することでその影響を取り除くことができる。

3.3.3　制限

ユーザが認証を成功させると作業 OS に対して認証の有効範囲内でのアクセス権が与えられ、認証の有効期間が切れるまでは認証アクセスできる。従って、作業 OS の制御が奪われる前に認証し、その認証が有効な

ファイルを攻撃者から守ることはできない。また、作業 OS の制御が奪われた後に認証したファイルも守ることはできない。

第4章　実装

我々はSAccessorのプロトタイプの実装を行った。この章ではSAccessorの実装に関して、アクセス制御、setuidプログラムの実行、ファイルキャッシュクリアの方法について述べる。

4.0.4　アクセス制御

SAccessorの実装は図4.1のようになっている。VMMとしてはXen[1]を使用し、認証OSをドメイン0で、作業OSをドメインUで動作させる。両OSにはLinuxを使用している。図のアクセスコントローラ、デバイスファイル、認証プロセスがSAccessorのために追加したプログラムである。ファイルサーバにはNFSを使用した。またNFSリクエストをアクセスコントローラで制御するようにNFSDを改造した。アクセスコントローラはポリシに従ってアクセス制御を行う。認証プロセスは認証ダイアログを出してユーザ認証を行うプログラムであり、アクセスコントローラとデバイスファイルを介して通信を行う。

ポリシーと認証ダイアログに表示するシークレット文字列はSAccessorの起動時に登録を行う。アクセスコントローラはNFSDが受け取ったリクエストを監視し、アクセス権のチェックを行う。認証を行う必要があるときには認証プロセスと通信し、認証ダイアログを出すように命令する。認証プロセスは認証ダイアログを表示して、ユーザ認証を行い、結果をアクセスコントローラに返す。認証に成功、あるいは認証が必要ない場合にはアクセスコントローラはそのファイルに対するinode構造体に認証済みを表すフラグを真にする。フラグが真になっているファイルに対する認証は省略される。このフラグは認証の有効期間が切れると偽にされる。

SAccessorは追記のみを許可するポリシーを提供している。しかし、NFSは追記がわからない。ファイルアクセスが追記かどうかは、ファイルサイズと書き込みのオフセットを比較して判断する。オフセットがファイルサイズに等しければ追記とする。

