平成23年度 修士論文

踏み台攻撃だけを抑制できる VMMレベルパケット・フィルタ

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 08M54012

安積 武志

指導教員

千葉 滋 教授

平成23年1月28日

概要

クラウドコンピューティングにおいて、ユーザに提供している仮想マシン（VM）からの踏み台攻撃はデータセンタにとって大きな問題である。データセンタ内 VM 二侵入されて踏み台攻撃が行われると、侵入されたホストのユーザだけでなくデータセンタの攻撃の責任を負うことになる可能性がある。しかしすべての VM 内のすべてのソフトウェアに対して、すべてのセキュリティパッチが適用されていることを保証することは困難であるため、踏み台攻撃をできるだけ早く止めることが重要になる。ファイアウォールで踏み台攻撃の通信を遮断することができるが、踏み台にされた VM からの通信を完全に遮断してしまうとサービス可用性が大幅に低下してしまう。踏み台攻撃の誤検知の可能性を考えると、正常なアプリケーションの通信はなるべく止めないようにして踏み台攻撃だけに対処すべきである。

本研究では仮想マシンモニタ（VMM）内で動作するきめ細かいパケットフィルタ xFilter を提案する。xFilter は VM から隔離されているため、xFilter 自身が VM 内の侵入者から攻撃される可能性は低い。高いサービス可用性を保つため、xFilter はゲスト OS 内のパケット送信元情報を利用することで踏み台攻撃だけを防ぐ。メモリ解析を行ってゲスト OS 内の情報を取得し、それを用いて踏み台攻撃を行っているプロセスからのパケットのみを破棄する。メモリ解析を VMM 内で動作させるとそのバグがシステム全体に影響を与えやすくなるため、xFilter はゲスト OS に依存する部分をモジュール化し、その開発サポートを提供する。開発者はモジュールを専用の VM 上でテストし、変更なしに VMM 内に移植することができる。また踏み台攻撃を検出する侵入検知システムも VMM で動作させることで、地味大攻撃を検出するとすぐに送信元を特定し、攻撃を止めるために効果的なフィルタリングルールを自動生成する。

xFilter はパケット送信処理の途中で VM のメモリ解析を行うため、その性能がネットワーク性能に大きな影響を及ぼす。メモリ解析のオーバーヘッドを削減するために、本研究ではいくつかの最適化を行った。ゲスト OS 内の情報を利用する多くの既存システムとは違い、ドメイン 0 と呼ばれる特権 VM ではなく、VMM 内でメモリ解析を行う。VMM からはオーバーヘッドなしで直接 VM のメモリにアクセスできる。さらに、同一 TCP

コネクション内のパケットのフィルタリング結果をキャッシュすることによってメモリ解析の回数を減らす。

本研究では、Xen 3.4.2 上に xFilter を実装した。Linux 2.6.18 を対象とした xFilter モジュールとポートスキャンの検出を行う IDS を実装し、Apache Bench ベンチマークツールを用いて実験を行った。その結果、xFilter は一般的なプロセス数、ソケット数の場合にネットワーク性能の低下を 4%程度に抑えることができ、実用に耐えうる性能であることを確認した。

In the cloud computing era, stepping-stone attacks via hosted virtual machines (VMs) are critical for data centers. When VMs attack external hosts, data centers may be regarded as attackers. External firewalls are useful for stopping such attacks, but the service availability of stepping-stone VMs remarkably lowers if all packets from the VMs are dropped. For higher service availability, we propose a fine-grained packet filter running in the virtual machine monitor (VMM), which is called xFilter. xFilter drops only packets from processes performing stepping-stone attacks by using information in guest operating systems. It analyzes the memory of VMs to obtain such information. An intrusion detection system in the VMM accurately specifies attacking processes. Our experimental results show that xFilter achieves low overheads thanks to several optimizations.

謝辞

本研究を進めるにあたり、研究の方針や論文の書き方について助言をいただいた指導教員の千葉先生に心より感謝いたします。九州工業大学の光来先生にはシステムの設計・実装、プログラミング等研究全般に渡り指導していただきました。深く感謝いたします。東京工業大学の堀江倫大氏には研究に関する助言だけでなく、生活面においても様々な面で支えていただきました。心より感謝いたします。最後に、ともに研究活動をおこなった研究室の皆様に感謝いたします。

目 次

第 1 章　はじめに **10**

第 2 章　問題意識と関連研究 **12**
2.1 仮想マシンモニタ 12
2.1.1 Xen Virtual Machine Monitor 12
2.2 踏み台攻撃の脅威 13
2.2.1 VM 外部のファイアウォールによる対処 13
2.2.2 VM 内部のファイアウォールによる対処 14
2.3 関連研究 15
2.3.1 Amazon EC2 15
2.3.2 Livewire、IntroVirt 15
2.3.3 Antfarm 15
2.3.4 Geoger 16
2.3.5 XenAccess 16
2.3.6 ident プロトコル 16
2.3.7 ステートフル・パケット・インスペクション 17
2.3.8 Chorus、CAPERA 17

第 3 章　提案:xFilter **18**
3.1 VMM におけるきめ細かいフィルタリング 18
3.2 システム構成 18
3.2.1 xFilter コア 20
3.2.2 xFilter モジュール 20
3.2.3 IDS 21
3.3 モジュールの開発サポート 22
3.4 モジュール開発時の制約 24
3.4.1 フィルタリングルールの自動生成 24
3.5 RAW ソケットへの対応 25
3.5.1 IDS における対処 26
3.5.2 xFilter モジュールにおける対処 27

第 4 章　実装 28
4.1 システム構成 28
4.1.1 運用時の xFilter モジュール 28
4.1.2 開発時の xFilter モジュール 29
4.2 Linux ゲスト OS のメモリ解析 29
4.2.1 デバッグ情報の取得 30
4.3 Xen のメモリ管理 31
4.3.1 開発時のドメイン U のメモリへのアクセス 34
4.3.2 Xen の提供するドメインのメモリを操作する関数群 35
4.4 一貫性を保ったメモリ解析 36
4.5 フィルタリング結果のキャッシュ 37
4.6 一括検査 37
4.6.1 ハイパーコールへのポインタ渡し 38
4.7 IDS 38
4.7.1 攻撃検出フェーズ 40
4.7.2 攻撃元特定フェーズ 40

第 5 章　実験 41
5.1 ポートスキャンの検出 41
5.2 メモリ解析のオーバーヘッド 44
5.3 ウェブサーバの性能の低下 44
5.3.1 パケットフィルタリングによる性能低下 45
5.3.2 RAW ソケットへの対処による性能低下 48
5.3.3 IDS による性能低下 51
5.3.4 開発時の性能 55

第 6 章　まとめ 56

図 目 次

2.1 ファイアウォール 13

3.1 VMM 内で動作する xFilter 19
3.2 フィルタリングルールの例 19
3.3 開発時の xFilter の構成 21
3.4 シンボルのアドレスとメンバのオフセット 23
3.5 フィルタリングルールの統合例 24
3.6 RAW ソケットを用いたソケットの送信 26

4.1 Xen 上の xFilter のシステム構成 29
4.2 init_task のアドレスの取得 30
4.3 メモリ解析によるプロセス情報の取得 31
4.4 task_struct の型情報の取得 32
4.5 Xen のメモリの管理 33
4.6 仮想アドレスからマシンアドレスへの変換 34
4.7 ハイパーコール引数の宣言 38
4.8 ゲストから VMM へのコピー 39

5.1 ポートスキャンを防ぐルールの追加 41
5.2 ポートスキャンを防ぐルールの追加 42
5.3 プロセス数の変化とメモリ解析時間 42
5.4 ソケット数の変化とメモリ解析時間 43
5.5 ルール数の変化とメモリ解析時間 43
5.6 プロセス数の変化とスループット（フィルタリング ） . . 45
5.7 プロセス数の変化とレスポンス（フィルタリング） 46
5.8 ソケット数の変化とスループット（フィルタリング） . . . 46
5.9 ソケット数の変化とレスポンス（フィルタリング） 47
5.10 ルール数の変化とスループット（フィルタリング） 47
5.11 ルール数の変化とレスポンス（フィルタリング） 48
5.12 プロセス数の変化とスループット（RAW ソケット） . . . 49
5.13 プロセス数の変化とレスポンス（RAW ソケット） 49
5.14 ソケット数の変化とスループット（RAW ソケット） . . . 50

5.15 ソケット数の変化とレスポンス（RAW ソケット） 50
5.16 ルール数の変化とスループット（RAW ソケット） 51
5.17 ルール数の変化とレスポンス（RAW ソケット） 52
5.18 プロセス数の変化とスループット（IDS） 53
5.19 プロセス数の変化とレスポンス（IDS） 54
5.20 ソケット数の変化とスループット（IDS） 54
5.21 ソケット数の変化とレスポンス（IDS） 55

表 目 次

5.1 攻撃検出フェーズの性能低下 52
5.2 開発時の性能 55

第1章　はじめに

クラウドコンピューティングを提供しているデータセンタにとって踏み台攻撃 [20] は大きな脅威となっている。踏み台攻撃では、攻撃者自身のホストからではなく前もって侵入したホストから攻撃を行う。データセンタ内のホストが外部ホストへの攻撃の踏み台として利用された場合、侵入されたホストのユーザだけでなくデータセンタも攻撃の責任を負うことになる可能性がある。データセンタは被害者であると同時に、攻撃者にもなってしまう。特に、Amazon EC2 のような Infrastructure as a Service（IaaS）[12] と呼ばれるサービスモデルは踏み台攻撃に対してより脆弱である。IaaS はユーザに仮想マシン（VM）を提供し、ユーザは OS とその上のソフトウェアをインストールする。データセンタにとってすべての VM 内のすべてのソフトウェアに対して、すべてのセキュリティパッチが適用されていることを保証することは困難である。

このように、完全にユーザの VM への侵入を防ぐことは困難であるため、踏み台攻撃をできるだけ早く止めることが重要になる。外部ファイアウォールによるパケットフィルタリングは最も安全で確実な方法である。ある VM からの踏み台攻撃が検出されたら、外部ファイアウォールは VM からの全パケットを拒否すればよい。この単純なフィルタリングルールによって、侵入された VM からの踏み台攻撃を完全に遮断することができる。しかし、踏み台にされた VM 内のサービスの可用性は著しく低下してしまう。この VM 内の正常なアプリケーションも外部にパケットを送信できなくなるためである。ファイアウォールで攻撃に利用されている特定のポートへのパケットのみを拒否したとしても、正常なアプリケーションはそのポートを使ってパケットを送信できなくなる。踏み台攻撃の誤検知の可能性を考えると、正常なアプリケーションの通信はなるべく止めないようにして踏み台攻撃だけに対処すべきである。

本研究では、仮想マシンモニタ（VMM）内におけるきめ細かいパケットフィルタである xFilter を提案する。xFilter は VM から隔離されているため、xFilter 自身が VM 内の侵入者から攻撃される可能性は低い。高いサービス可用性を保つため、xFilter はゲスト OS 内のパケット送信元の情報を利用することで踏み台攻撃だけを防ぐ。本来、VMM は VM 内のゲスト OS の内部構造を理解できないが、xFilter は VM のメモリを解析して

必要な情報を取得する。攻撃パケットの送信元プロセスを指定することによって、特定のプロセスから送信されたパケットのみを拒否することができる。このように、パケットフィルタを VMM 内で動作させるとそのバグがシステム全体に影響を与えやすくなるため、xFilter はそのモジュールの開発サポートを提供している。開発者はモジュールを別の VM 上でテストし、変更なしに VMM 内に埋め込むことができる。また、侵入検知システム（IDS）も VMM 内で動作させることで、踏み台攻撃を検出するとすぐに送信元を特定し、攻撃を止めるためのルールを自動的に追加する。

我々は xFilter を Xen [3] に実装した。xFilter はパケット送信処理の途中で VM のメモリ解析を行うため、その性能がネットワーク性能に大きな影響を及ぼす。メモリ解析のオーバーヘッドを削減するために、本研究ではいくつかの最適化を行った。ゲスト OS 内の情報を利用する多くのシステム [15, 16] と違い、xFilter は Xen におけるドメイン 0 と呼ばれる特権 VM ではなく、VMM 内でメモリ解析を行う。VMM からはオーバーヘッドなしで直接 VM のメモリにアクセスすることができる。さらに、同一 TCP コネクション内のパケットのフィルタリング結果をキャッシュすることによってメモリ解析の回数を減らす。これらの最適化によって、xFilter は標準的なプロセス数、ソケット数の場合にネットワーク性能の低下を 4%以下に抑えることができた。

以下、2 章では踏み台攻撃に対する既存の対処法と関連研究について述べる。3 章では VMM で動作する新しいパケットフィルタ、xFilter を提案し、4 章では Xen における実装について説明する。5 章では xFilter のオーバーヘッドを測定した実験の結果を示し、6 章で本稿をまとめる。

第2章　問題意識と関連研究

2.1　仮想マシンモニタ

2.1.1　Xen Virtual Machine Monitor

Xen [3] は仮想マシンモニタであり、仮想マシンをドメインとして管理する。Xen は既存の OS の上でほかの OS を動作させるのではなく、複数の OS を動作させるための基盤となるプラットフォームを提供する。

Xen は仮想マシン環境構築に、準仮想化を呼ばれる手法を用いている。実在のハードウェアを完全にエミュレートする代わりに、仮想マシン環境を実現するために都合の良い仮想的なハードウェアを再定義する。この準仮想化という手法は、Xen 上で動作させるゲスト OS に手を加えることを要求する。

