学位論文　博士（工学）

ネットワーク侵入検知・防御システムの実装法に関する研究

2011年度

慶應義塾大学大学院理工学研究科

花岡 美幸

論文要旨

ネットワークを介したリモート攻撃を検出・防止する手段の一つとして，ネットワーク侵入検知・防御システム（NIDS/NIPS: Network Intrusion Detection/Prevention Systems）が広く利用されている．リモート攻撃とは，脆弱性のあるサーバにインターネットを通じて攻撃メッセージを送り，被害を発生させる攻撃である．NIDS/NIPS はサーバに送信されてくるメッセージを検査することによって，攻撃を検知し管理者に警告を発することで，サーバを攻撃から防御する．

リモート攻撃が巧妙になりインターネット上のトラフィックが増大するなど，NIDS を取り巻く環境の変化により，次の 3 つの課題が出てきている．第一に，攻撃検知の精度向上が求められている．攻撃が高度化・巧妙化したことにより，従来の単純なシグネチャ・マッチングでは検知できない攻撃が増えているためである．第二に，NIDS の性能向上が求められている．インターネット・トラフィックの増大や検知手法の高度化に伴い，NIDS の負荷が増加しているためである．第三に，NIDS の障害発生時にも継続して攻撃検知が可能となる，耐障害性が求められている．汎用 PC を元にした構成が多い NIDS が，ハードウェア故障などにより NIDS に障害が起こった場合にも，継続して攻撃検知が行える必要がある．

本論文では，NIDS/NIPS の実装技術として検知精度・性能向上・耐障害性向上を解決する手法を提案する．まず，検知精度向上のためにレイヤ 7 コンテキストを考慮した攻撃検知を行えるようにする．近年，単純なバイトパターンのマッチングでなく，メッセージの順番やフォーマットなどのレイヤ 7 コンテキストを考慮することにより，検知精度を高める NIDS が提案されている．レイヤ 7 コンテキストを考慮した攻撃検知を行うためには，ネットワーク上を流れる個別のパケットをメッセージに再構成する，TCP ストリーム再構成機構が必要となる．レイヤ 7 NIDS/NIPS のための TCP ストリーム再構成機構は次の 4 つの要件を満たさなければならない．すなわち，1) 攻撃メッセージが攻撃対象アプリケーションに届くことがない完全な防御，2) 性能低下が少ないこと，3) NIDS/NIPS の設置に伴ってサーバやクライアントのアプリケーションに変更や再設定が必要ない，アプリケーション透過性，4) 監視する通信の振る舞いを乱さない，すなわち TCP フローや輻輳制御に与える影響が少ないトランスポート透過性，の 4 つである．本論文では，これらを全て満たす TCP ストリーム再構成機構として Store-Through 方式を提案する．Store-Through 方式では，個別のパケットからメッセージに再構築する際，順番が入れ替わったパケットの転送を止めずにコピーをとって転送することでトランス

ポート透過性を保持する．また，攻撃だと判断された時点で後から到着したパケットを破棄することで，攻撃の成功を妨げ完全な防御を達成する．また，IP レベルで実装することで，性能とアプリケーション透過性を達成する．プロトタイプをLinux 2.4.30 上に実装し，実験によって Store-Through 方式によるオーバヘッドは，パケットをそのまま IP 層で転送するだけの場合に比べて，3.8%以下であることを示した．また，実際のネットワークを用いた実験により，トランスポート透過性を保持できていることを示した．

次に，組織ネットワーク内に複数の場所に置かれた NIDS 同士を協調させることで，性能向上と耐障害性向上を行う手法を提案する．大学や企業など多くの組織では，組織内ネットワークとインターネットの境界のみでなく，内側のネットワークの様々な階層に複数の NIDS を設置している場合が多い．本論文で提案する NIDS 協調システム Brownie では，これらの組織ネットワーク内に置かれた NIDS 間でルール設定を交換し合い，定期的に負荷情報をやりとりすることで，NIDS 同士のルール設定を連携させる．そして，性能を向上するため，過負荷になった NIDS の負荷を減少させ，NIDS 間の冗長なルール設定を削除するようにルールを再設定する．また，耐障害性を向上するため，NIDS の障害時には障害が起こった NIDS で有効にしていたルールを，別の NIDS で有効にすることで攻撃検知を代替する．Brownie のプロトタイプを実装し，実験によって，ルールを再設定することで，web サーバベンチマークのスループットが 10%以上向上することを示した．また，NIDS を意図的に停止させ障害を起こした実験では，100 秒程度で別の NIDS でルールが有効になり攻撃が検知されるようになった．この時間は手動で障害回復を行う場合に比べて十分短いと言える．

Abstract

Network intrusion detection/prevention systems (NIDS/NIPS) are widely used for detecting or preventing network-based remote attacks. A remote attacker sends a malicious message to a vulnerable server and causes various damages on it. NIDS/NIPS monitor network traffic for malicious activities, and raise alerts or drop the packets when they detect attacks.

The environmental changes surrounding NIDS expose three issues of current NIDS implementation. First, NIDS needs more accurate detection mechanisms. Since attacks are becoming more complex and sophisticated, some attacks cannot be detected by simple byte-pattern matching. Second, performance improvement is necessary. Because of today's increased traffic volume and sophisticated attacks, NIDS needs enough performance to cope with hi-speed network and complex in-depth analysis to detect attacks. Third, NIDS needs a fault-tolerant mechanism. Although most of current NIDS are developed as software and run on commodity computers, NIDS should continue to detect attacks even under failures.

This dissertation proposes mechanisms to solve the above three issues: detection accuracy, performance, and fault-tolerance. First for detection accuracy, this dissertation proposes a mechanism for layer-7-aware detection with little performance overhead. Although simple string matching is traditionally adopted to detect malicious activities, exploiting layer 7 contexts has been recognized as an effective approach for improving the accuracy of detecting malicious messages in NIDS. Layer-7-aware NIDS requires a TCP stream reassembler which reassembles packets into a message without losing 1) complete prevention, which means the NIPS must be able to prevent target applications from receiving malicious messages, 2) performance efficiency, 3) application transparency, which means the NIDS installation does not require any modification or reconfiguration of the client or server applications, or 4) transport transparency, which means that the NIDS does not impair end-to-end TCP/IP semantics. This dissertation proposes the store-through mechanism which satisfies all the requirements. Store-through preserves transport transparency by forwarding each out-of-order packet immediately after copying the packet. Although the forwarded packet might turn out to be a part of an attack message, the store-through mechanism can successfully defend against the attack by blocking one of the subsequent packets that contain another part of

the attack message and thus provides complete prevention. In addition, IP-level implementation provides performance efficiency and application transparency. Testings of a prototype in Linux kernel 2.4.30 demonstrate that the overhead of store-through is less than 3.8% compared to simply IP forwarding the recieved packets. The experiments over the real Internet also suggest that store-through preserves transport transparency.

For performance and fault-tolerance, this dissertation proposes Brownie, a system which coordinates configurations of already-existing, independently-managed NIDSs in an organization. Our key observation is that most organizations, such as universities or companies, have several NIDSs managed by different administrators inside their internal networks, not only at the network entry point. With our proposed system Brownie, NIDSs exchange their own load status and rule configuration. Then Brownie achieves performance improvement by offloading overloaded NIDS and eliminating redundant rules. For fault-tolerance when a NIDS fails, Brownie enables rules once checked by the failed NIDS so that the other NIDS(s) takes over the failed NIDS. The experimental results with a web server benchmark suggest that Brownie increases the benchmark throughput by more than 10%. The experimental results also show that detections by a failed NIDS are took over by other NIDSs within 100 seconds. This is much faster than recovering manually by administrator.