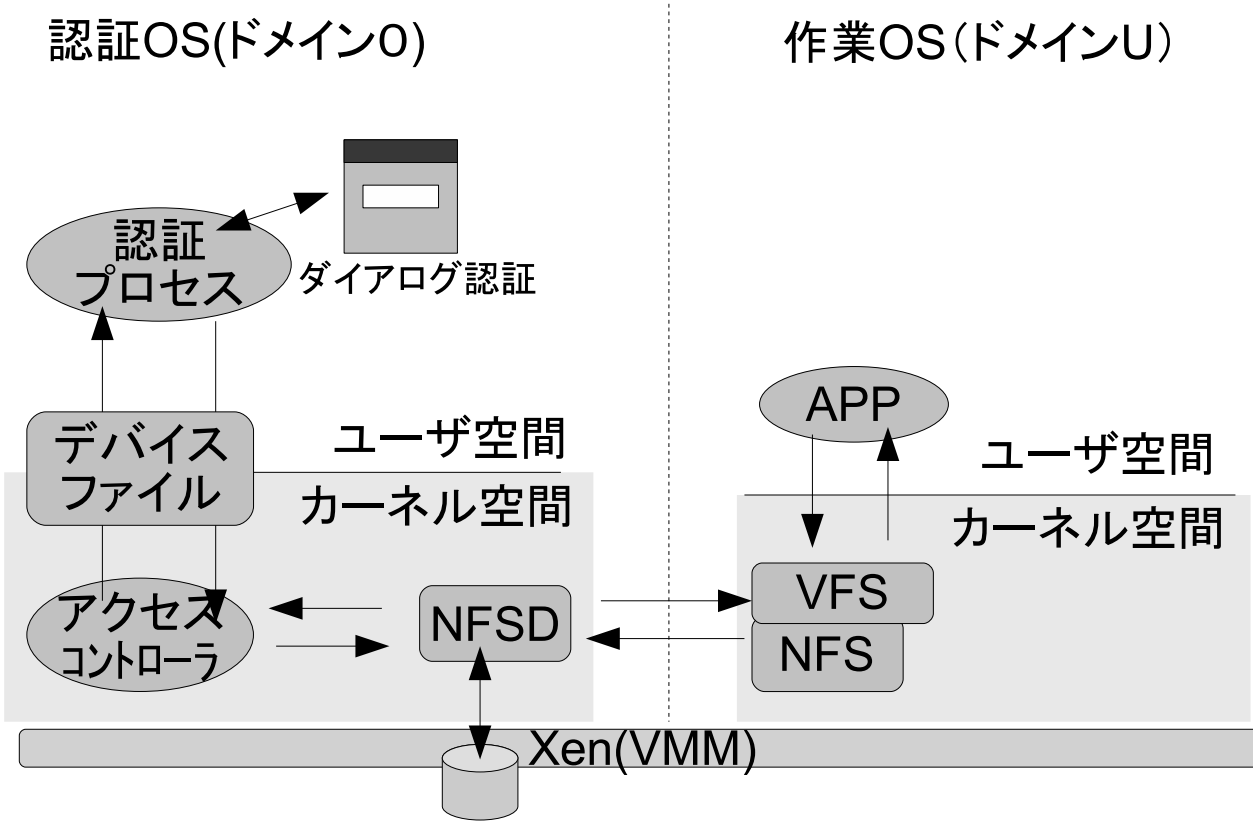


図 4.1: SAccessor の実装

4.0.5　setuid プログラムの自動転送

図 4.2 は setuid プログラムの実行の手順を示している。SAccessor では Glibc の execve 関数を改造した。我々の改造した execve 内では setuid されたプログラムかどうかを stat 関数で識別し、自動で認証 OS と通信を行い、認証プロセスが認証ダイアログによる認証を行う。認証に成功すると、認証プロセスによって認証 OS 上の setuid プログラムが起動される。