デバイスドライバの仮想化

仮想マシンモニタには、CPU だけでなく I/O デバイスも仮想化する必要がある。Xen の仕組みでは、I/O デバイスの仮想化に当たって実際のデバイスではなく、Xen 環境専用の仮想的なデバイスを定義している。ドメイン上のゲスト OS は、この仮想化デバイスに対して I/O アクセス要求を行う。この要求は実デバイスへの I/O アクセス要求に変換しなければならない。

ところが Xen は自分自身では実デバイスを管理しておらず、ドメイン 0 と呼ばれる特別なドメインに実デバイスの制御を任せている。ゲスト OS が仮想デバイスに I/O アクセスを要求すると、そのままドメイン 0 に転送し、ドメイン 0 が代わりにに実デバイスを操作する。

Xen ではネットワークインタフェースも仮想化されている。ドメイン間に仮想的なネットワークを立ち上げており、Linux カーネルが用意しているブリッジ機能を利用して、物理ネットワークインタフェースの先に仮想ネットワークインタフェースをつなぐ。ドメイン U が送信したパケットは、ドメイン 0 の物理ネットワークインタフェースを通して外部に送り出される（受信はその逆)。

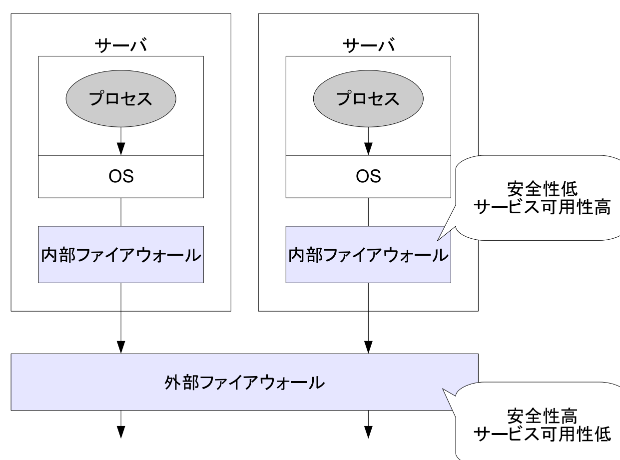

図 2.1: ファイアウォール

2.2　踏み台攻撃の脅威

踏み台攻撃は侵入された VM のユーザだけでなく、その VM を提供しているデータセンタの信用にも関わるため、できるだけ早く止める必要がある。データセンター内の VM が踏み台として利用されないようにすることが望ましいが、ソフトウェアには多くの脆弱性があるためそれは難しい。特にクラウドコンピューティングにおいては、物理ホストと違って必要な時だけ VM を動作させる。長い間動作させていなかった VM にはセキュリティパッチが適用されていないため、攻撃を受けやすい。また、セキュリティはパスワード管理などのようにユーザの教育にも依存する。幸い、踏み台として使われているコネクションの検知 [20, 23, 4, 6] やポートスキャンの検知 [19] などの様々な検知技術が提案されている。

本研究で扱う踏み台攻撃では、攻撃者に VM に侵入され、ゲスト OS の管理者権限を奪われることがあると仮定している。ただし、ゲスト OS のカーネルは改ざんされていないものとする。OS カーネルの改ざんは VMM が検出することができる [5, 13]。また、セキュリティハードウェアが VMM の整合性を保証できる [17, 22] ため、VMM も改ざんされていないものとする。

2.2.1　VM 外部のファイアウォールによる対処

データセンタにとって、外部ファイアウォールによるパケットフィルタリングが踏み台攻撃への最も簡単で確実な対処法である。外部ファイア

ウォールはデータセンタのネットワークの出入り口のように、踏み台にされる VM の外部に置かれている。VM への侵入者が外部ファイアウォールを攻撃することは困難なので、外部ファイアウォールによるパケットフィルタリングは安全である。しかし、外部ファイアウォールは IP アドレスやポート番号といった、パケットに含まれる情報しか検査できないため、踏み台 VM のサービス可用性を保つことは難しい。例えば、送信元の IP アドレスに基づくパケットフィルタリングが最も容易に適用できる。踏み台 VM からのパケットすべてを拒否するルールを追加すれば、踏み台攻撃を完全に止めることができる。その一方で、VM 内の正常なアプリケーションも外部にパケットを送信できなくなってしまう。その結果、踏み台 VM のサービス可用性はゼロになる。

パケットの他の情報を利用することでサービス可用性を向上させることもできる。送信先の IP アドレスに基づくパケットフィルタリングを行えば、特定のホストへのパケット送信だけを遮断することができる。このようなフィルタリングは特定ホストへのサービス妨害攻撃のように、攻撃対象ホストが少ないときは有効だが、対象ホストが多い場合にはすべてのホストを指定するのは難しい。侵入者が多くのホストに対して SMTP スキャンを行っているような場合は、送信先のポート番号に基づくパケットフィルタリングを行えば、すべてのホストの特定サービスへのパケット送信のみを遮断できる。侵入者はどのホストに対しても SMTP スキャンを行えなくなるが、正常なアプリケーションもメールを送れなくなる。送信元のポート番号を指定すれば、特定のコネクションだけを制限することができる。SMTP スキャンのような短時間のコネクションには効果がない。

2.2.2　VM 内部のファイアウォールによる対処

VM 内部のファイアウォールを利用すれば、高いサービス可用性を実現できる。このファイアウォールは OS カーネル内にあるため、パケットフィルタリングに送信元の情報も利用できる。例えば、Linux の iptables[14] や FreeBSD の ipfw は、フィルタリングルールにプロセス ID やユーザ ID を指定できる。侵入者の使用しているプロセスを特定することで、侵入者が SMTP スキャンを行ったときのみパケットを拒否することができ、それ以外の正常なアプリケーションはメールを送信することができる。しかし内部ファイアウォールは、踏み台攻撃に対して脆弱である。侵入者に管理者権限を奪われると、踏み台攻撃を防ぐためのフィルタリングルールを削除することで、内部ファイアウォールを簡単に無効化される。さらに、データセンタの管理者は VM にログインする権限も、内部ファイアウォールにルールを追加する権限もないことが多い。そのため、データセンタの管

理者が攻撃に気づいたとしても、VM のユーザに通知して踏み台攻撃を止めるには時間がかかってしまう。

2.3 関連研究

2.3.1 Amazon EC2

Amazon EC2 はセキュリティグループ [1] と呼ばれるファイアウォールを提供している。セキュリティグループは VMM 内で動作する外部ファイアウォールであり、ドメイン 0 に実装されていると思われる。このファイアウォールは対象 VM の外部に置かれているため、VM から攻撃されにくい。データセンタの出入り口に置かれているファイアウォールと違い、セキュリティグループは同一ホスト内の VM 間のパケットフィルタも可能である。しかし、セキュリティグループは外部からの攻撃に対する受信パケット専用のファイアウォールなので、踏み台攻撃を防ぐことはできない。xFilter は内部からの踏み台攻撃に対する送信専用のパケットフィルタであり、内部の送信元の情報を利用することができる。

2.3.2 Livewire、IntroVirt

Virtual Machine Introspection（VMI）は VMM からゲスト OS を検査する技術である。Livewire [5] と IntroVirt[10] は VM の外部で侵入検知を行うことができる。これらは VMI を用いて、VM 内のゲスト OS の内部状態を調べる。xFilter との 1 つの大きな違いは、これらのシステムにおいて性能はそれほど重要ではないことである。Livewire はオフライン検知ツールであり、IntroVirt はめったに実行されない実行パスでの侵入検知に使われる。xFilter はネットワークパケットの送信中に呼び出されるため、性能が直接ネットワーク性能に影響を与える。もう 1 つの違いは、xFilter は VMI を攻撃の検知ではなく攻撃に対する防御に利用していることである。

2.3.3 Antfarm

Antfarm [8] は、仮想マシンモニタ上からドメインに手を加えずにプロセスの状態を取得する技術である。さらに、取得したプロセスの状態を使って仮想マシンモニタ上で anticipatory I/O scheduling を実装している。

プロセスとアドレス空間は一対一に対応するので、アドレス空間を観察すればプロセスの様子が分かる。例えば i386 では、CR3 レジスタの値が

変化したら context switch したと分かり、CR3 レジスタの値が今までにない新しい値だったら新しいプロセスが生成されたと分かり、CR3 レジスタの値に対応するページマッピングが存在せずに TLB がフラッシュされたらその CR3 レジスタの値に対応するプロセスが終了したと分かる。

ドメイン上のオペレーティングシステムのソースコードに手を加えずに、オペレーティングシステムの情報を取得する点は本研究と同じである。しかし、取得できる情報はプロセスの状態の変化だけであり、何のプロセスかまでは分からない。オペレーティングシステムの内部構造まで解析していないので、取得できる情報は限られている。ドメイン上のオペレーティングシステムに依存せず、Linux 以外でも通用する技術であるという点で本研究より優れている。

2.3.4 Geoger

Geiger [9] は、仮想マシンモニタ上からドメインに手を加えずにバッファキャッシュの状態を取得する技術である。Antfarm [8] と同様に、ドメイン上のオペレーティングシステムのソースコードに手を加えずに、オペレーティングシステムの情報を取得する技術である。

2.3.5 XenAccess

XenAccess [15] は Xen のドメイン 0 からドメイン U 内のゲスト OS を調べるためのライブラリである。xFilter モジュールは開発時にはドメイン 0 内のプロセスとして動作するため、XenAccess を用いて実装することも可能である。しかし、運用時には xFilter モジュールは VMM 内で動作するため、XenAccess を利用することはできない。本研究では、ドメイン 0 からでも同一の API でゲスト OS を調べることができるライブラリを開発した。

2.3.6 ident プロトコル

ident プロトコル [7] はパケットを誰が送信したかを尋ねるために使われる。あるホストが別のホスト内で動作する ident サーバに送信元と送信先のポート番号のペアを送ると、サーバはそのネットワークコネクションを利用しているプロセスの所有者を返す。しかし踏み台攻撃が行われている時は、ident サーバは踏み台ホスト内で動作しているため攻撃を受ける可能性がある。さらに、このプロトコルはファイアウォールからではなくクライアントから利用することを想定している。xFilter は攻撃された

VM 内のサーバプロセスに依存せずに、OS カーネルを直接調べることでパケット送信元の情報を取得する。

2.3.7　ステートフル・パケット・インスペクション

フィルタリング結果のキャッシュは、ステートフル・パケット・インスペクション（SPI）に似ている。ファイアウォールの SPI は SYN フラグがセットされたパケットのみフィルタリングルールをチェックし、ステートテーブルに TCP コネクションの状態を保存する。コネクションが ESTABLISHED 状態になったら、コネクション内の全パケットを許可する。フィルタリング結果のキャッシュとの主な違いは、フィルタ結果のキャッシュはただのキャッシュであるが、ステートテーブルはそうではないことである。フィルタリング結果のキャッシュが一杯になり使われているコネクションのエントリが削除されても、xFilter がパケットを許可するかどうか決めるために再び VM のメモリを解析すればよい。一方、SYN flood 攻撃などでステートテーブルがオーバーフローすると、使われているコネクションのパケットを間違って破棄してしまう可能性がある。

2.3.8　Chorus、CAPERA

Chorus [18] や CAPERA [11] はカーネルモジュールの開発をサポートしている。これらの OS ではユーザプロセスとしてカーネルモジュールを実装し、修正なしにカーネルに埋め込むことができる。同様に、xFilter はヘルパー VM 内のユーザプロセスとしてモジュールを実装し、修正なしに VMM に埋め込むことができる。

第3章　提案:xFilter

安全性とサービス可用性を両立させる踏み台攻撃への対処法として、我々は新しいパケットフィルタ xFilter を提案する。

3.1　VMM におけるきめ細かいフィルタリング

xFilter は図 3.1 のように VMM 内で動作するパケットフィルタである。VMM は全ての VM から隔離されているため、VM への侵入者が VMM 内で動作する xFilter を攻撃することは難しい。さらに、全てのネットワークパケットは VMM を介して外部に送られるため、xFilter は VM からの全てのパケットを検査できる。外部ファイアウォールと違い、同一ホスト内の VM 間のパケットを遮断することで、同一ホスト内のある VM から別の VM への踏み台攻撃を防ぐこともできる。

フィルタリング対象の VM のサービス可用性を向上させるため、xFilter は VM のメモリを解析することでゲスト OS 内の情報を利用する。これにより、内部ファイアウォールで利用できるものと同様の情報を取得できる。例えば、xFilter に図 3.2 のように送信元のプロセスや所有者の ID を指定することができる。最初のルールはプロセス ID が 1234 のプロセスからの 25 番ポートへのパケット送信を拒否し、2 番目のルールはユーザ ID が 501 のユーザからのすべてのパケット送信を拒否する。このようにきめ細かいルールによって、踏み台攻撃を行っているプロセスやユーザからのパケットのみを遮断することができる。従来、VMM はゲスト OS の内部構造を理解しないため、ゲスト OS の内部の情報を利用することはできなかった。その意味では、VMM は踏み台 VM から完全に独立している外部ファイアウォールに似ている。しかし xFilter は VM のメモリにアクセスできるという点で、外部ファイアウォールとは異なっている。xFilter はゲスト OS に関する知識を利用することで、そのデータ構造を理解する。