目 次

第1章　序論 **1**
1.1　背景 1
1.2　課題と目的 3
1.3　提案 5
1.3.1　レイヤ7 NIDS/NIPS のための TCP ストリーム再構成機構: Store-through 方式 5
1.3.2　NIDS の協調による性能向上と耐障害性向上: Brownie 7
1.4　本研究の貢献 8
1.5　本論文の構成 9

第2章　関連研究 **10**
2.1　検知精度向上を目的とした研究 10
2.1.1　TCP ストリーム再構成機構 13
2.2　性能向上を目的とした研究 20
2.3　耐障害性向上を目的とした研究 21

第3章　レイヤ7 NIDS/NIPS のための TCP ストリーム再構成機構 **22**
3.1　レイヤ7 コンテキストを考慮した NIDS/NIPS 22
3.1.1　レイヤ7 コンテキストとは 22
3.1.2　レイヤ7 コンテキストを考慮したネットワーク侵入検知 . . 24
3.2　レイヤ7 NIDS のための TCP ストリーム再構成機構の要件 28
3.3　Store-through 方式 29
3.3.1　システム・アーキテクチャ 30
3.3.2　Store-through の動作 30
3.3.3　パケット管理 34
3.3.4　コネクション管理 35
3.3.5　IP フラグメンテーションの扱い 36

3.4 実装 . . . 39
3.4.1 TCP ストリーム再構成機構の呼び出し . . . 39
3.4.2 コネクション管理 . . . 39
3.5 議論 . . . 42
3.5.1 実装の考察 . . . 42
3.5.2 Denial-of-Service Attack . . . 43
3.5.3 Evasion 攻撃 . . . 44
3.6 実験 . . . 45
3.6.1 実験環境 . . . 45
3.6.2 ベンチマーク . . . 46
3.6.3 実験結果：性能 . . . 47
3.6.4 実験結果：CPU とメモリの使用率 . . . 48
3.6.5 Store-through と Stop-forward の比較 . . . 52
3.7 まとめ . . . 60

第 4 章 NIDS の協調による性能向上と耐障害性向上 62

4.1 提案 . . . 62
4.1.1 概要 . . . 62
4.1.2 性能向上 . . . 63
4.1.3 耐障害性 . . . 66
4.1.4 性能向上と耐障害性向上のトレードオフ . . . 66
4.1.5 本手法導入による影響の可能性 . . . 67
4.2 設計と実装 . . . 68
4.2.1 性能向上 . . . 68
4.2.2 耐障害性向上 . . . 75
4.2.3 警告ログの収集 . . . 76
4.3 実験 . . . 76
4.3.1 実験環境 . . . 76
4.3.2 性能向上：ベンチマーク . . . 77
4.3.3 性能向上：実トラフィック . . . 81
4.3.4 耐障害性向上 . . . 84
4.4 まとめ . . . 89

第 5 章　結論 **90**
5.1　本研究のまとめ 90
5.2　今後の展望 92

謝辞 **93**

論文目録 **95**

参考文献 **97**

図目次

1.1 NIDS を取り巻く環境の変化，本研究の課題と目的，提案手法の関係 4

2.1 本研究の関連研究 10
2.2 プロキシ方式 15
2.3 Stop-forward 方式 16

3.1 簡単な HTTP の流れ 23
3.2 システム・アーキテクチャ 29
3.3 Store-through 方式の動作 31
3.4 実験のネットワーク設定 45
3.5 実験結果：性能（0% フラグメンテーション） 49
3.6 実験結果：性能（1% フラグメンテーション） 50
3.7 実験結果：性能（10% フラグメンテーション） 51
3.8 実験結果：CPU とメモリの使用率（0% フラグメンテーション） . . 53
3.9 実験結果：CPU とメモリの使用率（1% フラグメンテーション） . . 54
3.10 実験結果：CPU とメモリの使用率（10% フラグメンテーション） . 55
3.11 token bucket filter を用いたローカルネットワークでの輻輳ウィンドウサイズの変化 58
3.12 インターネット経由での輻輳ウィンドウサイズの変化 59

4.1 NIDS の設置例 63
4.2 実験環境 77
4.3 実験結果：過負荷 NIDS の負荷軽減 (ベンチマーク，初期設定: DOWN) 78
4.4 実験結果：冗長ルール削除 (ベンチマーク，初期設定: BOTH) 80
4.5 実験結果：過負荷 NIDS の負荷軽減 (実トラフィック，初期設定: UP) 83
4.6 実験結果：冗長ルール削除 (実トラフィック，初期設定: BOTH) . . . 85
4.7 実験結果：耐障害性 (初期設定: BALANCED，上流 NIDS を停止) . 87
4.8 実験結果：耐障害性 (初期設定: BALANCED，下流 NIDS を停止) . 88

表 目 次

2.1 TCP ストリーム再構成機構の各方式の比較 14

3.1 MF フラグ，フラグメント・オフセット (FO) とフラグメントの関係 37
3.2 実験環境 46
3.3 送信所要時間の比較 (5 回測定の平均) 57

4.1 セキュリティ確保を考察する 4 つのケース 71
4.2 ルールが削除された場合の処理 75

第1章　序論

1.1　背景

インターネットが我々の生活に不可欠な社会基盤となっている一方，インターネット上のサーバを狙ったリモート攻撃は後を絶たない．リモート攻撃とは，攻撃者がインターネットを通じて攻撃メッセージを脆弱性のあるサーバに送信することで，被害を発生させる攻撃のことである．リモート攻撃によってサーバが攻撃されると，サーバは通常のサービスを提供できなくなる他，他のサーバへの攻撃の踏み台として悪用されるなど，さらなる被害を引き起こす.

さらに，近年のリモート攻撃は大規模な被害を起こすことがある．例えば，コンピュータウィルスはリモート攻撃を用いて，インターネット上の多数のコンピュータに瞬時にして感染を広げる．これにより，インターネットの正常な稼働を妨げ,大規模な損害を発生させる．過去には，2001 年に CodeRed [1] と呼ばれるワーム[1]がリモート攻撃を利用して 39 万台のホストに感染し，少なくとも 26 億ドルの経済的損失があったと推定されている [2]．また，2003 年には Slammer [3] と呼ばれるワームが 75,000 台以上のホストに感染したことで，10 億ドルの被害がでた [4].さらに，2008 年には Windows Server Service RPC の脆弱性 [5] を用いる Conficker (Downadup) [6] と呼ばれるワームが出現して 1,500 万台以上に感染し，91 億ドルの被害になったと推定されている [7, 8].

こうした不正攻撃への対策として，ネットワーク侵入検知・防御システム（NIDS/NIPS: Network Intrusion Detection/Prevention Systems）が提案され，広く利用されている [9, 10]．ネットワーク越しの攻撃の多くは，サーバに送信されてくるメッセージに攻撃コードを埋め込み，サーバの脆弱性を悪用して攻撃コードを実行することによって行われる．そのため，NIDS/NIPS はサーバに送信されてくるメッセージを検査することによって攻撃を検知し，サーバを攻撃から防御する．NIDS は，ネットワーク上からやってくるメッセージを検査し，攻撃であれば

[1] ウィルスが他のファイルに感染することで増殖するのに対し，独立したプログラムとして動作するものワームと呼ぶ.