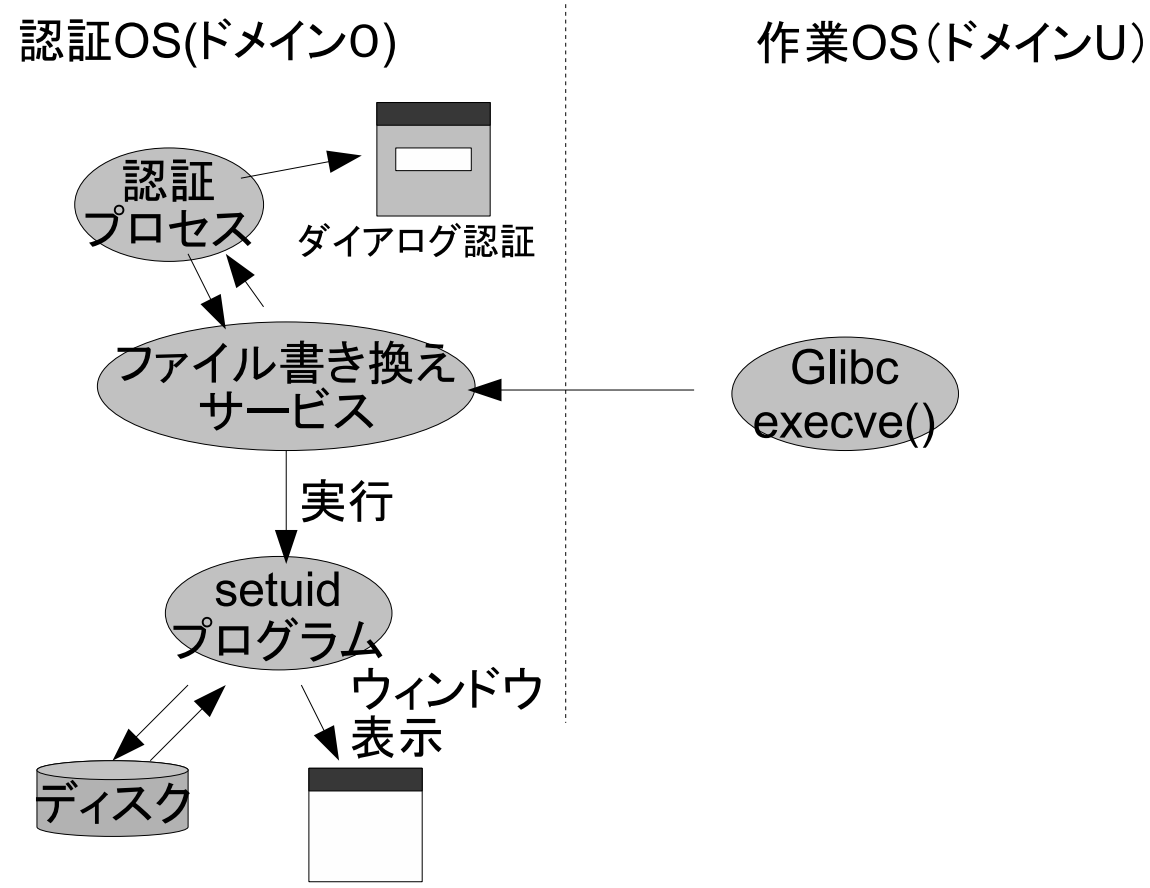


図 4.2: setuid プログラムの実行

4.0.6　作業 OS のファイルキャッシュクリア

認証 OS 上で Xen のハイパーコールを使用して、作業 OS のメモリ空間の一部を認証 OS のメモリ上にマップし、認証 OS から作業 OS のメモリを直接操作することでファイルキャッシュをクリアする。認証 OS から作業 OS のメモリを操作する間は Xen のハイパーコールによって作業 OS

を停止させる。これは作業 OS と認証 OS でファイルキャッシュに対するアクセスの競合を避けるためである。また、認証 OS から作業 OS のメモリを操作するときにはポインタをたどる度にアドレスを調べてメモリマップを行わなければならない。これは、Xen のハイパーコールではメモリをページ単位でマップしており、ポインタの指すアドレスが同一ページにあるとは限らないためである。作業 OS 上のファイルキャッシュは、ファイルパスと作業 OS のルートディレクトリの dentry 構造体からたどることができる（図 4.3）。dentry 構造体はファイルの名前とディレクトリやファイルの相互の参照関係を管理する構造体である。まず、Xen のハイパーコールを使い作業 OS の CPU レジスタの値を取得する。x86 アーキテクチャでは CPU のレジスタの値から現在実行されているプロセスに対応する task_struct 構造体のアドレスがわかる。task_struct 構造体のメンバ fs_struct からルートディレクトリの dentry 構造体を取得し、ファイルパスを元にディレクトリを順に巡回することでキャッシュクリアの対象となるファイルの nfs_inode 構造体を見つけることができる。fs_struct はプロセスの使用するファイルシステムの情報を持つ構造体である。nfs_inode からはファイルキャッシュのアドレスを知ることができるのでキャッシュの内容をゼロクリアする。また nfs_inode にはキャッシュが有効かを示すフラグが存在するので、そのフラグの値を書き換えてキャッシュを無効化する。これにより NFS はゼロクリアされたキャッシュを使わず、サーバからファイルを読み直す。