3.2　システム構成

xFilter は xFilter コア、xFilter モジュール、IDS の 3 つのコンポーネントからなっている。

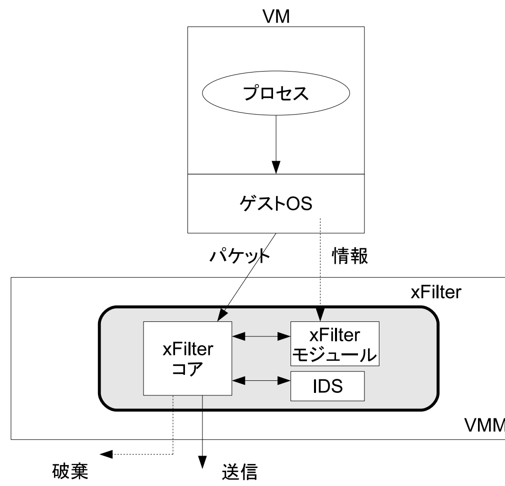

図 3.1: VMM 内で動作する xFilter

```
deny ip * port 25 pid 1234
deny ip * port * pid * uid 501
```

図 3.2: フィルタリングルールの例

3.2.1　xFilter コア

xFilter コアは、xFilter モジュール、IDS を統括する。コア自体は、送信パケットを横取りしてそのヘッダを解析する。ヘッダから取得する情報は送信先、送信元の IP アドレス及びポートの 4 つ組である。踏み台攻撃検出前はパケットを IDS に渡し、送信履歴が攻撃パターンを示していないかチェックされる。IDS によって攻撃が検出されるまでは xFilter モジュールは呼び出されず、IDS による攻撃の検出のみが行われる。踏み台攻撃検出後は、ヘッダ解析で取得した 4 つ組を引数に xFilter モジュールを呼び出し、送信の可否を問う。送信が許可されれば実際の送信処理を行い、拒否されれば送信処理を行わずに終了する。送信が許可されたパケットは送信処理の前に IDS に渡され、踏み台攻撃検出前と同様に、送信履歴が攻撃パターンを示していないかチェックされ、検出されていない新たな踏み台攻撃が行われていないかチェックする。

3.2.2　xFilter モジュール

xFilter ではパケットフィルタリングの実装をモジュールとして分離している。これは様々なゲスト OS に柔軟に対応できるようにするためである。送信元プロセスの情報を取得するためには、そのゲスト OS においてプロセスを管理している構造体を解析する必要があり、さらにプロセスが開いているソケットの情報を取得するために各プロセスがオープンしているファイルを解析する必要があるため、複雑なメモリ解析が必要になる。プロセスやファイルの管理はゲスト OS によって異なっており、対象ゲスト OS に合わせて個別にメモリ解析の実装を行う必要になる。また、同一ゲスト OS であっても利用するゲスト OS の情報が変わると新たなメモリ解析が必要になる。メモリ解析部をモジュールとして分離したことにより、ゲスト OS に合わせたモジュールに差し替えたり、新たにモジュールだけを開発することで様々なゲスト OS やゲスト OS 内の情報の利用に対処することが可能となっている。

xFilter コアがゲスト OS から送信されたパケットを受け取ると、xFilter モジュールが呼び出されフィルタリングルールに基づいてそのパケットを許可するかどうかを決定する。その際にゲスト OS のメモリ解析を行い、フィルタリングに必要な情報を取得する。特定のプロセスが送信したパケットかどうか調べるには、ゲスト OS のメモリからそのプロセスを探し出し、そのプロセスが行っている通信の一覧を取得して、受け取ったパケットの情報と比較する、といったことを行う。

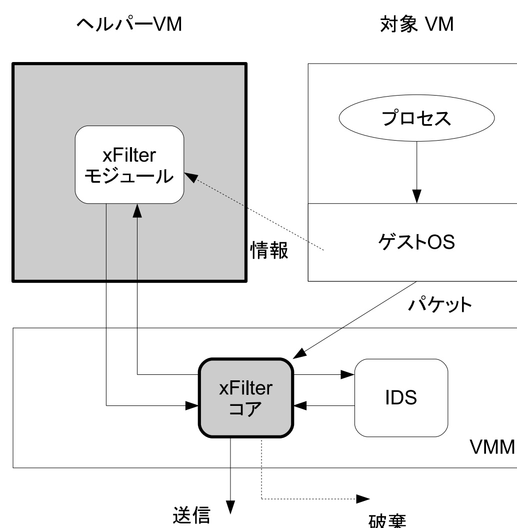

図 3.3: 開発時の xFilter の構成

3.2.3　IDS

IDS は、パケットの送信処理を行う前に xFilter コアによって呼び出される。IDS では送信するパケットの履歴をとっておく。新たなパケットを送信するたびに、送信履歴を参照して送信パターンをチェックし、踏み台攻撃と思しきパケットがないか調べる。

xFilter では VMM 内で IDS を動作させることにより、パケット送信から検出までのタイムラグを最小限に抑えることができる。従来のように外部の IDS を用いた場合、パケットの送信された後で IDS が攻撃を検知し、対象ホストに通知してルールを追加するという手順を踏むことになる。これではパケットの送信からルールの追加までに大きなタイムラグが発生してしまう。ルールを追加するために検出された攻撃パケットの送信元を特定しようとしたときには、すでにプロセスが終了してしまっていたり、ソケットをクローズしてしまっていたりして特定できない可能性が高い。xFilter 内の IDS がパケット送信処理を行う前に攻撃の検知を行うことで、送信元プロセスが特定できないという事態をできるだけ避けることができる。

3.3　モジュールの開発サポート

現在サポートされていないゲスト OS や OS のバージョンアップに対処するためには、新たな xFilter モジュールを開発してモジュールの差し替えを行う。しかし xFilter モジュールは VMM の一部であるため、その開発は容易ではない。開発者が新しいモジュールを実装したり、既存のモジュールを拡張する場合、VMM を修正する必要があるが、これには 2 つの問題がある。1 つ目は、モジュールに少しでも修正を加えたら、それを有効にするためにシステム全体を再起動する必要があることである。開発時は、モジュールへの変更が頻繁に行われる。2 つ目は、モジュールにバグがあった場合、VMM がクラッシュしてシステム全体の再起動が必要になることである。VM のメモリ解析を行うプログラムは 4 章で示すようにカーネルのデータ構造を VM の外部から間接的に扱わなければならないため複雑であり、開発の初期段階では多くのバグがある。システム全体の頻繁な再起動は開発効率を低下させる。

xFilter にはデバッグのために、図 3.3 のように、**ヘルパー VM** と呼ばれる別の VM 上でモジュールを動作させることができる。モジュールをヘルパー VM 上のプロセスとして動作させることによって、プロセスを実行しなおすだけで新しいモジュールを有効にできる。モジュールがクラッシュした時、開発者はプロセスを再実行するだけでよい。モジュールのクラッシュは、他のプロセスやヘルパー VM のゲスト OS、VMM、対象 VM に影響を与えない。ヘルパー VM 内のモジュールは VMM 内の xFilter 本体から呼び出され、VMM の機能を利用して対象 VM のメモリを解析する。

新しいモジュールの開発が完了したら、開発者はモジュールに修正を加えることなく VMM に埋め込むことができる。xFilter は VMM 内のモジュールとヘルパー VM 内のモジュールに同じ API を提供しているためである。モジュールが動作している場所による違いは xFilter の提供する API によって隠蔽される。例えば、VMM とヘルパー VM で他の VM のメモリにアクセスする方法は異なるが、xFilter は同じ関数を提供している。モジュールを VMM に埋め込みむことは、性能の点から必須である。モジュールをヘルパー VM 上で動作させると、パケットフィルタリングの性能が大幅に低下する。この場合、xFilter 本体は直接モジュールを呼び出すことができないため、ヘルパー VM とモジュールプロセスがスケジューリングされるのを待つ必要がある。また、ヘルパー VM から対象 VM のメモリへのアクセスにも時間がかかる。

```
#define init_task_addr  0xffffffff804674e0
#define tasklist_lock   0xffffffff804df000

#define PER_TASK_TASKS  200
#define PER_TASK_PID    308
#define PER_TASK_UID    632
#define PER_TASK_FILES  1464
#define PER_FILES_FDT   8
#define PER_FDT_MAXFDS  0
#define PER_FDT_FD      8
#define PER_FD_FDI      8
#define PER_FILE_FOP    32
#define PER_FILE_PRI    200
#define PER_SOCKET_SOCK 40
#define PER_INET_SPORT  584
#define PER_INET_DPORT  560
#define PER_INET_SADDR  564
#define PER_INET_DADDR  552
#define PER_SOCKET_TYPE 72
```

図 3.4: シンボルのアドレスとメンバのオフセット

```
deny ip * pid 1234 uid 501
deny ip * pid * uid 501
deny ip * pid * uid *
```

図 3.5: フィルタリングルールの統合例

3.4 モジュール開発時の制約

xFilter におけるモジュールの開発には、1 つ制約がある。それはカーネル内で定義されれいる構造体を直接利用できないことである。ドメイン 0 のカーネルとドメイン U のカーネルが同じとは限らない。つまりドメイン 0 内のカーネル内で定義されている構造体やそのメンバは、ドメイン U 内では定義されていなかったり、メンバが異なっているためにオフセットが違う可能性がある。よって、offsetof でメンバへのオフセットをとってくることが必ずできるわけではない。また VMM 内では定義されていないため VMM が構造体を理解できず、そもそも VMM ではその構造体を利用できない場合もある。メモリ解析時には静的に決定されるシンボルを起点に、構造体のメンバに格納されているポインタにアクセスし、順にポインタが示すアドレスにある構造体をたどっていく。オフセットが異なっていると、アクセスしようとしているメンバが異なっていたりポインタではないメンバにアクセスしてしまい、実行時に存在しないアドレスへのアクセスとなりエラーとなってしまう。また、静的に決定されるシンボルについても、そのアドレスが異なっているためにアクセスできない可能性がある。そこで、構造体のメンバや静的に決定されるシンボルへアクセスする場合は、図 3.4 のようにそのオフセットをすべてヘッダファイルに書き出す必要がある。これらの情報はデバッグ情報から取得することができる。

ヘッダファイルへの書き出しはバージョンが異なる OS への対応にも役立つ。バージョンが違っていても、プロセスやソケットの管理に大きな違いがあることは少なく、せいぜい構造体のメンバが異なっている程度である。本来ならばその程度であってもうまく動作せず、そのバージョンに適した新たな xFilter モジュールを開発する必要があった。しかし、異なっているのがオフセットやシンボルのアドレスであれば、それを書き出したヘッダファイルだけを差し替えれば対処できる。

3.4.1 フィルタリングルールの自動生成

xFilter 内の IDS は踏み台攻撃を検出すると、攻撃元を特定してフィルタリングルールを自動生成する。IDS が攻撃を検出すると、xFilter モジュールを呼び出してゲスト OS のメモリ解析を行い、攻撃パケットを送信した

プロセスを特定する。例えば、プロセス ID が 1234 で所有者のユーザ ID が 501 のプロセスがポートスキャンを行っていた場合、図 3.5 の 1 番目のようなルールが生成される。また、IDS が生成したルールを登録する際には、できるだけ効果的なルールになるように既存ルールとの統合を行う。例えば、攻撃元プロセスが次々に変わる場合には、図 3.5 の 2 番目のルールのようにユーザ ID の指定のみを行うルールに統合する。同一プロセスが攻撃を続ける場合にはプロセス ID の指定が有効であるが、プロセスを fork しながら攻撃されるとルールを追加した時にはそのルールは既に有効でない可能性が高い。ユーザ ID のみを指定することにより、それらのプロセスの所有者が同一であれば、以降の同様の攻撃に対しても有効なルールとなる。ただし、同一のユーザが正しいパケットを送信しても拒否されるため、サービスの可用性は低下する。

一方、管理者権限を奪われてプロセスの所有者も次々に変えて攻撃された場合には、図 3.5 の 3 番目のような、プロセス ID もユーザ ID も指定しないルールに統合する。この場合、外部ファイアウォールでのパケットフィルタリングと同様のサービス可用性した提供することができなくなる。しかし、内部ファイアウォールのようにルールを無効化かれて踏み台攻撃を防げなくなることはないので、管理者権限を奪われた場合でも安全性は保たれる。

3.5　RAW ソケットへの対応

SYN flood 攻撃などでは RAW ソケットを使って攻撃者が作成した TCP パケットを送信して踏み台攻撃が行われることがある。RAW ソケットは図 3.6 のように生の IP パケットをユーザ空間で組み立てて送信することができる。RAW ソケットはヘッダを自由に変更することができ、Protocol 番号を変更することにより、TCP 以外に UDP、ICMP 等も実装できる。パケットを作成して送信したプロセスがオープンしているソケットは TCP ソケットではなく、そのため、このようなパケットに関してはゲスト OS 内にその TCP の情報がなく、そのパケットの送信元プロセスを特定することができない。この RAW ソケットに対応できなければ IDS の攻撃元特定フェーズにおいて、受け取ったパケットの送信元を特定できないという問題が発生する。

RAW ソケットとしてソケットを作成するとき、第二引数に SOCK_RAW を用いて socket システムコールを呼び出す。つまり、ソケットの作成時に RAW ソケットであることを宣言しなければならない。socket システムコールの第二引数は socket 構造体内の type メンバに格納され、その値が SOCK_RAW であれば RAW ソケットであることが分かる。よって、各プ

```
int sock;
int on = 1;
struct iphdr *iphdrptr;
struct tcphdr *tcphdrptr;

sock = socket(AF_INET, SOCK_RAW, IPPROTO_RAW);

setsockopt(sock, IPPROTO_IP, IP_HDRINCL, &on, sizeof(on));

/* ヘッダに必要情報をセット */

sendto(sock,
       iphdrptr, sizeof(struct iphdr) + sizeof(struct tcphdr),
       0, (struct sockaddr *)&sin, sizeof(sin));
```