管理者に警告を発することで，管理者が早期に攻撃に対する対策を行えるようにする．NIPS は，NIDS と同様にメッセージを検査し，攻撃ならばそのメッセージを破棄するなどの対策をとることによって，攻撃の成功を妨げる．文献 [9] によれば，2005 年の時点でも NIDS の導入率は 64% と多くの企業が NIDS を導入している．また，文献 [10] で指摘されている通り，NIDS と NIPS の機能の統合化や，NIDS から NIPS への置き換えも進みつつあり，NIDS だけでなく NIPS の重要性も増してきている．なお，本論文ではこれ以降，特に区別が必要な場合を除いて，NIDS/NIPS を単に NIDS を表記する .

NIDS における攻撃検出の手法は大別すると，シグネチャ型 (signature-based detection) と異常検知型 (anomaly-based detection) の 2 種類に分類できる．シグネチャ型 NIDS では，攻撃を表すシグネチャとして，攻撃メッセージに含まれるバイトパターンをあらかじめ定義しておく．そして，ネットワーク上を流れるパケットとシグネチャを比較し，シグネチャと同じバイトパターンを含む場合に攻撃であると判断する．シグネチャを追加していくことで検知できる攻撃のパターン数を増やすことができるが，シグネチャが定義されていない未知の攻撃は検知できない．シグネチャ型 NIDS としては，Snort [11] や Bro [12, 13] が知られている．一方，異常検知型 NIDS は，正常時のネットワークトラフィックの特性をあらかじめ統計的に学習しておく．そして検査時は，学習した特性から大きく外れているトラフィックを攻撃として検知する．シグネチャ型と異なり，学習時の特性から外れていれば未知の攻撃も検知できるが，正確に攻撃と正常の区別を行うためにどのようなトラフィックを正常時のトラフィックとして与えれば良いかの判断が難しい．実運用時の正常なトラフィックとは異なるトラフィックを，正常トラフィックとして学習すると，誤検知が増える原因となる．異常検知型 NIDS として，HTTP リクエストに着目した手法 [14] や，ペイロードに着目した PAYL [15] などがある．本研究では，商用・オープンソース双方で広く用いられている手法であるシグネチャ型 NIDS を対象とする．

NIDS の歴史は古く，そのコンセプトは 1980 年代から存在し，システムも 1990 年代には登場している．例えば，最初の商用 NIDS は 1994 年に登場し，現在広く用いられているオープンソースのシグネチャ型 NIDS である Snort は 1998 年に公開されている [16, 17]．しかしながら，近年の NIDS を取り巻く環境は初期の頃よりも厳しい．その要因として，次の 3 つの NIDS を取り巻く環境の変化が挙げられる．まず第一に，初期に比べて攻撃が複雑で巧妙になっていることである．攻撃者は，NIDS による攻撃の検知を回避するために，巧妙に攻撃メッセージを作成

するようになってきている．1 つのメッセージではなく，複数のメッセージを組み合わせることで攻撃となる場合もある．第二に，インターネットの普及に伴って，ネットワークトラフィックの量と速度が増加していることである．ネットワークトラフィックの量と速度の増加は，NIDS が単位時間あたりに検査しなければならないトラフィックの量が増えることを意味する．第三に，汎用 PC を用いて NIDS を構成することが多くなっているということである．ASIC 等専用のハードウェアで NIDS を構成することも可能だが，高価になりやすく汎用性に乏しい．そのため，多くの NIDS はソフトウェアで実装され，汎用 PC を元にしたハードウェア上で動作する構成となっている．

1.2 課題と目的

前述した 3 つの環境の変化により，NIDS の新たな課題として，検知精度の向上，性能向上，耐障害性向上の 3 つが出てきている．本研究の目的は，これらの 3 つの課題を解決する手法を提案することである．この節では，それぞれの環境の変化により出てきた課題について述べる．環境の変化と課題の関係を図 1.1 に示す．

1 つ目の環境の変化は，攻撃の複雑化・巧妙化である．このため，従来のシグネチャ型 NIDS では検知が難しい攻撃が増えており，NIDS の攻撃検知精度を向上させる必要がある．従来のシグネチャ型 NIDS では，シグネチャと一致するバイトパターンがトラフィックに含まれれば，攻撃と判断する．そのため，攻撃者は巧妙に攻撃メッセージを作成することで，NIDS による検知を回避しようとする．例えば，複数のメッセージに攻撃を分割することで，パケット毎にシグネチャとマッチングして攻撃検知をする NIDS を回避することができる．また，攻撃の意味を変えずにメッセージのバイト列を変更することによって，攻撃検知を回避する亜種も存在する．一方で，様々な攻撃を検知するためにより汎用的なシグネチャを定義すると，今度は攻撃でないメッセージも攻撃と誤検知するフォールス・ポジティブが発生する．フォールス・ポジティブによる不必要な警告が多発すると，管理者が真の攻撃に気付きにくくなるという問題がある．そのため，攻撃の複雑化に対応し，NIDS の検知精度を向上する必要がある．

検知精度を向上させた結果，攻撃検知のための処理が増え，NIDS の負荷が大きくなるという性能面での課題もある．通常，検知手法が高度化すると，メッセージの検査処理の負荷が増加する．例えば，正規表現を用いたマッチングは，単純

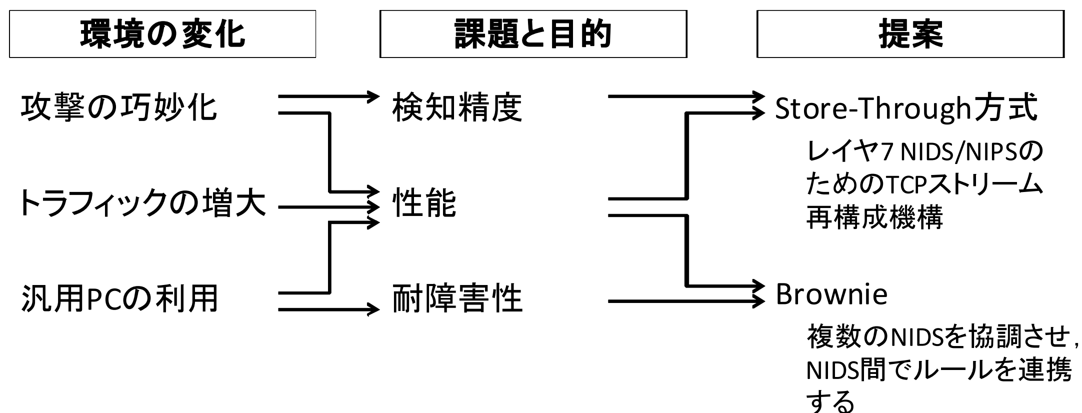


図 1.1: NIDS を取り巻く環境の変化，本研究の課題と目的，提案手法の関係

なバイトパターンのマッチングよりも検知精度は高いが，正規表現マッチングのための処理が追加で必要となる．また，複数のメッセージにまたがる攻撃を検知するためには，それまでのメッセージの状態を記憶しておく必要がある．すなわち，攻撃の高度化に対処した場合でも，性能低下を最小限にする必要がある．