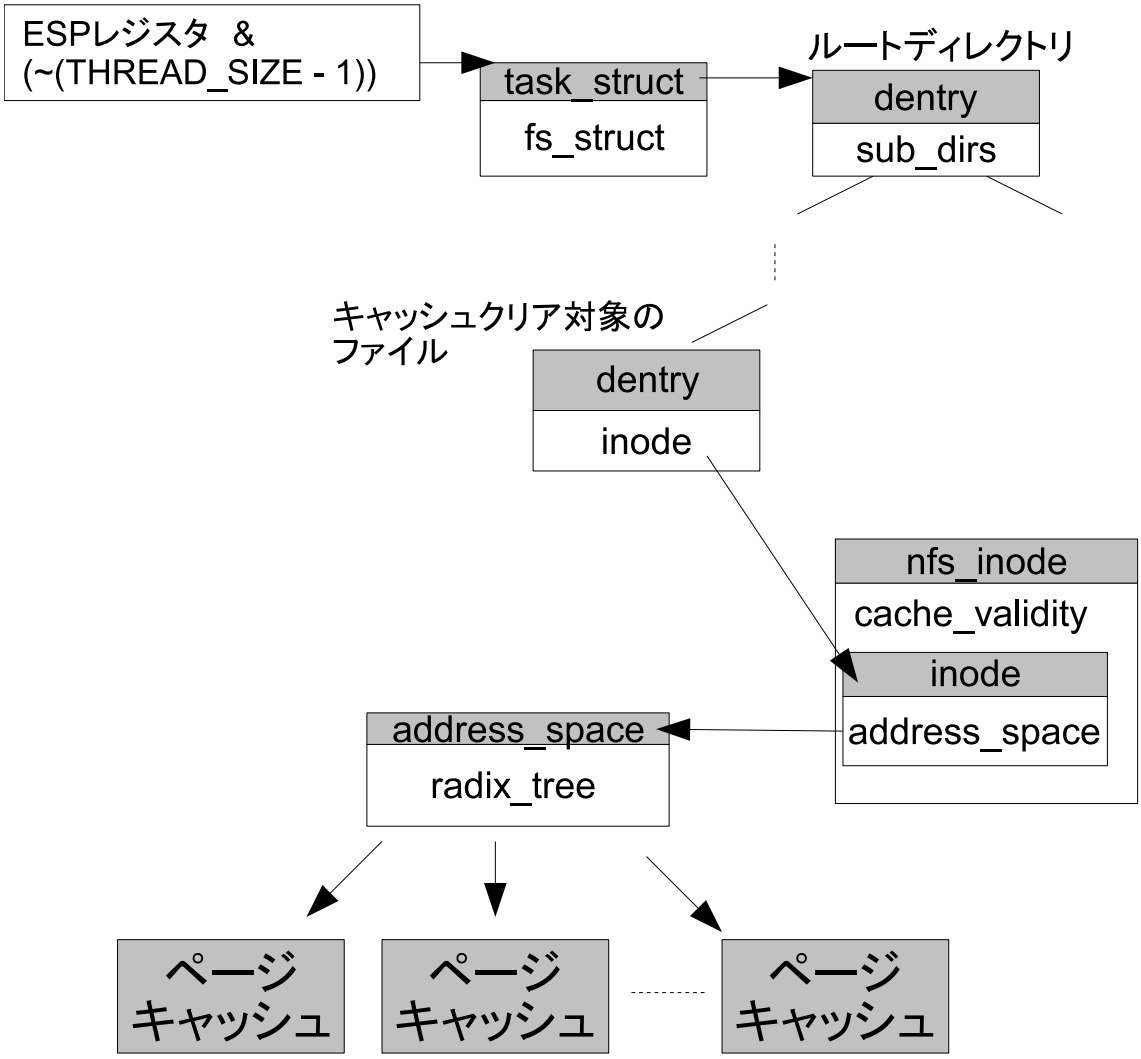


図 4.3: ファイルキャッシュのクリア

第5章　実験

我々はSAccessorを使用したときにどの程度のオーバーヘッドが生じるかを測定する実験を行った。実験対象の計算機として、PentiumD 3.0GHzのCPU, メモリ 1GBの計算機を使用した。VMMとしてはXen 3.0.4、その上で動くOSにはLinux 2.6.16.33を用いた。認証OSを動かすVMにはメモリを512MB、作業OSを動かすVMにはメモリを384MB割り当てた。

5.1　ファイルアクセスの性能

ベンチマークソフトbonnie++[2]を用いて読み込みと書き込みのスループットの測定を行った。ポリシとしては1000行のポリシを使用した。この実験は(1)VMを用いずにローカルでext3ファイルシステムを使用した場合(native)、(2)VM間でNFSを用いた場合(VM+NFS)、(3)SAccessorを使用した場合について行った。ファイルサイズとしては2Gバイトを指定し、writeバッファを使用する場合と、-bオプションを指定して使用しない場合を測定した。文字単位のI/Oではputc,getcを使い、ブロック単位の場合はwrite、readシステムコールを用いて測定を行っている。

実験結果は図5.1、図5.2、図5.3のようになった。縦軸はファイルI/Oのスループットを表す。SAccessorとnativeを比較した場合、最悪の場合は文字単位の読み込みのときで、約33%、のオーバーヘッドとなった。nativeとVM+NFSを比較した場合は約26%のオーバーヘッドになることから、SAccessorのオーバヘッドのうち26%がVMMとNFSを利用していること、残りの7%がSAccessorで新たにアクセス制御を行うために追加した処理の時間であると考えられる。書き込み性能は最悪の場合は文字単位の書き込みのときであった。SAccessorとnativeを比較したときで約42%のオーバーヘッドになった。SAccessorとVM+NFSでは0.4%のオーバーヘッドであった。