図 3.6: RAW ソケットを用いたソケットの送信

ロセスのオープンしているソケットを見る際、type の値を調べることでRAW ソケットを送信したプロセスを見つけることができる。プロセス A のオープンしているソケットの type の値が SOCK_RAW であればプロセス A は RAW ソケットを送信したということになる。しかし、xFilter コアが受け取ったパケットが RAW ソケットかどうかは分からない。TCP パケットを受け取ったとしても、それはプロセス A が TCP に偽装して送信した RAW ソケットかもしれないし、本当に TCP であるかもしれない。RAW ソケットの存在により、IDS の攻撃元特定フェーズや xFilter モジュールのメモリ解析にも影響が出てくる。

3.5.1　IDS における対処

RAW ソケットを考慮しないと、常にパケットの送信元を特定できない可能性がある。よって IDS の攻撃元特定フェーズでは常に RAW ソケットを意識しなければならない。RAW ソケットでは IP アドレスやポートなどの偽装を行う。各プロセスが RAW ソケットを送信したかを調べなければ、送信しようとしているパケットの送信元が見つからないかもしれない。逆に RAW ソケットを送信したプロセスを発見したとしても、送信しようとしているパケットが本当に RAW ソケットかわからない。そこで RAW ソケットを送信しているプロセスを発見したら、暫定的にそのプロセスを送信元であるとし、さらにメモリ解析を続けて送信元プロセスの特

定を続ける。全プロセスを調べた結果、そのパケットの送信元が特定できればそのプロセスが送信元であるとする。送信元が特定できなければ、暫定的に送信元であるとしていた RAW ソケットの送信元プロセスが送信元であると断定する。

3.5.2　xFilter モジュールにおける対処

IDS と同様に RAW ソケットを考慮しないと、送信しようとしているパケットの送信元を特定することができない。ユーザ A がフィルタリングルールに登録されていても、RAW ソケットを用いて攻撃を行っていれば攻撃を防ぐことができない。ユーザ A が所有者であるプロセスがオープンしている全ソケットを調べても、送信しようとしている RAW ソケットを見つけることができないからである。攻撃元特定フェーズと同様の手法で RAW ソケットに対処することが考えられるが、全プロセスのメモリを解析せねばならず性能に影響する。IDS の送信元特定フェーズで RAW ソケットによる攻撃を行っているプロセスが検出されるまでは、RAW ソケットを考慮せずにメモリ解析を行う。IDS によって RAW ソケットを用いた攻撃が検出されれば、RAW ソケットを考慮して全プロセスのメモリを解析する。これにより、RAW ソケットによる攻撃が行われていないときの性能低下を抑えつつ、RAW ソケットによる攻撃に対処できる。

第4章　実装

我々は、xFilter を Xen 3.4.2 [3] に実装した。Xen は**ドメイン 0** と呼ばれる特権 VM と**ドメイン U** と呼ばれるユーザ VM を提供しており、ドメイン 0 はドメイン U の I/O 処理を行う。ドメイン U 上で動作するゲスト OS として、x86_64 向け Linux 2.6.18 を対象にした。

4.1　システム構成

図 4.1 に Xen における xFilter のシステム構成を示す。ドメイン U 内で `send` システムコールが発行されると、ドメイン U 内の OS カーネルは、*netfront* と呼ばれるデバイスドライバにパケットを送る。netfront はドメイン 0 のカーネル内の *netback* と呼ばれるデバイスドライバにパケットを渡す。netback ドライバは物理 NIC デバイスドライバを呼び出す代わりに xFilter コアを呼び出す。xFilter モジュールがパケットの送信を許可し、IDS が攻撃を検出しなければ、パケットはネットワークドライバに送られ、NIC に渡される。

4.1.1　運用時の xFilter モジュール

運用時には、モジュールは性能のために VMM 内で動作する。VMM 内のモジュールを呼び出すために、ドメイン 0 内の xFilter コアは VMM にハイパーコールを発行する。ハイパーコールはパケットを送信したドメイン U の ID とパケット情報を引数にとり、xFilter モジュールにリンクされたスタブを呼び出す。スタブは指定したドメイン U を停止してからモジュールを呼び出す。VMM 内のモジュールはドメイン U のメモリに直接アクセスすることでメモリ解析を行う。フィルタリング結果はハイパーコールの返り値として xFilter コアに返される。xFilter コアは 1 つ以上のフィルタリングルールがあるときのみモジュール呼び出すため、踏み台攻撃が検出されるまではパケットは即座に IDS を通ってネットワークドライバに渡される。

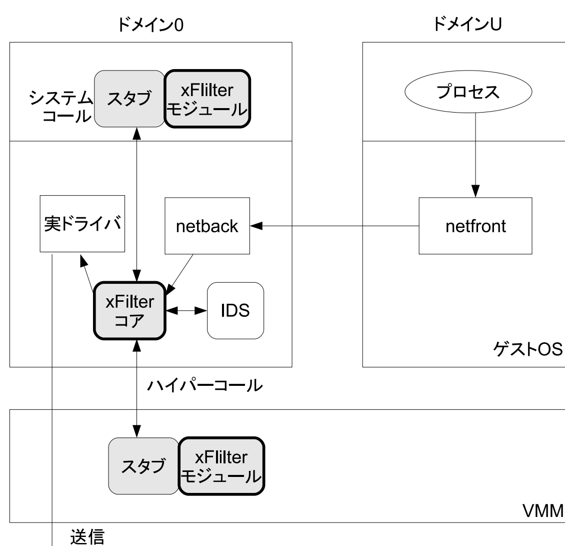

図 4.1: Xen 上の xFilter のシステム構成

4.1.2　開発時の xFilter モジュール

一方、開発時には、モジュールはヘルパー VM であるドメイン 0 内のプロセスとして動作する。モジュールにリンクされたスタブはシステムコールを発行し、xFilter コアが netback ドライバからパケットを受け取るのを待つ。パケットを受け取ったら、xFilter コアはモジュールのプロセスを起こし、システムコールの返り値でパケットを送信したドメイン U の ID とパケット情報を返す。スタブは VMM の機能を利用してドメイン U を停止し、モジュールを呼び出す。モジュールが動作するドメイン 0 はドメイン U から隔離されているため、モジュールは VMM の機能を利用して間接的にドメイン U のメモリにアクセスする。スタブは次のパケットを待つためのシステムコールを発行することで xFilter コアにフィルタリング結果を渡す。

4.2　Linux ゲスト OS のメモリ解析

xFilter はゲスト OS 内のデータ構造とシンボルのアドレスを取得するために、OS カーネルのデバッグ情報を利用する。本研究では Linux ゲスト OS を対象にパケットを送信したプロセスの情報を利用するモジュールの実装を行った。図 4.3 は関係するデータ構造をたどる手順を示している。Linux の場合、プロセスの ID や所有者の情報は task_struct 構造体に格納

```
$less System.map |grep init_task
ffffffff804439d0 r __ksymtab_init_task
ffffffff8044ee00 r __kcrctab_init_task
ffffffff80454858 r __kstrtab_init_task
ffffffff804674e0 D init_task
```

図 4.2: init_task のアドレスの取得

されており、init_task シンボルが init プロセスの task_struct 構造体に対応する。init_task のアドレスは静的に決定され、直接そのアドレスを指定する。静的に決定されるアドレスは図 4.2 のように System.map に管理されており、System.map から取得することができる。task_struct 構造体はプロセス ID 順にリンクされたリストとして管理されており、init_task のアドレスだけ取得すれば、リンクをたどって全てのプロセスの task_struct 構造体のアドレスにアクセスすることができる。ゲスト OS の情報を取得するためにまず、xFilter モジュールは init_task を起点として、ドメイン U 内のプロセスリストをたどる。プロセスリストをたどりながら、プロセスの ID または所有者がフィルタリングルールと一致するかどうかをチェックする。両方がどのルールとも一致しなかった場合、次のプロセスをチェックする。プロセス ID または所有者が少なくとも 1 つのルールに一致したプロセスに関しては、そのプロセスが使っているネットワークソケットを検査する。task_struct 構造体はオープンしているファイルを管理している file 構造体の配列へのポインタを持っている。file 構造体にはファイルとソケットの両方が格納されうるが、f_op メンバがグローバル変数 socket_file_ops のアドレスと一致すればソケットと判断できる。socket_file_ops のアドレスも System.map から取得できる。file 構造体からソケットを管理している sock 構造体までポインタをたどることができ、最終的に sock 構造体から送信元と送信先の IP アドレスとポート番号を取得する。

4.2.1　デバッグ情報の取得

xFilter はデバッグオプションつきでコンパイルしたカーネルイメージから情報を取得する。GNU C Compiler(GCC) を使い、デバッグ情報を付けてコンパイルするオプション -g を付けてコンパイルすると、デバッグ情報の付いた実行バイナリが生成される。デバッグ情報のフォーマットは Debug With Attributed Record Format(DWARF) である。DWARF には型情報以外にも様々な情報があるが、今回必要なのは型情報だけである。DWARF ファイルから型情報を取得するために、The GNU Project

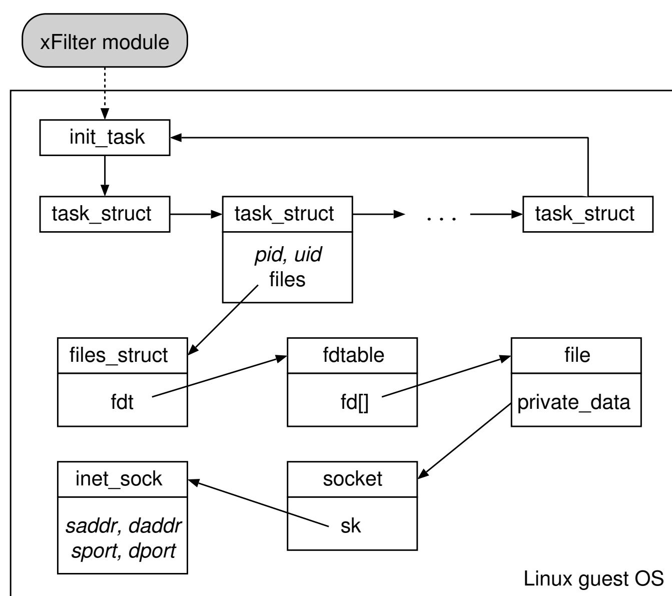

図 4.3: メモリ解析によるプロセス情報の取得

Debugger(gdb) を用いた。gdb の組み込みコマンド ptype を使えば図 4.4 のように型情報を取得できる。

4.3　Xen のメモリ管理

VMM からドメイン U のメモリにアクセスするために、モジュールはドメイン U 内の仮想アドレスを VMM が用いている**マシンアドレス**に変換する必要がある。Xen はマシンアドレスで管理されているメモリ領域の一部を、あるドメイン空間のアドレスにマップし、そのドメインに貸し出している。また、Xen はメモリをページサイズ毎に管理している。このページサイズ毎のまとまりをマシンフレーム (machine frame) と呼ぶ。メモリをページサイズ毎に区切った各マシンフレームに machine frame number(mfn) と呼ばれる 0 から始まる連続した番号が付いている。ドメインに割り当てられたマシンフレームの machine frame number は連続しているとは限らない。ドメインに割り当てられたマシンフレームを仮想物理フレーム (pseudo physical frame) と呼ぶ。ドメインに割り当てられたマシンフレームに、physical frame number(pfn) と呼ばれる番号が付いている。physical frame number は machin frame number の順序に従って順番に番号が付いている。ドメイン上の OS は、この仮想物理フレームを physical frame number の順序に連続した本物のメモリだと思って管理している。

```
$ gdb vmlinux
GNU gdb Fedora (6.8-37.el5)
Copyright (C) 2008 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu"...
(gdb) ptype struct task_struct
type = struct task_struct {
    volatile long int state;
    struct thread_info *thread_info;
    atomic_t usage;
    long unsigned int flags;
    long unsigned int ptrace;
    int lock_depth;
    int load_weight;
    int prio;
    int static_prio;
    int normal_prio;
    struct list_head run_list;
    struct prio_array *array;
    short unsigned int ioprio;
    unsigned int btrace_seq;
    long unsigned int sleep_avg;
               :
    int cpuset_mems_generation;
    int cpuset_mem_spread_rotor;
    struct robust_list_head *robust_list;
    struct compat_robust_list_head *compat_robust_list;
    struct list_head pi_state_list;
    struct futex_pi_state *pi_state_cache;
    atomic_t fs_excl;
    struct rcu_head rcu;
    struct pipe_inode_info *splice_pipe;
    struct task_delay_info *delays;
}
(gdb)
```