2 つ目の環境の変化は，ネットワークトラフィックの量と速度の増加である．まず，ネットワークの通信速度が増大していることから，NIDS はその通信速度を損なわない性能が必要とされている．また，インターネットの利用人口が増大していることから，トラフィックの量も増大している．トラフィックの内容についても，以前はテキストや画像等比較的サイズが小さいデータが中心であったが，最近では動画等サイズが大きいデータが増えていることが，トラフィック量の増大に拍車をかけている．従って，NIDS はより多くのトラフィックをより速く検査しなければならない，という課題も解決しなければならない．

3 つ目の環境の変化は，汎用 PC を用いた NIDS が一般に利用されるようになった，ということである．NIDS の攻撃検知処理は，シグネチャとメッセージのマッチングであるため，CPU インテンシブである．しかしながら，CPU の高速化のトレンドは，ネットワークトラフィックの増大におけるトレンドよりも緩やかである．そのため，性能向上も課題としてあげなければならない．また，汎用 PC を元に構成されているため，ハードウェア障害等により，NIDS に障害が起こることは避けられない．障害が起こっても，攻撃の検知を継続できる耐障害性向上の仕組みが必要である．

1.3 提案

本研究では，近年の NIDS を取り巻く環境の変化に伴って浮上した，検知精度・性能・耐障害性の 3 つの課題を解決する手法を提案する．課題と提案手法の関係を図 1.1 に示す．まず，性能を低下させずに検知精度を向上するために，レイヤ 7 NIDS/NIPS のための TCP ストリーム再構成機構の方式として，store-through 方式を提案する．次に，性能や耐障害性を向上するために，組織ネットワーク内に複数の場所に置かれた NIDS 同士を協調させ，NIDS 間でルール設定を調整する手法を提案する．前者は単体の NIDS を対象として手法であり，後者は複数の NIDS を協調させる手法であるため，これらは補完的な関係となっている．

1.3.1 レイヤ 7 NIDS/NIPS のための TCP ストリーム再構成機構: Store-through 方式

本研究ではまず，レイヤ 7 コンテキストを考慮した攻撃検知を行うレイヤ 7 NIDS/NIPS のための TCP ストリーム再構成機構の方式として，store-through 方式を提案する．本手法により，性能を低下させずに検知精度を向上させることが可能となる．

従来のシグネチャ型 NIDS では，シグネチャと一致するバイトパターンがトラフィックに含まれれば，攻撃と判断する．そのため，複数のメッセージにまたがるなどの複雑な攻撃は検知ができない．また，様々な攻撃を検知するために，より汎用的なシグネチャを定義すると，今度は攻撃でないメッセージも攻撃と誤検知するフォールス・ポジティブが発生する．そこで，近年単純なバイトパターンのマッチングでなく，レイヤ 7 コンテキストを考慮することにより検知精度を高めるレイヤ 7 NIDS が提案されている．レイヤ 7 コンテキストとは，1) メッセージの送受信順序，2) 各メッセージの構文（メッセージ・フィールドがどのような順で並ぶか），3) 各フィールドのフォーマット（フィールドがどのような文字から成るか）のことである．メッセージ・フォーマットやメッセージの送受信順序を考慮することによって，そのバイト列がメッセージのどの部分に現れるか等，より多くの情報を用いてメッセージの検査を行うことができる．そのため，単なるバイト列とのマッチングよりも，精度の高い攻撃検知が可能になる．

レイヤ 7 コンテキストを考慮した攻撃検知を行うためには，ネットワーク上を流れる個別のパケットを，メッセージに再構成する TCP ストリーム再構成機構が必

要となる．すなわち，順番が入れ替わって届いたパケットは正しい順番に並べ直し，IP フラグメントされたパケットは再構成しなければならない．レイヤ 7 NIDS/NIPS のための TCP ストリーム再構成機構が満たさなければならない要件は，以下の 4 つである．

まず第一に，**完全な防御**ができる必要がある．NIPS においては，悪意のある攻撃メッセージが攻撃対象のアプリケーションに届き，その結果攻撃が成功するということを防ぐことができなければならない．すなわち，NIPS があるメッセージを攻撃だと判断したら，そのメッセージが攻撃対象となっているアプリケーションに渡ることがあってはならない．この要件は，攻撃を防御するという観点から非常に重要である．攻撃メッセージが攻撃対象アプリケーションに渡った後に NIPS が行動を起こしても，そのアプリケーションは既に攻撃されている可能性があるからである．

第二に，**性能**である．NIDS による性能のオーバヘッドはできる限り少なくするべきである．NIDS を使用することによって，性能が大きく低下する場合，管理者は NIDS の使用を躊躇したり使用をやめる可能性がある．理想的には，TCP ストリーム再構成機構の性能は，単に IP 層で転送を行う場合の性能に，できるだけ近づけるべきである．

第三に，**アプリケーション透過性**を保持する必要がある．アプリケーション透過性とは，NIDS を新たに設置する際，サーバやクライアント・アプリケーションの変更や再設定が必要ないことをいう．アプリケーションの変更や再設定が必要であると，管理者は NIDS の導入を躊躇する可能性もあるため，アプリケーション透過性を保持することは重要である．

最後に，**トランスポート透過性**を保持する必要がある．トランスポート透過性とは，NIDS が TCP フローや輻輳制御に与える影響が無視できるほどに小さいことをいう．NIDS は end-to-end の TCP/IP の振る舞いに与える影響を少なくするべきである．単純な実装方法では，end-to-end の TCP/IP の振る舞いに影響を与えてしまい，エンドホストの TCP/IP プロトコル・スタックが通常と異なる振る舞いをするおそれがある．

本論文では，これらを全て満たす TCP ストリーム再構成機構として，*store-through* 方式を提案する．Store-through 方式では，個別のパケットからメッセージに再構築する際，順番が入れ替わったパケットはコピーをとってすぐに転送することで，トランスポート透過性を保持する．また，攻撃だと判断された時点で，後から到着したパケットを破棄することで，攻撃の成功を妨げ，完全な防御を達成する．ま

た，IP レベルで実装することで性能とアプリケーション透過性を達成する．本研究ではプロトタイプを Linux 上に実装して実験を行うことにより，オーバヘッドが小さいこと，トランスポート透過性が保持できることを示す．

1.3.2　NIDS の協調による性能向上と耐障害性向上: Brownie

次に，性能や耐障害性の向上を目的として，組織ネットワーク内に複数の場所に置かれた NIDS 同士を協調させ，NIDS 間でルール設定を調整する手法を提案する．大学や企業など多くの組織では，組織内ネットワークとインターネットの境界のみでなく，内側のネットワークの様々な階層に複数の NIDS を設置している場合が多い．例えば大学では，大学の入り口と外のネットワークとの境界以外にも，各学部や各研究室で個別に NIDS を設置していることがある．

そこで，組織内に置かれた複数の NIDS を協調させ，NIDS 間でのルール設定を調整するシステム Brownie を提案する．そして，複数の NIDS が協調することで，性能向上や耐障害性向上ができることを示す．Brownie では，組織ネットワーク内に置かれた NIDS 間でルール設定を交換し合い，定期的に負荷情報をやりとりすることで，NIDS 同士のルール設定を連携させる．そして，性能を向上するために，過負荷になった NIDS の負荷を減少させ，NIDS 間の冗長なルール設定を削除するようにルールを再設定する．また，耐障害性を向上するために，NIDS の障害時には障害が発生した NIDS で有効にしていたルールを，別の NIDS で有効にすることで攻撃検知を代替する．本研究では，Brownie のプロトタイプを実装し，ルール設定を連携することで，性能が向上することと耐障害性が向上することを，実験によって示す．