ブロック単位の書き込み性能では、SAccessorの方が性能がよくなった。これはローカルファイルシステムでは、ディスクへ書き込みが終了した時点で実験が終了となるが、NFSを使った場合にはディスクにデータを書き込み終わった時点ではなく、NFSサーバに全てのデータを送った時点

で実験が終了するためと考えられる。

5.2　ファイルキャッシュクリアの性能

ファイルキャッシュのクリアに掛かる時間を調べるために、作業 OS 上で cat コマンドを使いファイルを一度読み込んだ後、認証 OS 側からファイルキャッシュのクリアを行った。キャッシュクリアのために作業 OS を停止させてから、キャッシュをクリアして作業 OS を再開させるまでの時間を測定した。クリアするファイルの数は１つである。

図 5.4 は作業 OS 上のファイルのキャッシュをクリアするのにかかる時間を測定した実験結果である。横軸がファイルサイズになる。縦軸はクリアに要した時間（秒）である。ファイルサイズに比例して必要な時間が増え、10MB では約 0.4 秒、100MB では約 3.7 秒かかった。キャッシュクリア時に作業 OS は停止するが、その間のマウスイベントやキーストロークの取りこぼしはなかった。

予備的な実験では作業 OS のメモリを認証 OS へ 1 ページ分マップするのに約 0.4 ミリ秒かかっていることがわかった。キャッシュを 1 ページ探すのに 3 回メモリマップを行う必要があり、ページサイズは 4KB であるため 100MB では約 75000 回のメモリマップを行う必要があり、これに掛かる時間は約 3 秒である。ドメイン間のメモリマップは VMM が行っているので、SAccessor のキャッシュクリアの実装を VMM の中に実装することで、キャッシュフラッシュの性能は大幅に改善されると考えている。

5.2.1　suid

我々は転送すべき setuid プログラムがどの程度あるか調査した。調査対象は FedoraCore5 である。setuid プログラムを (1) 認証 OS 上で実行する必要があるもの、(2) 任意のプログラムを実行できるため認証 OS 上で実行するべきでないもの、(3) ネットワークにアクセスするために認証 OS 上で実行するべきでないもの、(4) 作業 OS 上で実行しなければ意味がないものに分類することができた。結果は図**??**のようになった。認証 OS 上で実行すべき setuid コマンドでよく使われそうなものはパスワード変更やタスクスケジューラであり、使用頻度は少ないため認証ダイアログで認証する頻度は少なくて済むと考えられる。また、ccreds_validate も setuid されたプログラムであったが、何をするプログラムなのかわからなかった。