図 4.4: task_struct の型情報の取得

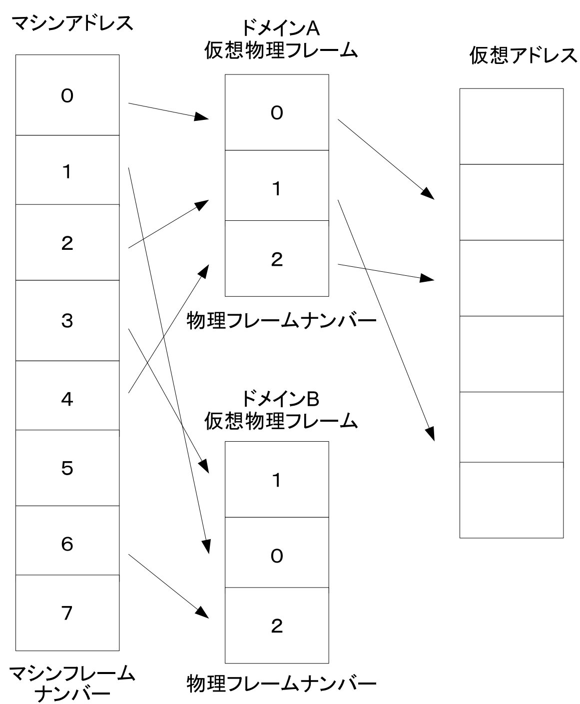

図 4.5: Xen のメモリの管理

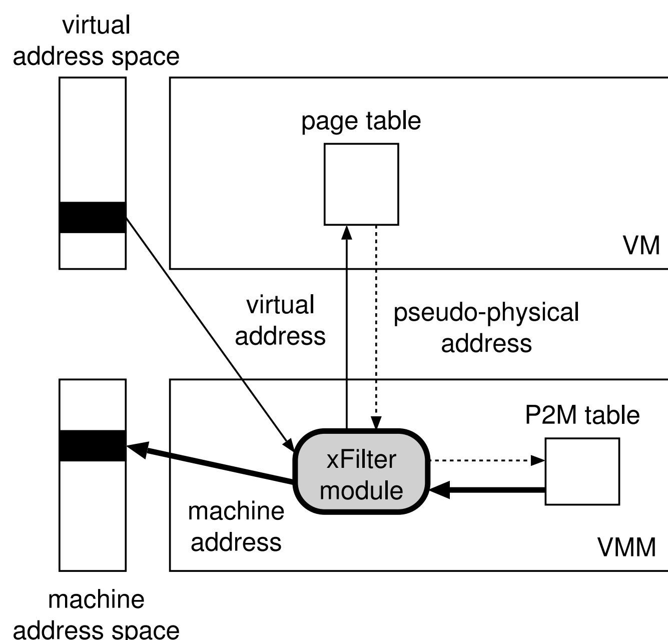

図 4.6: 仮想アドレスからマシンアドレスへの変換

図 4.5 は、Xen におけるメモリの管理を表している。1 つの箱は 1 ページ分を示している。マシンフレームナンバーが 4 のマシンフレームはドメイン A の物理マシンナンバーが 2 の仮想物理フレームに対応している。モジュールはドメイン U のページテーブルを調べることにより、仮想アドレスを仮想物理アドレスに変換する。さらに VMM の管理する P2M テーブルを調べることにより、仮想物理アドレスをマシンアドレスに変換する。

4.3.1　開発時のドメイン U のメモリへのアクセス

Xen の機能によって、ドメイン 0 のプロセスのアドレス空間にマシンフレームを割り当てることで、ドメイン 0 からドメイン U のメモリにアクセスすることができる。例えば、マシンフレームナンバーが 7 のマシンフレームをドメイン 0 のプロセス空間の 0x10000 から 0x11000 までのアドレスに貼り付けたとする。マシンフレーナンバーが 7 のマシンフレームはドメイン C の物理フレームナンバーが 3 の仮想物理フレームに対応する。

この場合、ドメイン 0 上のプロセスの 0x10000 から 0x11000 までのページへの書き込みは、マシンフレームを通してドメイン C の仮想物理フレームを変更したことになる。さらに、ドメイン C の物理フレームナンバーが 3 の仮想物理フレームがドメイン C 上のプロセスの 0x1000 から 0x2000

までのアドレスに割り当てられているならば、ドメイン C 上のプロセスのアドレス空間を変更することができる。

4.3.2 Xen の提供するドメインのメモリを操作する関数群

以下は、ドメイン 0 のプロセス内から呼びドメインのメモリを操作することができる関数であり、全て xenctrl.h で宣言されている。

- xc_transkate_foreign_address(handle, domID, cpu, addr)

  プロセスの仮想アドレスから、そのアドレスが存在する pfn を求める関数である。handle は Xen へのハンドラを表す。domID は仮想アドレスがあるプロセスのドメインの ID を表す。cpu は domID のドメインに割り当てられた CPU の中のひとつを表す。この関数を実行した時点での、そのプロセスのアドレスを求めることになる。

  前節でドメイン C の CPU1 でプロセスが走っているとすると、

  xc_translate_foreign_address(handle, domainC, 0, 0x1023) は、その 0x1023 が存在するマシンフレームである 7 を返す。

- xc_get_pfn_list(handle, domID, *pnf_buf, pnf_buf_size)

  ドメインに割り当てられた pfn のリストを返す関数である。呼び出し側で十分なサイズの配列を取っておき、その配列の先頭へのポインタ pfn_buf とその配列のサイズ pfn_buf_size を渡す。pfn_buf に入る pfn の順番は、ドメインから見て連続しているように見える順序である。

  前節の例で xc_get_pfn_list(handle, domainC, pfn_buf, 3) と呼ぶと、1、0、2 という順番で配列 pfn_buf に入る。

- xc_map_forign_range(handle, domID, size, prot, mfn)

  ドメイン 0 のプロセスのアドレス空間にマシンフレームを割り当てる関数である。返り値として、プロセスのアドレス空間に割り当てたアドレスが返ってくる。mfn は割り当てたいマシンフレームナンバーである。prot は割り当てたページのメモリ保護をどのように行うかを表す。prot は mmap システムコールの第三引数と同じであり、以下の論理和になっている。

  `PROT_EXEC` ページは実行可能である

  `PROT_READ` ページは読み込み可能である

  `PROT_WRITE` ページは書き込み可能である

  `PROT_NONE` ページはアクセスできない

- xc_map_foreign_batch(handle, domID, prot, pfn_buf, pfn_buf_size)

  あるドメインの pfn を連続してドメイン 0 のプロセスのアドレス空間に割り当てる関数である。pfn_buf は割り当てたい pfn を要素に持つ配列であり、pfn_buf_size はその要素数である。返り値は割り当てたプロセスのアドレス空間である。prot は xc_get_pfn_list と同じく割り当てたページのメモリ保護を表す。

  前節の例で、pfn_buf に 1、0、2 と入れて xc_map_foreign_batch を呼ぶと、返り値としてドメイン C 全体のメモリをマップしてその先頭のアドレスが返ってくる。

4.4　一貫性を保ったメモリ解析

外部からプロセスリストにアクセス使用とした場合、カーネルがプロセスリストを操作していないときに操作する必要がある。カーネルがプロセスの追加や削除によってプロセスリストのメンバに変更を加えているときは、正しいリストになっていない。正しくリンクされていないリストをたどろうとしても、うまくたどることができない。

本研究では、一貫性を保ってゲスト OS のメモリを解析するために、カーネルがプロセスリストを操作していないことが保証されているかを調べる。カーネルがプロセスリストを操作するときには、プロセスリスト内のスピンロックを取得する。スピンロックをしらべ、ロックが取得されていれば、カーネルがプロセスリストを操作しているとみなす。このスピンロックのアドレスは静的に決定され、その情報は前述の System.map に格納されている。

ハイパーコール呼び出し時に xFilter モジュールにリンクされたスタブは、まずメモリ解析を行うドメイン U をポーズする。次に、プロセスリストのスピンロックのメモリを解析し、ロックが取得されていないかをチェックする。スタブはロックが取得されていないことを確認するだけで、ロックの取得は行わない。メモリ解析の間はメモリ解析対象のドメインはポーズしているためロックを取得することができず、スタブはスピンロックを取得する必要はない。カーネルがロックを取得していればスタブはモジュールを呼び出すことを一旦中止し、しばらく動かした後で再度試みる。一般的に、スピンロックはすぐに解放されるため、ロックがカーネルに取得され続けたためにメモリ解析が行えなくなることはないと考える。プロセスリスト以外へのアクセスについては、カーネルはスピンロックを取得せずアトミックに処理を行っているため、スタブがロックをチェックする必要はない。

4.5　フィルタリング結果のキャッシュ

ゲスト OS のメモリ解析の回数を減らすために、xFilter コアはフィルタリング結果のキャッシュを行う。これにより、TCP コネクションを流れるパケットについては最初のパケットについてだけメモリ解析を行えばよくなる。xFilter がパケット送信を許可すると判断した時、フィルタリング結果をパケット情報とともにキャッシュに保存する。同じコネクションを流れるパケットはキャッシュにヒットするため、xFilter モジュールを呼び出すことなくパケット送信を許可する。xFilter コアが xFilter モジュールを呼び出したとしても、基本的にはキャッシュされたフィルタリング結果と同じになる。送信元プロセスの所有者が変わったり子プロセスが同じコネクションを使ったりした場合、xFilter モジュールによるフィルタリング結果とキャッシュは異なる場合がある。しかし同一コネクション内のパケットは元のプロセスや所有者と関係あると考えられるため、xFilter コアはキャッシュされたフィルタリング結果を適用する。

TCP コネクションに関しては、xFilter はパケットヘッダ内の TCP 制御ビットを基にキャッシュを管理する。xFilter コアが SYN フラグのセットされたパケットを受け取ると xFilter モジュールを呼び出し、送信を許可する場合はキャッシュにエントリを追加する。SYN フラグは新しいコネクションを確立するときにセットされる。キャッシュエントリはパケット情報として送信元と送信先の IP アドレスとポート番号を持つ。xFilter コアが FIN または RST フラグのセットされたパケットを受け取ったら、キャッシュから一致するエントリを削除する。FIN フラグは既存のコネクションを終了するときにセットされ、RST フラグはコネクションをリセットするときにセットされる。上記のフラグがセットされていないパケットに関しては、xFilter はキャッシュを調べ、一致するエントリがない場合だけ xFilter モジュールを呼び出す。

4.6　一括検査

ゲスト OS のメモリ解析の回数を減らすために、パケットの一括検査を行う。パケットが到着してもすぐに xFilter モジュールを呼ばず、ある程度キューにためておいてまとめて xFilter モジュールにパケットを渡す。xFilter コアはパケットを受け取ったとき xFilter モジュールを呼び出す代わりに、専用のキューにパケットを入れる。xFilter コアは一定時間毎にそのキューにたまったパケットをすべて xFilter モジュールに渡す。これにより、パケット到着毎に行っていたゲスト OS のメモリ解析を、キューにたまっているパケットに対して一度だけ行えばよくなる。

```
typedef struct hdr_info hdr_info_t;
DEFINE_XEN_GUEST_HANDLE(hdr_info_t);
```

図 4.7: ハイパーコール引数の宣言

4.6.1　ハイパーコールへのポインタ渡し

キューにたまったパケットのヘッダ情報を xFilter モジュールに一括で渡すには、ヘッダ情報を配列として VMM に渡す必要がある。ハイパーコール呼び出しの引数に配列を用いるためには、Xen の用意している機能を用いてドメイン 0 上から VMM 内、または VMM 内からドメイン 0 へその配列のメモリをコピーする必要がある。特に構造体の場合は、配列をハイパーコールの引数として使用すること事前にヘッダファイルに登録しなければならない。

例えば、

```
struct hdr_info
{
  short  sport;
  short  dport;
  int    saddr;
  int    daddr;
};
```

という構造体があるとする。この構造体の配列をハイパーコールの引数として使用するためには、図 4.7 のように宣言しなければならない。

さらにハイパーコールハンドラ側でも、図 4.8 のように引数を特殊な書き方で記述する必要がある。さらに Xen の用意する関数を用いて、VMM 内のメモリに配列の内容のコピーを行ってからでなければアクセスすることができない。配列への変更を反映させたい場合も同様に、Xen の用意する関数を用いて VMM 内のメモリからその配列をコピーしてから返す

4.7　IDS

本研究の目的は IDS の開発ではないため、ドメイン 0 に内部から外部へのポートスキャンを検出するだけの簡単な IDS を実装した。この IDS はパケットの情報だけから攻撃を検出するネットワーク IDS であるが、外部への攻撃を検出するて点が一般的な IDS とは異なる。ネットワーク型 IDS は、攻撃のパターンをシグネチャベースやアノマリベースで検知した

```
int
do_xFilter(XEN_GUEST_HANDLE(hdr_info_t) u_info,
           XEN_GUEST_HANDLE(int) u_result, int n)
{
  int ret = 0, flag = 0, result[n];
  struct hdr_info info[n];

  if( !IS_PRIV(current->domain) )
    return -EPERM;

/* 引数の配列を VMM 内のメモリへのコピー */
  if( copy_from_guest(info, u_info, n) )
    return -EPERM;

/* 様々な処理を行うコード */

/* 配列への変更をゲスト OS 内の配列にコピー */
  if( copy_to_guest(u_result, result, n) )
    return -EPERM;

  return ret;
}
```

図 4.8: ゲストから VMM へのコピー

り、連続したパケットを解析して攻撃の有無を判断する。本研究における IDS はアノマリベースの検知である。アノマリベースの検知とは通常のシステムではありえない行動を検出するもので、検知対象として以下のような異常行動がある。例えば、ping コマンドを使ってネットワーク機器の状態を確認することは通常の管理行動の一つだが、コマンドを送出するのはせいぜい 1 秒間に 1 回とか 1 分に 20 回といった頻度である。それが 1 秒間に 100 回送られたとすれば、それは異常行動と言える。目的のはっきりしない管理コマンドやメールの送付件数が異常に多くなったなどの状況が捉えられれば不正アクセスの発見につながる。また本研究で実装した IDS は攻撃検出フェーズと攻撃元特定フェーズの 2 つのフェーズで侵入検知を行う。