なお，性能と耐障害性は，基本的にはトレードオフの関係となる．性能向上のためには処理の分散化が，耐障害性向上のためには処理の冗長化が必要だからである．そのため，本機構では，管理者が性能重視や耐障害性重視の設定を柔軟に選択できるようにした．性能向上を最優先する場合には，NIDS が過負荷にならないように複数の NIDS 間で完全に処理を振り分け，負荷分散を行う．この場合，処理の冗長性を取り除くことになり耐障害性は低下するが，本機構により他の NIDS が肩代わりすることで，短い時間で攻撃の検知を再び行えるようになる．一方，耐障害性を最優先する場合は，NIDS の処理を冗長化するという手法をとる．極端に言えば，全ての NIDS で全て同じ処理を行えば，完全冗長となり耐障害性は上がる．しかしこの場合，全ての処理を有効にすることで，NIDS は過負荷になりやす

くなり性能は低下する．これらの中間として，処理を部分的に冗長化させることで，一定の耐障害性を保ちつつ性能低下を抑える設定もありえる．例えば，よく攻撃を検知するルールや危険性の高いルールのみを冗長させる，という方法がある．

1.4　本研究の貢献

本研究の貢献は，NIDS/NIPS の実装に関して，近年の環境の変化に伴う課題を解決する手法を提案し，NIDS/NIPS をより有用なシステムにすることである．

まず，検知精度を向上することで，NIDS がより多くの攻撃を検知・防御することができるようになり，リモート攻撃による被害の削減が期待できる．本研究の提案手法により，検知精度を向上させる手法であるレイヤ 7 コンテキストを用いた攻撃検知が，より有用になる．従来のレイヤ 7 NIDS では，検知精度の向上は達成できても，検知した攻撃を完全に防御できない可能性があったり，性能が低下するという問題があった．本研究で提案する store-through 方式の TCP ストリーム再構成機構により，レイヤ 7 コンテキストを用いた NIDS を効率的に実装することができるようになり，その有用性が高まる．

次に，本研究では組織内のネットワーク上に配置された NIDS を連携させることで，性能向上と耐障害性向上を行う．性能を向上することで，NIDS の監視下に置かれたサーバやクライアントマシンの利用者が受ける，NIDS による性能低下の影響を少なくすることができる．NIDS の性能が低いと，NIDS で検査すべきパケットが破棄され，適切に攻撃を検知できなくなる．また，パケットの破棄・改変を行う NIPS では，NIPS の処理スループットが低いとその NIPS を通るネットワーク・トラフィックの通信遅延が増大し，サーバのサービス提供やクライアントのインターネット通信の応答性が低下する．本研究で提案する NIDS 協調システム Brownie により，ネットワーク内の NIDS を連携させることで，ネットワーク内全体の性能向上を行い，サーバやクライアントユーザが受ける性能低下を減少させることができる．さらに，本手法は単体の NIDS の性能を向上させる手法 [18,19,20] と組み合わせることで，さらに性能を向上することも可能である．

また，耐障害性を向上することで，障害による攻撃の検知漏れを防ぐことができ，リモート攻撃による被害を小さくすることができる．従来，ある NIDS が障害により停止すると，その NIDS が監視していたネットワークトラフィックは攻撃に対して素通りとなる．本研究では，複数の NIDS を連携させることで，ある NIDS

が障害により攻撃を素通りさせた場合でも，他のNIDSが代わりに検知を行う手法を提案する．これにより，あるNIDSが障害により攻撃検知が不可能になっても，他のNIDSによって攻撃検知を継続することができるようになる．

1.5 本論文の構成

本論文は全5章からなる．第1章では本研究の背景，動機および目的について述べ，本研究の学術的貢献について説明した．

第2章では本研究の関連研究をまとめる．NIDSの課題である，検知精度・性能・耐障害性のそれぞれについて既存手法をまとめ，それらとの違いを明らかにする．

続く2章で，検知精度・性能・耐障害性のそれぞれの課題に対する本研究の提案手法を説明する．まず，第3章では，検知精度向上のために，レイヤ7コンテキストを考慮した攻撃検知を行うNIDS/NIPSについて述べる．レイヤ7コンテキストを考慮した検知を行うために必要となるTCPストリーム再構成機構の要件を述べたあと，提案手法store-through方式について説明する．さらに実験を行い，store-through方式が要件を満たすことを示す．

次に，第4章では，組織ネットワーク内に複数の場所に置かれたNIDS同士を協調させることで，性能向上と耐障害性向上を行う手法を提案する．複数のNIDSを協調させるシステムBrownieの概要を述べた後，Brownieを用いた性能向上と耐障害性向上についてそれぞれ手法を述べる．また，実験を行い，性能向上と耐障害性向上が行えることを示す．

最後に第5章で本論文をまとめ，今後の研究の方向性を示す．

第2章　関連研究

本章では，本研究の関連研究についてまとめる．図 2.1 に本研究と関連研究の関係を示す．第 1 章で本研究の課題としてのべた検知精度・性能・耐障害性の 3 つの課題について，それぞれまとめる．

2.1　検知精度向上を目的とした研究

現在広く利用されている NIDS の多くは，シグネチャを用いて攻撃を検知する．シグネチャによる攻撃検知では，攻撃メッセージの特徴をあらかじめシグネチャとして定義しておき，メッセージがシグネチャと一致した場合に攻撃と判断する．商用・オープンソース共に，広く用いられている方式である [21]．シグネチャによる攻撃検知は，そのシンプルさ故に，なぜ攻撃と判断されたか，どの攻撃を検知したか等が分かりやすく，シグネチャを増やしていくことで検知できる攻撃を増やすことができるという利点があるためである．

初期のシグネチャ方式による検知は，攻撃に含まれるバイトパターンを定義し，ネットワークを流れるパケットと定義されたバイトパターンが完全一致するかどうかを判断していた．このような NIDS として，Snort [11] が広く知られている．

	検知精度向上		性能向上	耐障害性向上
単体の NIDS	レイヤ7 NIDS (Bro, Shield等) Byte-code emulation (SigFree等) 異常検知型	← TCPストリーム再構成機構 Store-Through方式 メッセージに再構成する機能を提供	アルゴリズムによる高速化手法 ハードウェアによる高速化手法	--
複数の NIDS	DOMINO等		NIDS Cluster等 組織内NIDSの協調: Brownie	Kuangら, Siqueiraらによる研究 （組織内NIDSの協調: Brownie）

図 2.1: 本研究の関連研究

しかし，バイトパターンとの完全一致による検知では，誤検知が多発するという問題がある．もし，ある攻撃に合致するバイトパターンを厳密に定義すると，類似の攻撃を検知できず見逃す可能性がある．攻撃者は，NOP 等意味のないバイト列を挿入したり，元の攻撃と同じ意味を持つコードを別のバイト列で表現する等の方法により，多くの種類の亜種を作成するためである．また，逆に粗いシグネチャでは，攻撃でない通常のメッセージも攻撃と誤検知するフォールス・ポジティブが発生する．実際には，厳密なシグネチャを記述することは難しく，多くのシグネチャが粗いシグネチャとなってしまっている．そのため，フォールス・ポジティブによる不必要な警告が多発し，管理者が本当の攻撃に気付きにくくなるという問題がある．