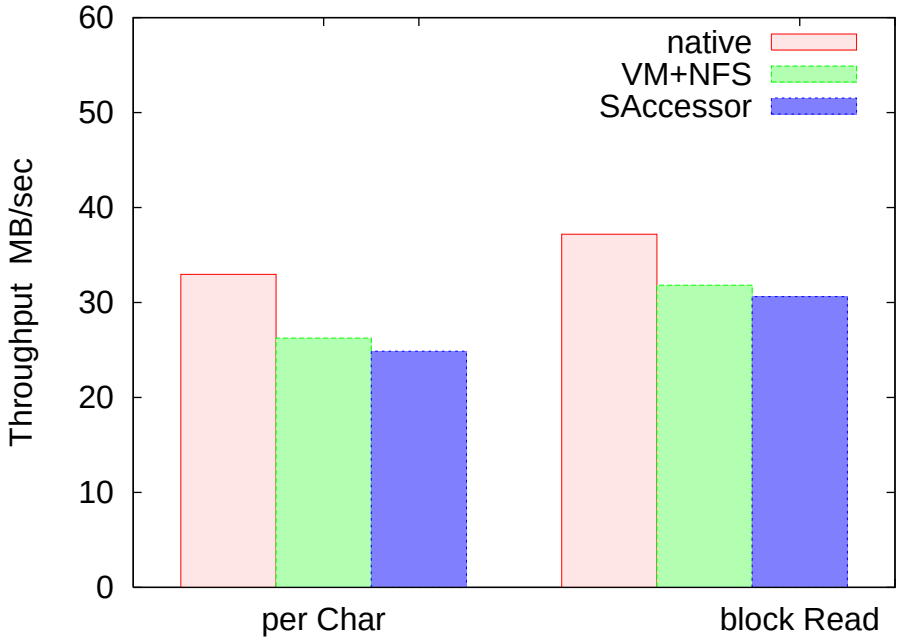


図 5.1: 読み込み性能

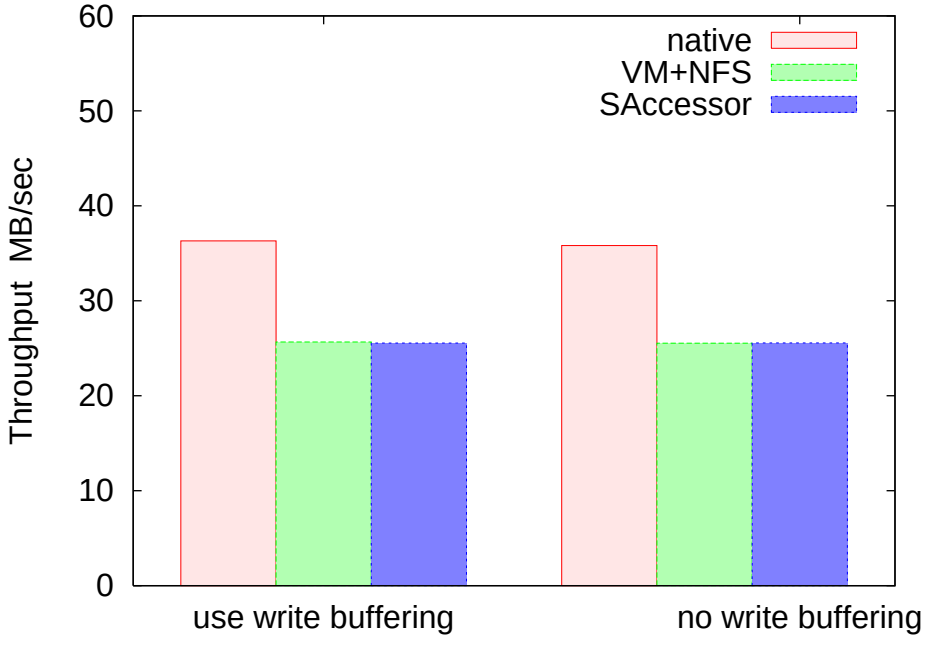


図 5.2: 書き込み性能 (文字単位)