4.7.1　攻撃検出フェーズ

攻撃検出フェーズでは、TCP の SYN パケットのヘッダ情報と送信時刻を送信履歴として管理し、新たなパケットが送信されるたびに送信履歴に情報を追加してポートスキャンのチェックを行う。同一の IP アドレスに対して閾値を超える数のホストに送信を行っていれば、特定ホストへのポートスキャンと判断する。同一ポートに対して閾値を超える数のホストに送信を行っていれば、特定ポートへのポートスキャンと判断する。攻撃検出フェーズにおいてポートスキャンが検出されれば攻撃元特定フェーズに移行する。

4.7.2　攻撃元特定フェーズ

攻撃元特定フェーズでは送信履歴に送信元の情報も記録しながら再度、ポートスキャンの検出を行う。これは、ポートスキャンが検出されるまでは送信元の情報を取得するオーバーヘッドがかからないようにするためである。IDS は xFilter モジュールを使ってメモリ解析を行い、すべてのソケットを調べる必要がある。攻撃が継続されていれば攻撃元特定フェーズにおいてもポートスキャンが検出されるので、送信履歴を基にポートスキャンを防ぐフィルタリングルールを生成する。ポートスキャンとして検出された最後のパケットの送信元を攻撃元とすると、ポートスキャン中にたまたま正常なパケットが送信された場合に正常なプロセスを攻撃元と判断してしまう。このような誤検知を防ぐために、ポートスキャンと判定されたパケット群の送信元の中で、過半数を占めるプロセス ID またはユーザ ID を攻撃元とする。

第5章　実験

xFilter の有効性を調べるための実験を行った。まず、xFilter が踏み台攻撃を防げるかどうかの確認を行った。次に、xFilter のオーバヘッドを調べるために、VM のメモリ解析のオーバヘッドおよびウェブサーバの性能の低下を測定した。xFilter を動作させるサーバは、Intel Core i7 860 の CPU を 1 基、メモリ 8GB、ギガビットイーサネットを搭載した計算機を使用した。VMM としては Xen 3.4.2 、VM 内で動かす OS には Linux 2.6.18 を用いた。Xen のドメイン 0 にはメモリを 7GB、ドメイン U にはメモリを 1GB 割り当てた。ドメイン U 上で Apache ウェブサーバ 2.0 を動かした。一方クライアントには、Athlon 64 Processor 3500+ の CPU を 1 基、メモリ 2GB、ギガビットイーサネットを搭載した計算機を使用した。これらの計算機はギガビットスイッチで接続した。

5.1　ポートスキャンの検出

xFilter がドメイン U からのポートスキャンを遮断できることを示すために、nmap を用いて特定の外部ホストに対してポートスキャンを行わせた。その結果、nmap からのポートスキャンの検出し、、図 5.1 のルールを追加して攻撃を遮断した。また、ssh 等のほかのプロセスからの通信は行えることを確認した。次に、nmap プロセスを次々に fork しながらポートスキャンを行わせた。この実験でもポートスキャンの検出ができ、図 5.2 の上の 2 つのルールを追加して攻撃を遮断した。さらにルールの統合が行われ、3 番目のルールにマージされ、以降の攻撃も遮断した。別のユーザからは外部のサービスを利用できることを確認した。

```
deny ip * port * pid 16532
```

図 5.1: ポートスキャンを防ぐルールの追加

```
deny ip * port * pid 27904 uid 0
deny ip * port * pid 28281 uid 0
deny ip * port * pid * uid 0
```

図 5.2: ポートスキャンを防ぐルールの追加

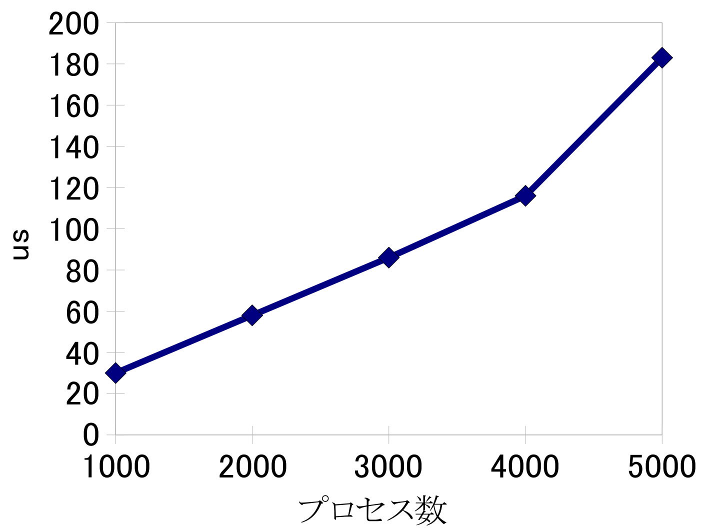

図 5.3: プロセス数の変化とメモリ解析時間

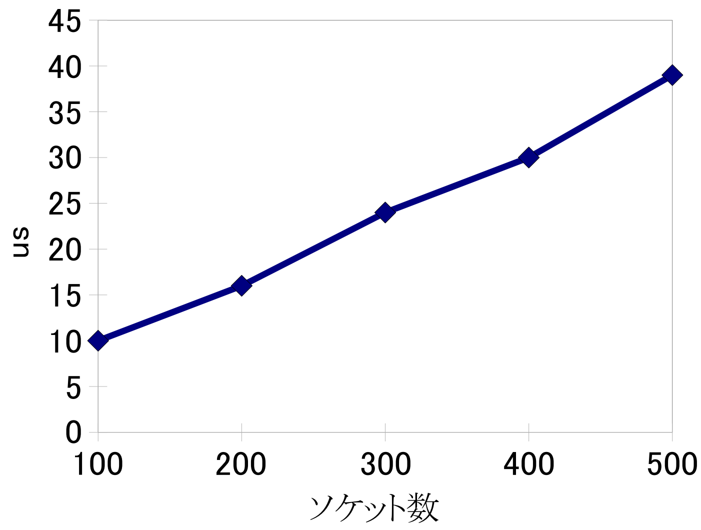

図 5.4: ソケット数の変化とメモリ解析時間

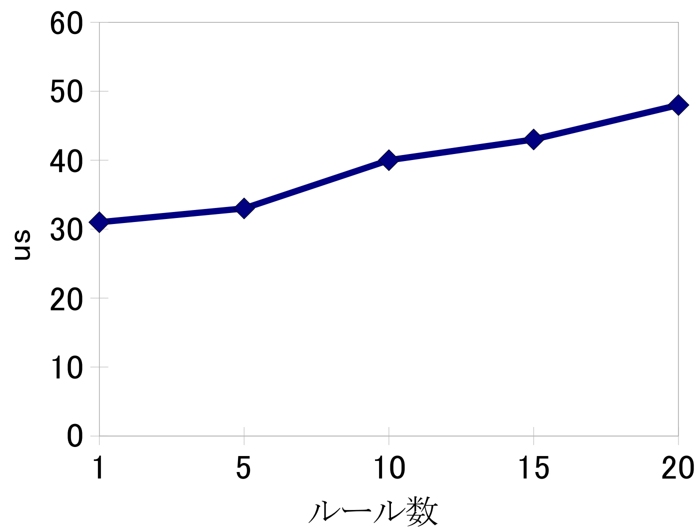

図 5.5: ルール数の変化とメモリ解析時間

5.2　メモリ解析のオーバーヘッド

VM のメモリ解析のオーバヘッドを調べるために、xFilter のモジュールを呼び出してメモリ解析を行うハイパーコールの実行時間を測定した。まず、xFilter にプロセス ID を指定したフィルタリングルールを登録し、ドメイン U 内のプロセス数を変えて実験を行った。プロセス数を調節するために、スリープし続けるプログラムを書いた。存在しないプロセスの ID を指定することで xFilter モジュールはプロセスリストの全エントリをたどり、全プロセスのプロセス ID を調べる。指定された ID のプロセスが存在しないため、xFilter モジュールはソケットを調べない。図 5.3 はメモリ解析にかかる時間を示している。実行時間はほぼプロセス数に比例し、プロセス毎に 30ns であった。5000 プロセスで 183μs になるが、一般的にはこれほど多くのプロセスが動作することはない。より現実的な 1000 プロセスでは 30μs であった。

次に、xFilter にユーザ ID のみを指定したフィルタリングルールを登録して、ドメイン U 内のプロセスがオープンしているソケット数を変えて同様の実験を行った。xFilter モジュールはプロセスリストをたどり各プロセスのユーザ ID を調べる。ユーザ ID がフィルタリングルールで指定したものであったら、xFilter モジュールはソケットまでデータ構造をたどっていく。図 5.4 はその結果を示している。実行時間はほぼソケット数に比例し、ソケット毎に 80ns であった。500 ソケットでは 39μs であった。

さらに xFilter のフィルタリングルール数を変えて同様の実験をおおなった。すべてのフィルタリングルールには存在しないプロセスの ID を指定した。xFilter モジュールはプロセスリストをたどりながらすべてのフィルタリングルールとプロセス ID を比較する。この実験では、ドメイン U 内で 1000 プロセスを動作させた。実験結果は図 5.5 のようになった。メモリ解析時間はルール数に比例し、ルール毎に 1μs であった。

5.3　ウェブサーバの性能の低下

実際のアプリケーションの性能の低下を調べるため、Apache ウェブサーバ [2] のスループットとレスポンス時間を測定した。ApacheBench ベンチマークツール [21] をクライアントマシンで動作させた。ApacheBench はサーバマシンの VM 内で動作する Apache に HTTP リクエストを送る。リクエストする HTML ファイルのサイズは 50KB とした。前節の実験と同様のフィルタリングルールで実験を行った。フィルタリング結果のキャッシュの効果を示すため、キャッシュを有効にした場合と無効にした場合について実験を行った。xFilter を用いなかった場合、スループットは毎秒 961 リクエスト、応答時間は 1.04ms であった。

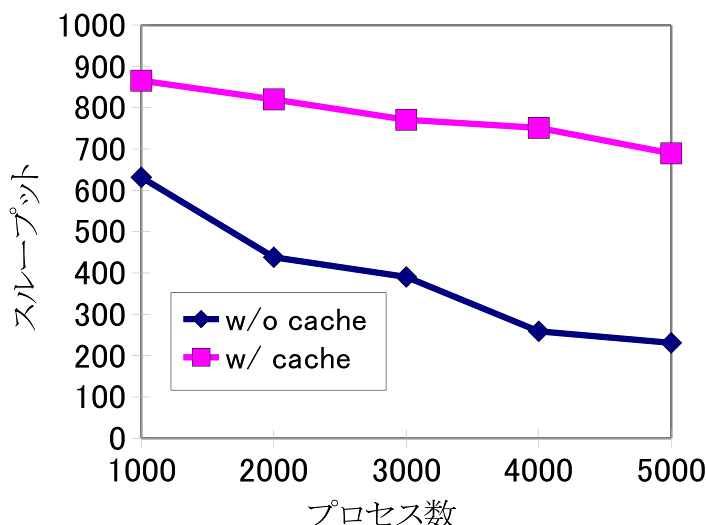

図 5.6: プロセス数の変化とスループット（フィルタリング ）

5.3.1 パケットフィルタリングによる性能低下

この実験では IDS を動作させず、パケットフィルタリングのみを行った場合のウェブサーバの性能低下について調べた。まず、プロセス数を変えてウェブサーバの性能を測定した。図 5.6、5.7 はその結果を示しており、性能はプロセス数に比例して低下している。キャッシュを有効にすると、5000 プロセスでもスループットの低下は 21%、応答時間の増大は 33%であった。より現実的な 1000 プロセスでは性能の低下は 8%であった。キャッシュを無効にすると、1000 プロセスでさえスループットは 28%の低下、応答時間は 32%の増大になった。これらの結果より、キャッシュはオーバヘッドの削減に効果的であることが分かる。

次に、ドメイン U 内のプロセスがオープンしているソケット数を変えてウェブサーバの性能を測定した。スループットと応答時間は図 5.8、5.9 のようになり、性能の低下はソケット数に比例している。キャッシュを有効にすると、500 ソケットでもスループットの低下は 5%、応答時間の増大は 6%であった。100 ソケットでは、性能の低下は 3%であった。キャッシュを無効にすると、スループットの低下は 44%、応答時間の増大は 77%になった。