そこで，Bro [12] はレイヤ 7 コンテキストを導入することで，検知精度を上げる手法を提案している [13]．具体的には，シグネチャの記述に正規表現を用いることと，コネクションの状態を持つことの 2 つを提案している．正規表現を用いることにより，攻撃を表すバイトパターンが，メッセージの中のどの部分に含まれるか，すなわちレイヤ 7 のプロトコルにおいてどのフィールドに含まれるかを記述することが可能となる．これにより，攻撃を表すバイト列が，メッセージ中の特定のフィールドに含まれるときのみ成功する攻撃を，検知することができる．また，コネクションの状態を持つことにより，複数のシグネチャを関連づけ，それら全てに合致したとき（又は合致しなかったとき）に警告を発することで誤検知を減らすことができる．

Shield [22] は，フィールドの内容やメッセージの送受信順序などのレイヤ 7 コンテキストを用いて，既知の脆弱性を攻撃するメッセージ列を定義できるようにすることで，サーバを攻撃から防御する手法を提案している．例えば，HTTP リクエストの URI フィールドのうち，引数部分 (?以降の文字列) の長さを調べることにより，CodeRed が対象とする脆弱性 [1] を攻撃するメッセージを検知する．また，メッセージの送受信を状態遷移として扱うことで，複数のメッセージを送受信したことによる攻撃検知を可能としている．WebSTAT [23] は，HTTP を利用した web サーバ向けの攻撃の検知に特化したシステムである．WebSTAT では，web サーバのアクセスログを対象に，正規表現や状態遷移を用いることで攻撃を検知する．

近年では Snort も正規表現を用いてシグネチャを記述できたり，HTTP リクエスト内の URI フィールド等，一部のフィールドについては他のデータとは別に検査しており，レイヤ 7 コンテキストを用いた検知が可能となっている．Application Intelligence [24] や IntruShield [25] といった商用の NIDS でも，レイヤ 7 コンテキ

ストを用いて攻撃の検知を行うものがある．

また，シグネチャを用いずに攻撃メッセージを検知するため，メッセージ中に攻撃メッセージとしての特徴があるかを検査する手法も多く提案されている．これらは主に攻撃メッセージの特徴を統計的に定義する異常検知型が多い．PAYL [15] では，メッセージ中に含まれるバイト毎の出現率を調べ，その分布が統計的に通常と異なる場合に攻撃であると判断する異常検知型のシステムである．しかし，この手法はどのメッセージも同様に扱うため，用いられているプロトコルによっては有効でない場合がある．例えば，HTTP のようなテキスト主体のプロトコルは，バイト毎の出現率がある程度偏りがあるため有効である．しかし，FTP のように任意のバイト列を送信する可能性がある場合には，バイト毎の出現率から攻撃メッセージを検出することは難しい．

web サーバ向けの攻撃に特化した異常検知型のシステムの提案も多い [14,26,27]．脆弱性や攻撃の全体の中で，web に関係するセキュリティ問題が占める割合が大きいためである．例えば，Kruegel と Vigna [14] は，HTTP クエリのパラメータを統計的に分析することで異常を検知する，異常検知型の NIDS を提案している．HTTP に特化し，HTTP プロトコルのコンテキストを利用することで検知精度を高めているが，他のプロトコルに簡単に応用できるものではない．これらは，サーバ・ログを利用することで，レイヤ 7 コンテキストを利用した攻撃の検知を可能にしている．

また，メッセージ内の攻撃コードを解析することで検知をおこなうシステムもある．SigFree [28] では，メッセージを逆アセンブルして制御フローグラフを作り，メッセージ中に閾値以上の連続した実行可能な命令列があれば攻撃と判断する．Network-level Code Emulation [29, 30, 31] では，メッセージを攻撃コードの命令列と仮定して，命令列を疑似実行することで，攻撃コードを検出する．これらの手法は，レイヤ 7 コンテキストを用いた検知ではないが，パケットではなくメッセージ単位で攻撃検知をおこなっているため，本研究で提案するような TCP ストリーム再構成機構が必要となる．

このように，検知精度向上のためにレイヤ 7 コンテキストを利用するなど，パケット毎ではなくメッセージを検査することは，広く受け入れられてきている．これらの研究は主にどのように攻撃を検知するかに着目している．しかし，ネットワークの中間地点に置かれる NIDS がレイヤ 7 コンテキストを利用した検知を行うためには，ネットワークを流れるパケットをメッセージに変換するという処理が必要となる．本研究では，これらのレイヤ 7 コンテキストを利用した検知に必

要な TCP ストリーム再構成機構の実装について議論し，その手法を提案する．レイヤ 7 コンテキストを利用した検知を行うこれらの NIDS が用いている TCP ストリーム再構成機構と，本研究で提案する TCP ストリーム再構成機構との比較は後述する．

検知精度を向上させる別のアプローチとして，ネットワーク上に多くの NIDS を設置するという手法がある [32]．ネットワーク上の様々な場所に NIDS を分散して置き，それぞれの NIDS で発生した警告を収集することで，1 台の NIDS では検知できない攻撃を検知する [33, 34]．例えば，DOMINO [33] ではネットワーク上に分散した NIDS が，互いに P2P プロトコルで NIDS の警告を交換する．他の NIDS で検出した警告を用いることにより，1 台の NIDS で 1 つのローカルネットワークを監視しているだけでは検知不可能な，インターネット規模の攻撃を検知することができる．さらに，収集した攻撃や警告を元に新しい検知ルールを作成し，各 NIDS に配布することによって，新たな攻撃を検知する手法も提案されている [35]．しかし，これらのシステムでも，分散された各々の NIDS での攻撃の検知精度が低いと，全体の攻撃検知の精度も低くなる．そのため，個々の NIDS の攻撃検知精度を向上させる必要がある．

シグネチャ型 NIDS の検知精度向上の別の観点として，シグネチャの自動生成がある [36, 37]．シグネチャ型 NIDS では，シグネチャを個々の攻撃に対応して作成しなければならないためである．これらのシステムでは，攻撃に共通するバイト列をシグネチャとすることで，新しい攻撃に対して自動的にシグネチャを作成する．例えば，HoneyComb [36] は，サービスを提供しない，おとりホストであるハニーポットを用いて攻撃メッセージを収集し，同じポート番号で収集されたメッセージの共通バイト列をシグネチャとする．また，EarlyBird [37] は内部ネットワークから共通バイト列を持つメッセージが大量に発信された際に，その共通バイト列をシグネチャとする．しかし，これらは，共通バイト列を取り出す手法であるため，生成されたシグネチャには攻撃に関係ない部分のバイト列が含まる場合がある．そのため，人手で作成された通常のシグネチャより精度が低くなるという問題がある．

2.1.1　TCP ストリーム再構成機構

本節では，TCP ストリーム再構成機構の従来の方式を取り上げ，第 1 章で述べた TCP ストリーム再構成機構の要件が満たされてないことを示す．表 2.1 に本節で比

表 2.1: TCP ストリーム再構成機構の各方式の比較（◯ は要件を満たしていること，× は満たしていないことを示す.）

	完全な防御	性能	アプリケーション透過性	トランスポート透過性
プロキシ	◯	×	×	×
Stop-forward	◯	◯	◯	×
Bro や Snort	×	◯	◯	◯
提案手法 (Store-through)	◯	◯	◯	◯

較する各手法，および我々の提案手法である store-through 方式をまとめた．TCP ストリーム再構成機構の要件である，完全な防御，性能，アプリケーション透過性，トランスポート透過性の全てを満たす従来の手法はない．我々の提案手法である store-through は，この 4 つの要件全てを満たす.