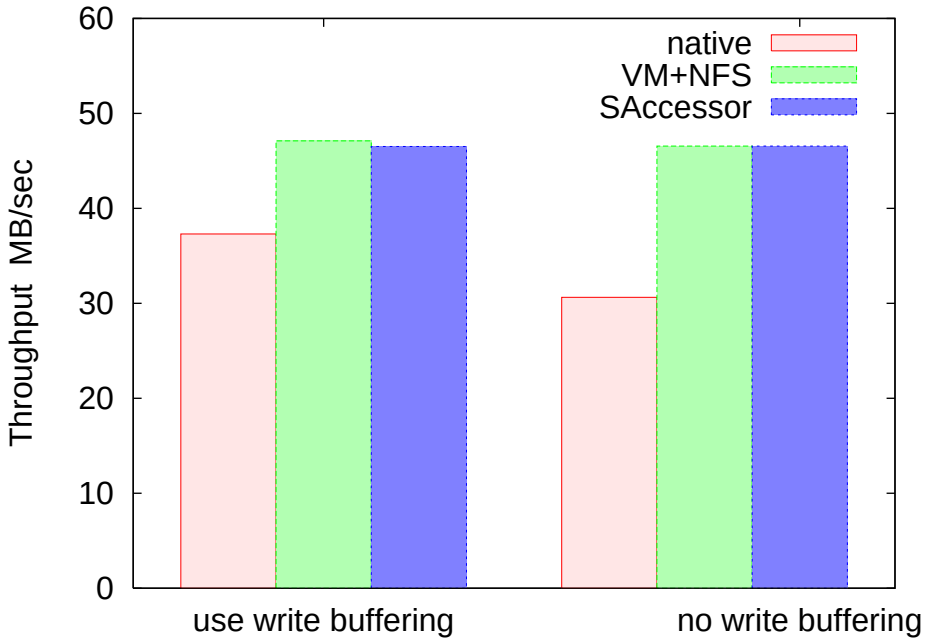


図 5.3: 書き込み性能（ブロック単位）

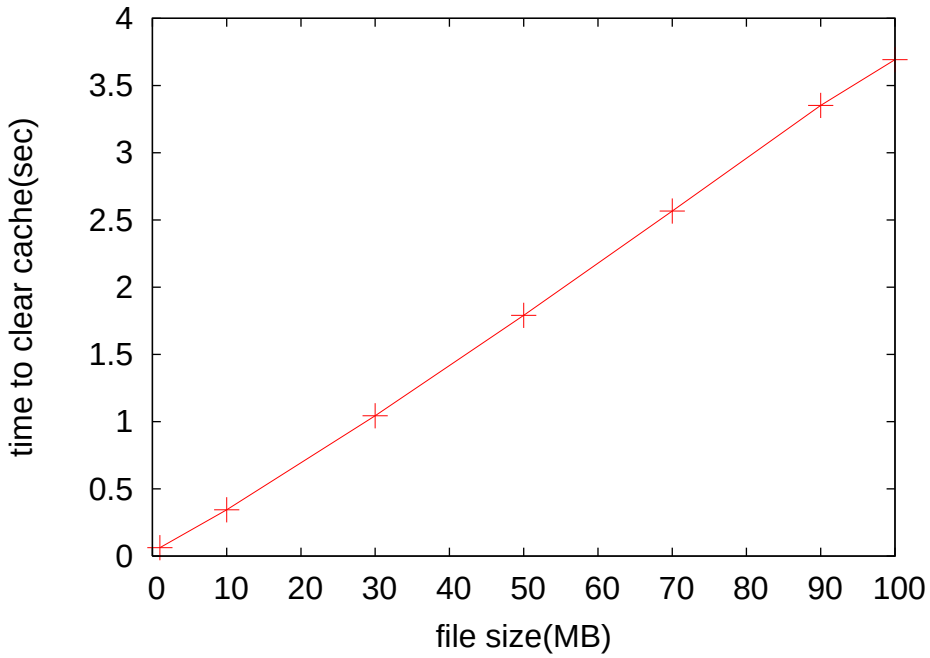


図 5.4: キャッシュクリア性能

	コマンド
認証 OS に転送する	change, at, chfn, chsh, gpasswd, crontab, passwd, lppasswd, pam_timestamp_check, unix_chkpwd, userhelper
認証 OS で動かすと任意のプログラムを実行できて危険	su, sudo, sudoedit, kon, newrole, suexec, ksu
認証 OS で動かすとネットワークを使うので危険	ping, ping6, rcp, rlogin, rsh
作業 OS で実行しないと意味がない	mount, unmount, Xorg, newgrp, kpac_dhcp_helper, newvc, usernetctl, kgrantpty, userisdnctl

表 5.1: setuid プログラムの一覧

5.3　描画性能

VNC を使うことにより、描画性能にどの程度影響がでるか調べるために x11perf[18] を用いて性能を測定した。500x500 の塗りつぶされた正方形を描画する場合 (rect500)、ピックスマップからウィンドウへ 500x500 の正方形をコピーする場合 (copy pixmap)、500x500 の正方形のイメージを画面に転送する場合 (putimage) を測定した。比較対象は X11org 7.0（native）と SAccessor において VNC を用いて作業 OS の画面を描画した場合 (SAccessor) について実験を行った。

結果は図 5.5 のようになった。縦軸は描画回数/秒である。rect500 で最悪の性能になり、native と比べて SAccessor では 172.4%のオーバーヘッドになった。putimage の場合には SAccessor が約 3 倍の性能になった。