さらに、フィルタリングルール数を変えてウェブサーバの性能を測定

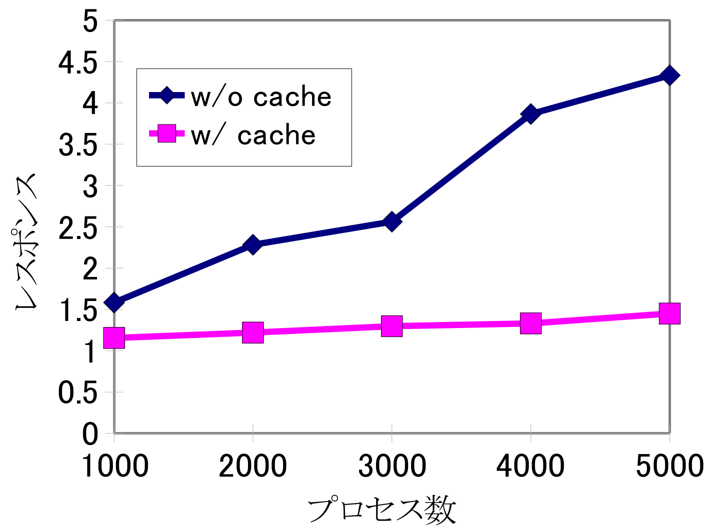

図 5.7: プロセス数の変化とレスポンス（フィルタリング）

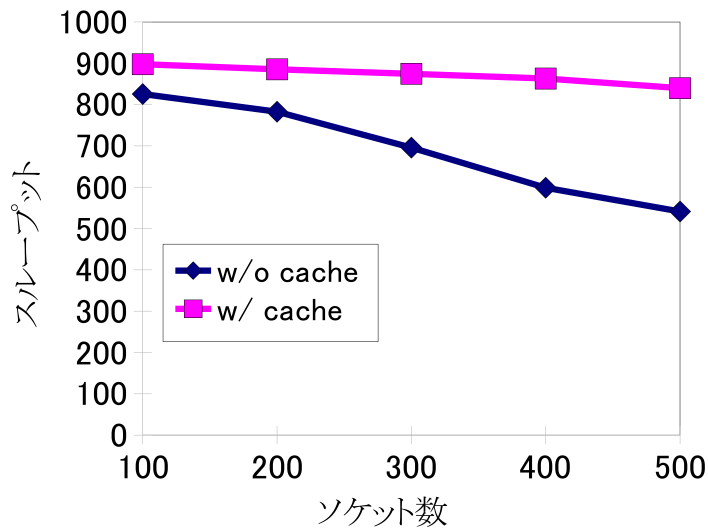

図 5.8: ソケット数の変化とスループット（フィルタリング）

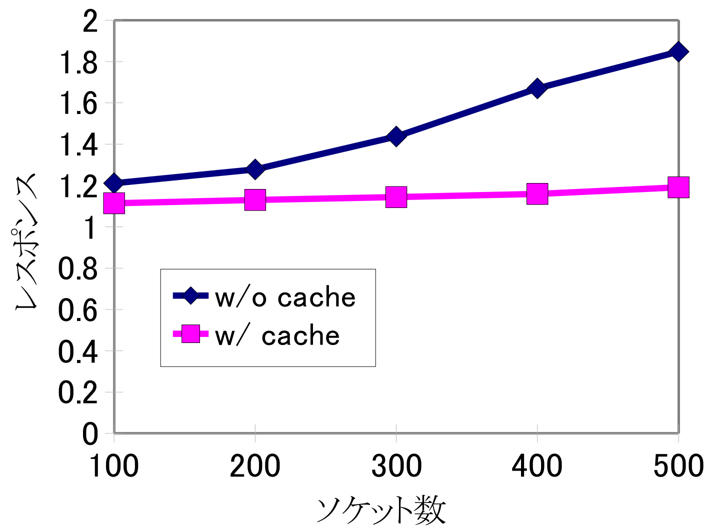

図 5.9: ソケット数の変化とレスポンス（フィルタリング）

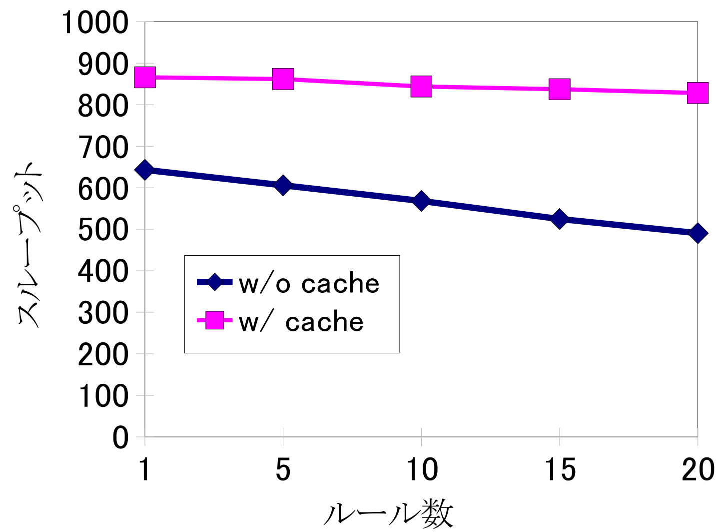

図 5.10: ルール数の変化とスループット（フィルタリング）

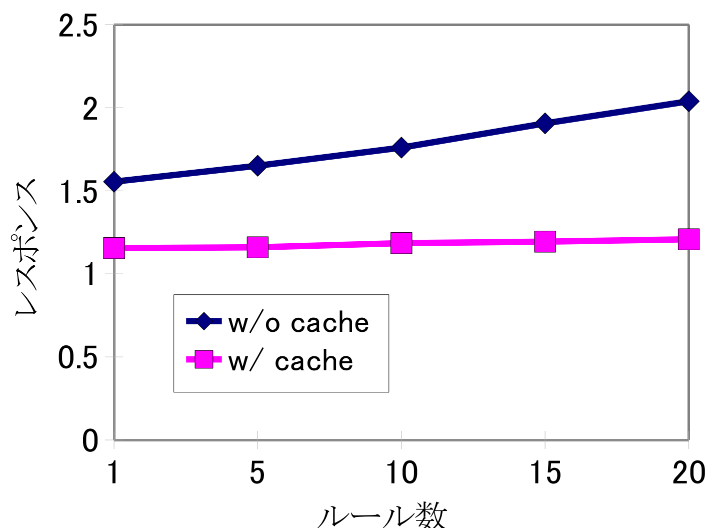

図 5.11: ルール数の変化とレスポンス（フィルタリング）

した。スループットと応答時間は図 5.10、5.11 のようになり、性能の低下はルール数に比例している。ルール数が 20 の時のスループットの低下は 10%、応答時間の増大は 11%であった。キャッシュを無効にすると、スループットの低下は 49%、応答時間の増大は 100%であった。このことから、IDS がルールを登録する際に行うルールの統合はオーバヘッドの削減にも役立つことが分かる。

5.3.2　RAW ソケットへの対処による性能低下

IDS によって RAW ソケットを用いた攻撃が検出された場合の、全プロセスの全ソケットについてメモリ解析を行う場合の性能の低下を調べた。この実験でも、キャッシュを有効にした場合と無効にした場合について実験を行った。

まず、プロセス数を変えてウェブサーバの性能を測定した。図 5.12、5.13 はその結果を示しており、性能はプロセス数に比例して低下している。キャッシュを有効にすると、5000 プロセスでのスループットの低下は 58%、応答時間の増大は 140%であった。より現実的な 1000 プロセスでは性能の低下は 26%であった。キャッシュを無効にすると、1000 プロセスでさえスループットは 82%の低下、応答時間の増大は 15 倍になった。

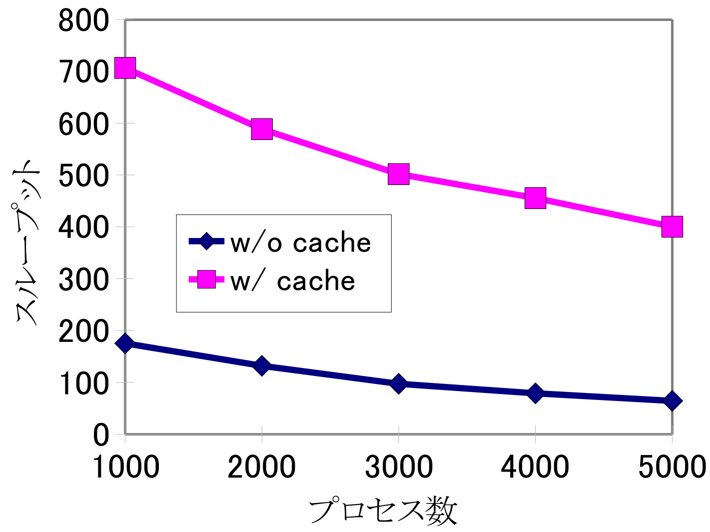

図 5.12: プロセス数の変化とスループット（RAW ソケット）

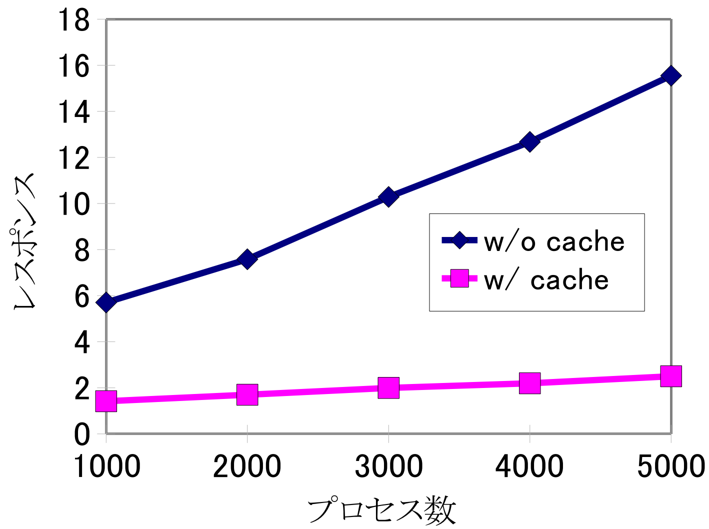

図 5.13: プロセス数の変化とレスポンス（RAW ソケット）

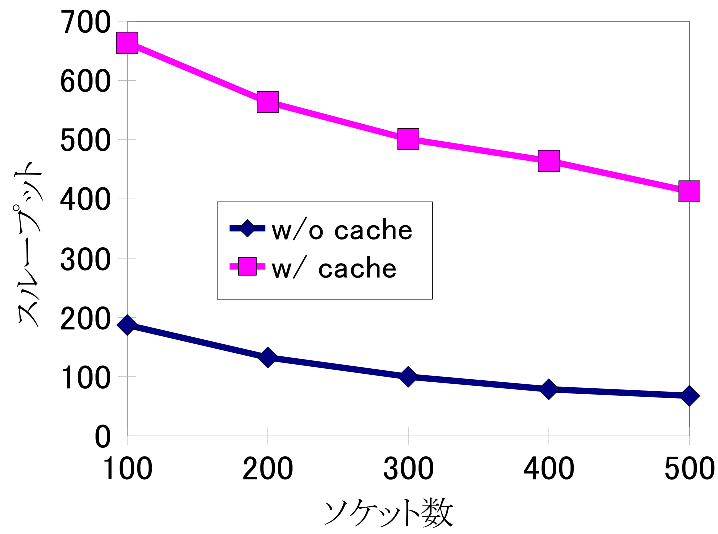

図 5.14: ソケット数の変化とスループット（RAW ソケット）

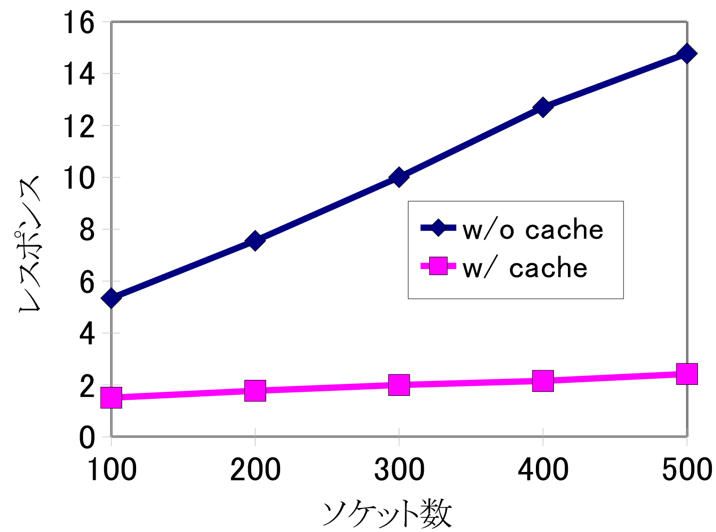

図 5.15: ソケット数の変化とレスポンス（RAW ソケット）

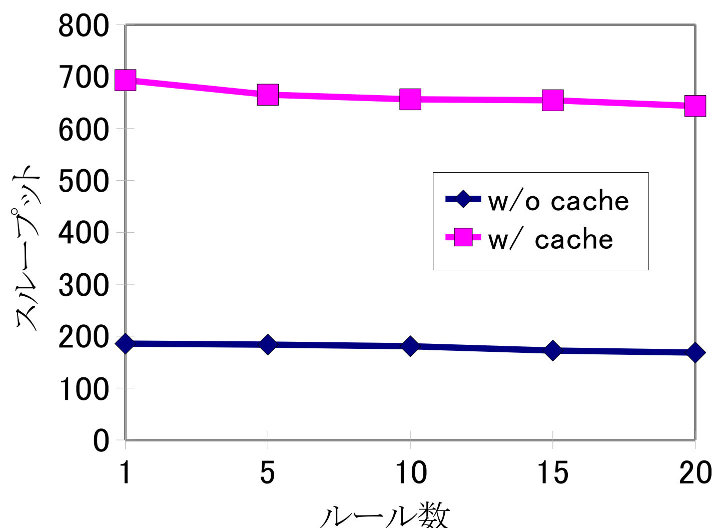

図 5.16: ルール数の変化とスループット（RAW ソケット）

次に、ソケット数を変えてウェブサーバの性能を測定した。スループットと応答時間は図 5.14、5.15 のようになり、性能の低下はソケット数に比例している。キャッシュを有効にすると、500 ソケットでスループットの低下は 57%、応答時間の増大は 133%であった。100 ソケットでは、性能の低下は 31%であった。キャッシュを無効にすると、スループットの低下は 80%、応答時間は 15 倍になった。