プロキシ方式

レイヤ 7 NIDS の簡単な実装方法として，プロキシを用いる手法がある．図 2.2 に示すように，プロキシはサーバとクライアントの間に置かれ，その通信を仲介する．プロキシは，サーバとクライアント双方とコネクションを 1 つずつ，計 2 つのコネクションを確立して通信の中継をする．全てのパケットはオペレーティング・システム（OS）内にある TCP/IP プロトコル・スタックを通過してメッセージに再構成されるため，NIDS 自身はパケットを扱う必要がない．別の言い方をすれば，OS 内の TCP/IP プロトコル・スタックが TCP ストリーム再構成機構の役割を果たしているのである．プロキシ方式を用いているシステムとして，SigFree [28] がある．SigFree は，プロキシを用いてメッセージの再構築をした上で，逆アセンブルをすることで攻撃を検知する.

しかしながら，プロキシによるレイヤ 7 NIDS の実現は，性能，アプリケーション透過性，トランスポート透過性の 3 点において充分であるとは言えない．まず，プロキシを用いた実装では，すべてのパケットが 2 回ずつ TCP/IP プロトコル・スタックを通過するため，性能が良いとはいえない．プロキシでは，受信したパケットをプロトコル・スタックを通してユーザ空間で動作するセンサに渡す．センサでは渡されたメッセージを検査して，攻撃でないと判断すると，転送のためにもう一度プロトコル・スタックを通す．そのため，プロキシによる方法では，全て

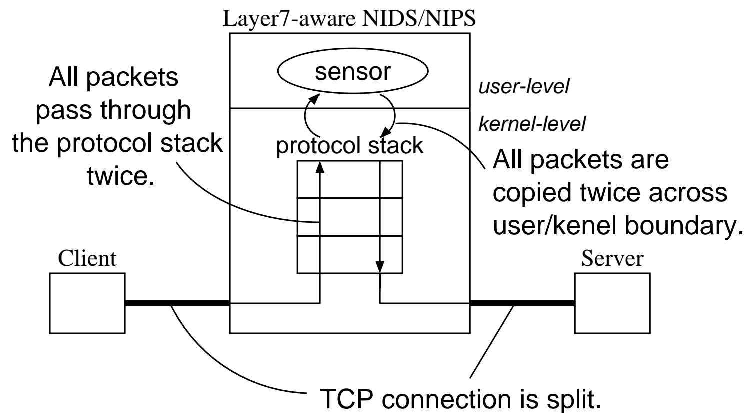


図 2.2: プロキシ方式

のパケットがプロトコル・スタックを 2 回通らなければならない．それに加えて，全てのパケットについてカーネルとユーザ空間間のデータ・コピーも 2 回行わなければならない．

次に，プロキシ方式では，アプリケーション透過性が保持されない．NIDS をプロキシとして実装すると，サーバやクライアントで実行されるアプリケーションの変更や再設定が必要となる．例えば，クライアント・アプリケーションは，サーバではなく NIDS に接続するように設定し直さなければならない．元々のサーバの IP アドレスを，NIDS の IP アドレスとして設定する方法も考えられるが，NIDS は複数のサーバが接続するネットワーク上の上流に置かれることもあり，その場合には管理下のサーバの IP アドレスを NIDS に割り当てることはできない．また，サーバ・アプリケーションの変更が必要となる場合もある．サーバがクライアントの IP アドレスを元にアクセス管理などをしている場合，サーバにとってのクライアントの IP アドレスは，プロキシの IP アドレスのみとなる．そのため，そのままでは適切に管理ができなくなり，サーバ・アプリケーションやクライアント・アプリケーションに何らかの変更が必要となる．

最後に，プロキシ方式はコネクションを分断するため，トランスポート透過性も保てない．例えば，選択的確認応答オプション（SACK: selective acknowledgment）や最大セグメントサイズ（MSS: maximum segment size）などの TCP オプションがサーバとクライアント間で利用できない可能性がある．また，TCP フローや輻輳制御が適切に行われない可能性もある．例えば，サーバ側の TCP 処理が過負荷になったとしても，それと通信しているクライアント側はそれに気づかなかったり，

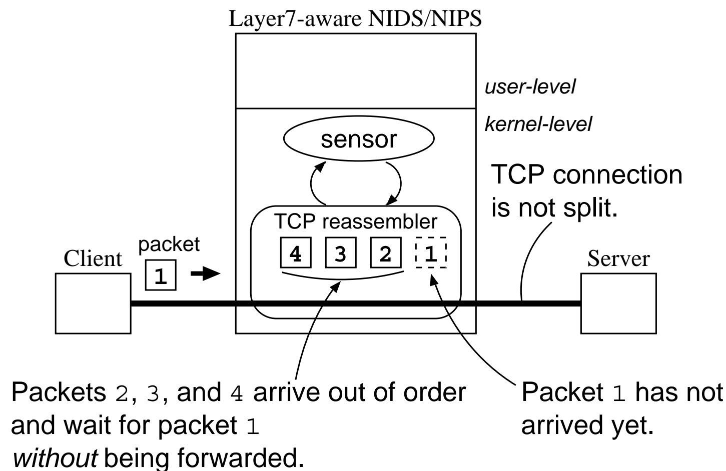


図 2.3: Stop-forward 方式

適切に対応できない可能性がある．

Stop-forward 方式

別の方法として，*stop-forward* という方法が考えられる．プロキシ方式とは異なり，stop-forward はカーネル内で動作し，TCP/IP パケットをそのまま処理する．もし到着したパケットが順番通りであった場合，すぐにセンサに渡してメッセージの検査をする．ここでいう『順番通りであるパケット』とは，TCP ストリーム中で，直前にセンサでチェックしたパケットの次に来るべきパケットを指す．もし順番通りでない，すなわち後ろのパケットが先に到着してしまった場合，前のパケットが到着するまで，先に到着したパケットは転送せずに止める．そして，全てのパケットが揃った時点で，それら全てのパケットをセンサに渡す．センサが検査した結果，攻撃でないと判断したら，最後に本来の宛先に転送する．この動作を図 2.3 に示す．この図では，パケット 1 の前にパケット 2，3，4 が到着してしまっている．そのため，stop-forward 方式ではパケット 1 が到着するまで，先に到着した 3 つのパケットを止めて待つ．その後，パケット 1 が到着すると，パケット 1〜4 の全てのパケットがセンサに渡される．このように，stop-forward 方式では，パケット 1 が到着するまで，先に到着した全てのパケットは転送されることなく NIDS で止められるということになる．stop-forward 方式を用いているシステムとして，Clark ら [38] による，ネットワーク・プロセッサ (NP) と FPGA を用いて，エンド・ホスト上で動作するハードウェア・ベースのネットワーク・ノード IDS (NNIDS)