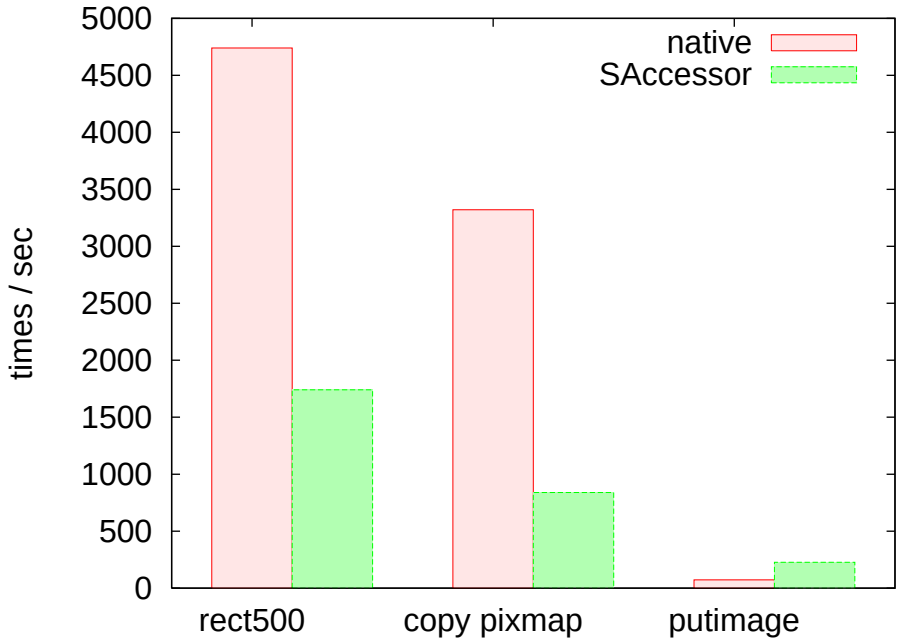

5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
times / sec
native
SAccessor
rect500
copy pixmap
putimage

図 5.5: 描画性能

第6章　まとめと今後の課題

本稿では、VM を用いてファイルアクセス制御を隔離して安全に実行する SAccessor を提案した。SAccessor は同一ホスト上で作業 OS と認証 OS を動作させ、認証 OS が認証とファイルアクセス制御を行う。認証 OS がディスプレイ上に認証ダイアログを表示し、ユーザに直接パスワードを入力させることで安全に認証を行うことができる。認証ダイアログに適したアクセス制御を行うために、作業 OS が用いるファイルシステムの拡張した。この拡張により認証の頻度を抑え、システムファイルの安全な書き換えが可能になった。

今後の課題は作業 OS 上のファイルキャッシュをゼロクリアする実装を VMM の中で行い、オーバーヘッドの削減を行うことである。

参考文献

[1] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[2] bonnie++. "`http://www.textuality.com/bonnie/intro.html`".

[3] CVE-2005-1263.

[4] CVE-2005-1750.

[5] IntelVirtualizationTechnology. `http://www.intel.com/jp/technology/virtualization/`.

[6] Kerberos. B. Clifford Neuman and Jennifer G. Steiner. Authentication of Unknown Entities on an Insecure Network of Untrusted Workstations. In Proceedings of the Usenix Workshop on Workstation Security, Portland, OR. August, 1988.

[7] LIDS. `http://www.secureos.jp/LIDS-JP/index.html`.

[8] Adam G. Pennington, John D. Strunk, John Linwood Griffin, Craig A. N. Soules, Garth R. Goodson, and Gregory R. Ganger. Storage-based intrusion detection: watching storage activity for suspicious behavior. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[9] Nguyen Anh Quynh and Yoshiyasu Takefuji. A real-time integrity monitor for xen virtual machine. In *ICNS '06: Proceedings of the International conference on Networking and Services*, Washington, DC, USA, 2006. IEEE Computer Society.

[10] Nguyen Anh Quynh and Yoshiyasu Takefuji. A novel approach for a file-system integrity monitor tool of xen virtual machine. In *ASIACCS '07: Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pages 194–202, New York, NY, USA, 2007. ACM.

[11] RSBAC. `http://www.rsbac.org/`.

[12] SELinux. `http://www.selinux.gr.jp/`.

[13] J. Steiner, C. Neuman, and J. Schiller. An authentication service for open network systems, 1988.

[14] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: protecting data in compromised system. In *OSDI'00: Proceedings of the 4th conference on Symposium on Operating System Design & Implementation*, Berkeley, CA, USA, 2000. USENIX Association.

[15] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: making trust between applications and operating systems configurable. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 279–292, Berkeley, CA, USA, 2006. USENIX Association.

[16] VirtualPC. `http://www.microsoft.com/japan/windows/virtualpc/default.mspx`.

[17] VMware. `http://www.vmware.com/jp/`.

[18] x11perf. `http://xjman.dsl.gr.jp/man/man1/x11perf.1x.html`.

[19] Xin Zhao, Kevin Borders, and Atul Prakash. Towards protecting sensitive files in a compromised system. In *SISW '05: Proceedings of the Third IEEE International Security in Storage Workshop (SISW'05)*, Washington, DC, USA, 2005. IEEE Computer Society.