さらに、ルール数を変えて同様にウェブサーバの性能を測定した。スループットと応答時間は図 5.16、5.17 のようになり、性能の低下はルール数に比例している。ルール数が 20 のときのスループットの低下は 33%、応答時間の増大は 50%であった。キャッシュを無効にすると、スループットの低下は 82%、応答時間の 6 倍になった。これらの結果から、常に RAW ソケットを考慮するとオーバーヘッドが大きく、RAW ソケットが検出されていないときは RAW ソケットを考慮しないことは必要であることがわかる。

5.3.3　IDS による性能低下

ポートスキャンを検出する IDS を動作させた時のウェブサーバの性能を測定した。フィルタリングルールは登録せず、パケットフィルタリングが

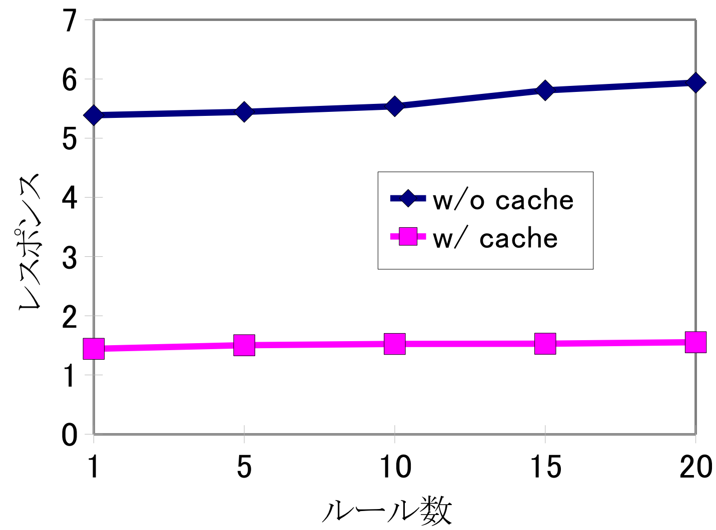

図 5.17: ルール数の変化とレスポンス（RAW ソケット）

表 5.1: 攻撃検出フェーズの性能低下

	スループット (reqs/s)	応答時間 (ms)
IDS なし	961	1.04
IDS あり	950	1.05

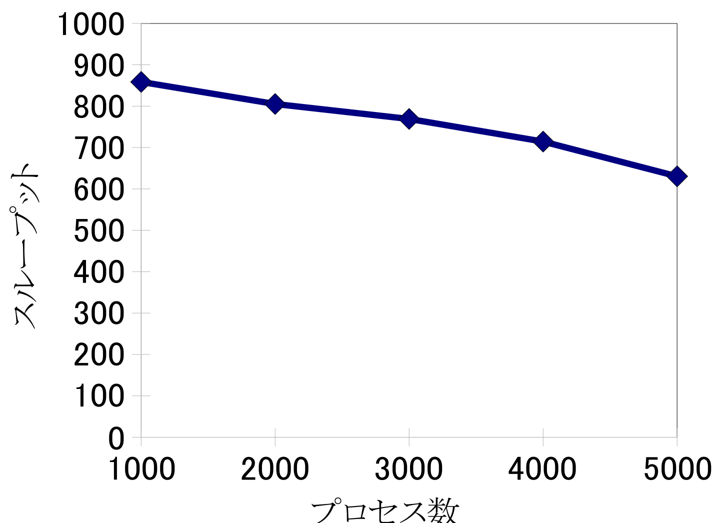

図 5.18: プロセス数の変化とスループット（IDS）

行われないようにして実験を行った。攻撃検出フェーズにおける性能の測定結果は表 5.1 のようになった。攻撃検出フェーズでは送信元を特定するために xFilter モジュールを呼び出さないため、ウェブサーバの性能はドメイン U 内のプロセス数やソケット数に依存しない。パケットヘッダの情報を記録した送信履歴をチェックするだけなので、性能の低下は約 1%であった。踏み台攻撃が検出されるまではこれ以外のオーバヘッドはかからない。

次に、ポートスキャンが検出された後の攻撃元特定フェーズにおけるウェブサーバの性能を測定した。攻撃元特定フェーズでは xFilter モジュールを呼び出して送信元の特定を行うため、前節の実験と同様にプロセス数、ソケット数を変えて実験を行った。送信元の特定にはフィルタリングルール数は影響しないため、ルール数を変えた実験は行わなかった。ドメイン U 内のプロセス数を変えて実験を行った結果は図 5.18、5.19 のようになり、性能はプロセス数に比例して低下している。また、オープンされているソケット数を変えた場合の実験結果は図 5.20、5.21 のようになり、性能の低下はソケット数に比例している。ここで用いた IDS は SYN パケットしかチェックしないため、5.2.1 節の実験でフィルタリング結果のキャッシュを有効にしたときと同程度のオーバヘッドであった。

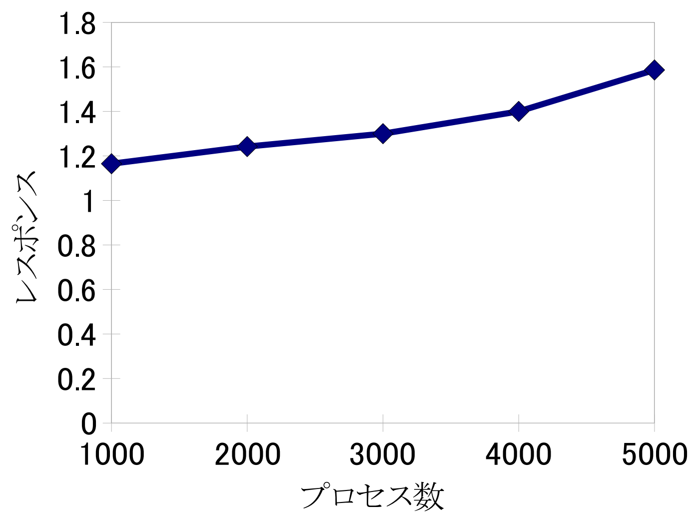

図 5.19: プロセス数の変化とレスポンス（IDS）

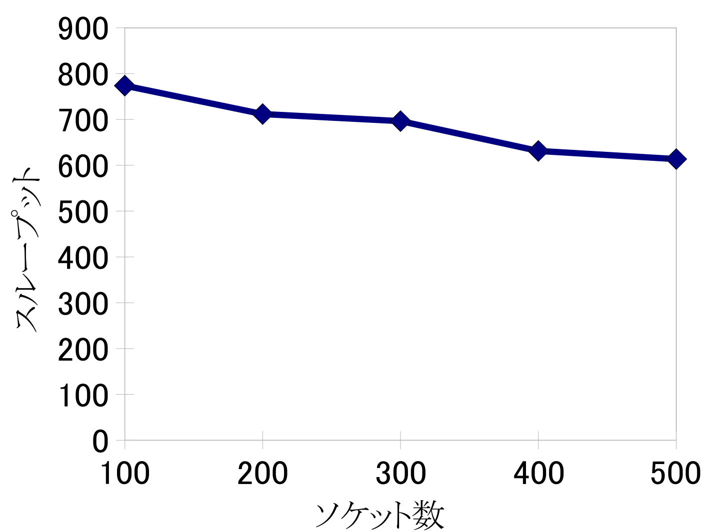

図 5.20: ソケット数の変化とスループット（IDS）

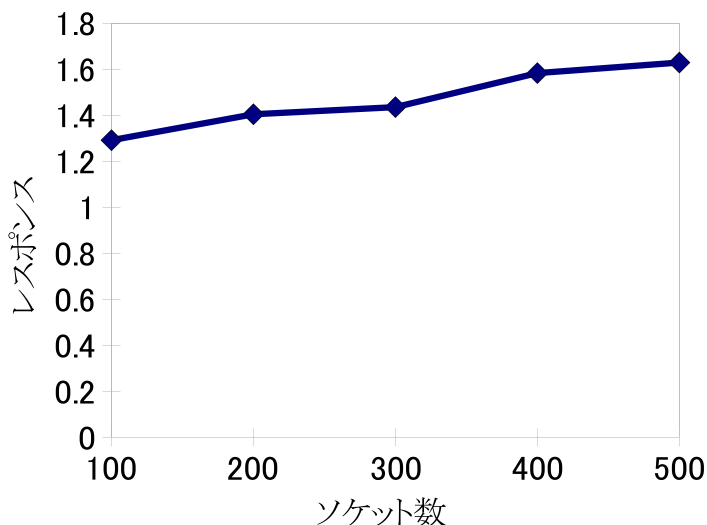

図 5.21: ソケット数の変化とレスポンス（IDS）

表 5.2: 開発時の性能

	スループット (reqs/s)	応答時間 (ms)
開発時	8.7	115
運用時	865	1.15

5.3.4　開発時の性能

開発時の xFilter モジュールは VMM 内ではなく、ドメイン 0 内のプロセスとして動作するためオーバーヘッドが大きい。モジュールをドメイン 0 内で動かした場合の性能を測定した。存在しないプロセスの ID を指定したフィルタリングルールを登録し、元々ドメイン U 上で動作している 64 プロセスに対して実験を行った。

表 5.2 の結果となった。この結果から、モジュール開発時の性能は運用時の VMM で動作する xFilter モジュールの 1%しかない。この実験から、xFilter モジュールを VMM 内で動作させることはネットワーク性能のために必須であることが分かる。

第6章　まとめ

本稿では、VMM で動作するきめ細かいパケットフィルタ xFilter を提案した。xFilter を用いてパケットの送信元のプロセスやユーザを指定してパケットフィルタリングを行うことで、可能な限り踏み台攻撃の通信のみを制限することができる。ゲスト OS 内の情報は、VMのメモリを参照しカーネルの型情報を用いて解析することによって取得する。IDS も VMM 内で動作させることで、踏み台攻撃を検出してから送信元を特定するまでのタイムラグをできるだけ減らしている。さらに、検査結果をキャッシュしておくことで xFilter のオーバーヘッドを削減した。

参考文献

[1] Amazon, Inc. Amazon Web Services: Overview of Security Processes. http://aws.amazon.com/security/, 2009.

[2] Apache Software Foundation. Apache HTTP Server Project. http://httpd.apache.org/.

[3] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proc. Symp. Operating Systems Principles*, pp. 164–177, 2003.

[4] D. Donoho, A. Flesia, U. Shankar, V. Paxson, J. Coit, and S. Staniford. Multiscale Stepping-stone Detection: Detecting Pairs of Jittered Interactive Streams by Exploiting Maximum Tolerable Delay. In *Proc. Int. Symp. Recent Advances in Intrusion Detection*, pp. 17–35, 2002.

[5] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proc. Network and Distributed Systems Security Symp.*, pp. 191–206, 2003.

[6] T. He and L. Tong. Detecting Encrypted Stepping-stone Connections. *IEEE Trans. Signal Processing*, Vol. 55, pp. 1612–1623, 2007.

[7] M. Johns. Identification Protocol. RFC 1413, 1993.

[8] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Antfarm: tracking processes in a virtual machine environment. In *USENIX-ATC'06: Proceedings of the Annual Technical Conference on USENIX'06 Annual Technical Conference*, pp. 1–1, Berkeley, CA, USA, 2006. USENIX Association.

[9] Stephen T. Jones, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Geiger: monitoring the buffer cache in a virtual machine environment. In *ASPLOS-XII: Proceedings of the 12th*

international conference on Architectural support for programming languages and operating systems, pp. 14–24, New York, NY, USA, 2006. ACM.

[10] A. Joshi, S. King, G. Dunlap, and P. Chen. Detecting Past and Present Intrusions through Vulnerability-specific Predicates. In *Proc. Symp. Operating Systems Principles*, pp. 91–104, 2005.

[11] Kenichi Kourai, Shigeru Chiba, and Takashi Masuda. Operating System Support for Easy Development of Distributed File Systems. In *Proc. Int. Conf. Parallel and Distributed Computing and Systems*, pp. 551–554, 1998.

[12] P. Mell and T. Grance. The NIST Definition of Cloud Computing, Version 15. http://csrc.nist.gov/groups/SNS/cloud-computing/, 2009.

[13] Jr. N. Petroni and M. Hicks. Automated Detection of Persistent Kernel Control-flow Attacks. In *Proc. Conf. Computer and Communications Security*, 2007.

[14] Netfilter Core Team. The netfilter.org Project. http://www.netfilter.org/.

[15] B. Payne, M. Carbone, and W. Lee. Secure and Flexible Monitoring of Virtual Machines. In *Proc. Annual Conf. Computer Security Applications*, pp. 385–397, 2007.

[16] B. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proc. Symp. Security and Privacy*, pp. 233–247, 2008.

[17] N. Petroni, T. Fraser, J. Molina, and W. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proc. USENIX Security Symp.*, 2004.

[18] M. Rozier, V. Abrossimov, F. Armand, I Boule, M. Gien, M. Guillemont, F. Herrman, C. Kaiser, S. Langlois, P. Léonard, and W. Neuhauser. Overview of the Chorus Distributed Operating System. In *Proc. Symp. Microkernels and Other Kernel Architectures*, pp. 39–69, 1992.

[19] S. Staniford, J. Hoagland, and J. McAlerney. Practical Automated Detection of Stealthy Portscans. In *Proc. Conf. Computer and Communications Security*, 2000.

[20] S. Staniford-Chen and L. Heberlein. Holding Intruders Accountable on the Internet. In *Proc. Symp. Security and Privacy*, pp. 39–49, 1995.

[21] The Apache Software Foundation. Apache HTTP Server Benchmarking Tool. http://httpd.apache.org/.

[22] Trusted Computing Group. http://www.trustedcomputinggroup.org/.

[23] Y. Zhang and V. Paxson. Detecting Stepping Stones. In *Proc. USENIX Security Symp.*, pp. 171–184, 2000.