がある．

Stop-forward 方式は，TCP/IP プロトコル・スタックを通過しない上，カーネルとユーザ空間間のデータ・コピーも発生しない．そのため，stop-forward 方式はプロキシよりも良い性能を期待できる．また，プロキシ方式とは異なり，サーバ・クライアント間のコネクションを分断しないため，アプリケーション透過性も保持できる．クライアントは，本来のサーバと直接 TCP コネクションを確立するため，クライアントとサーバ双方ともアプリケーションの変更や再設定は必要ない．

しかしながら，stop-forward 方式は，トランスポート透過性において欠点を持つ．Stop-forward 方式では，順番通りでなかったパケットは止めるため，通常のネットワーク・トラフィックの振る舞いを乱す．例えば，図 2.3 のパケット１が NIDS の手前で損失したとする．すると，パケット２，３，４はいつまでもサーバに転送されない．その結果，サーバにはパケットが 1 つも届かないため，サーバ側の TCP の輻輳制御機構がネットワークに負荷がかかっていると誤って判断する可能性がある．その結果，実際には不必要であるにもかかわらず，輻輳回避のためのアルゴリズムを開始し，例えばウィンドウサイズを小さくする等のおそれがある．また，順番通りでないパケットを保存するためのバッファ容量が少ないと，多くのパケットを転送せずに破棄することになり，さらにトランスポート透過性に大きな影響を与えるため，十分なバッファ容量を確保しなければならない，という問題もある．

Bro や Snort の TCP ストリーム再構成機構

Bro [12, 13] はレイヤ 7 NIDS であり，TCP ストリーム再構成機構を持っている．また，シグネチャ型 NIDS として有名な Snort [11] でも，複数のパケットにまたがった攻撃の検知ができるように，複数のパケットを再構成する機構を持っている．Bro や Snort の TCP ストリーム再構成機構は，受信したパケットを順番通りであるものも含めて全てコピーし，すぐに本来の宛先に転送する．その後，宛先からの ACK を受け取り再構成するためのパケットがそろっていれば，それらを再構成し，Bro や Snort のセンサに渡して検査をする．

Bro や Snort の TCP ストリーム再構成機構は，性能，アプリケーション透過性，トランスポート透過性を保持している．Stop-forward 方式と同様に，Bro や Snort の TCP ストリーム再構成機構でも，TCP/IP プロトコル・スタックを通さないため，プロキシ方式よりも性能は良いと考えられる．また同様に，TCP コネクションを

分断しないため，アプリケーション透過性も保持される．さらに，stop-forward 方式とは異なり，Bro や Snort の TCP ストリーム再構成機構では受信したパケットを止めることなく，パケットをコピーした後すぐに転送するため，トランスポート透過性も保たれている．

しかしながら，Bro や Snort の TCP ストリーム再構成機構では，完全な防御という点で不十分である．Bro や Snort は NIPS として用いることも可能となっているものの，Bro や Snort の TCP ストリーム再構成機構では，たとえ攻撃を検知したとしても，攻撃を完全に防御することができずに，攻撃が成功する可能性がある．Bro を NIPS として用いる場合，Bro のセンサが攻撃を検知すると，RST パケットを送信し後続のパケットを遮断することで攻撃の防御を行う．そのため，センサが悪意のあるメッセージを検知したときには，既にそのメッセージは攻撃対象のアプリケーションに渡ってしまっている可能性がある．その結果，RST パケットが届く前に，攻撃が成功する可能性があり，攻撃を完全に防御できるとはいえない．

Snort でも同様に，NIPS として用いることのできる inline モードが用意されている．しかしこのモードでは，1 つ 1 つのパケットの検査はパケットを転送する前に行うのに対し，複数のパケットにまたがったメッセージの検査はサーバに届いた後に再構成して行う．そのため，Snort においても，攻撃メッセージが複数のパケットにまたがっていた場合は，攻撃を検知した時点で既に攻撃が成功してしまっている可能性があり，攻撃を完全に防御できるとはいえない．

その他の手法

Shield [22] はメッセージの列を検査していき，既知の脆弱性をつく攻撃メッセージをブロックする．Shield では，TCP ストリーム再構成機構を持っておらず，その代わりに防御したいアプリケーションと同じエンド・ホスト内で動作させなければならない．Shield は防御したい各アプリケーションにリンクする，動的リンクライブラリとして実装されているからである．Shield はエンド・ホストのオペレーティング・システムの上で動作するため，TCP ストリーム再構成機構は必要としない仕組みとなっている．しかしながら，この仕組みでは，いくつものサーバがある場合に，監視ポイントを 1 点にできずに，各サーバマシンに Shield をインストールしなければならないという問題点がある．監視ポイントが各マシンに分散すると，例えば新しいシグネチャ（Shield では Shield ポリシー）を導入する時に，各マシンに対してアップデートを行わなければならなくなる，というように管理

が煩雑になる．ただし，Shieldをセンサとして，本提案手法のstore-through方式のTCPストリーム再構成機構と組み合わせることは可能だと考えられる．

WebSTAT [23] やその他のレイヤ7 NIDS [14, 26, 27] は，webサーバ向けの攻撃に特化した検知システムである．これらは，サーバ・ログを元にすることで，レイヤ7コンテキストを利用した攻撃の検知を可能にしているが，サーバで処理した後の検知であるため，攻撃を防御するということは考えられていない．

TCP Splitter [39, 40] はFPGAを用いてTCPストリームを監視する．TCP Splitterはメッセージ・ストリームを監視するために，順番が入れ替わったパケットは全て破棄するという単純な方法をとっている．こうすることにより，順番が入れ替わったパケットを記憶しておく必要はなくなる．しかし，順番が入れ替わっているとパケットは破棄されるため，TCPフローや輻輳制御に与える影響は大きく，トランスポート透過性を保持できていない．Sugawaraらは，メモリ効率を考慮しつつ複数のパケットにまたがった攻撃を検知するために，順番が入れ替わったパケットの一部のみを保存しておく手法を提案している [41]．しかし，対象とするシグネチャは，長さが一定のバイト列である必要があり，正規表現やレイヤ7コンテキストを用いたシグネチャには適用できない．

また，NSS Group [42] のレポートによると，商用のNIDS/NIPSの中にもTCPストリーム再構成機構を提供しているNIDS/NIPSが数多くある．Symantecのネットワーク侵入防御アプライアンスSNS-7160 [43] は，IPフラグメンテーションの再構成は行っているが，TCPストリームの再構成については述べられていない．Cisco SystemsのIPS 4200シリーズ センサ [44] や，Juniper NetworksのIDP [45] は，データ・ストリームを検査するために，完全なTCP/IPストリーム再構成機構を備えているとしている．Westline SecurityのAthena Aegis IPS [46] が持つTCPストリーム再構成機構は，コピーしたパケットを並べ直し，攻撃を含んでいるパケットは最終的な宛先に転送しないことで，攻撃が完全になるのを妨げるとある．これは，提案手法のstore-through方式に類似している可能性もあるが，それ以上の詳細は公表されていないため，断定できない．他にstore-throughと似た方式を採用しているものもあるかもしれないが，技術的な詳細が公開されていないので，直接比較することは難しい．もしこれらの商用NIDS/NIPSの技術的な詳細を手に入れ，store-through方式との比較が行うことができれば，非常に興味深いと考える．

2.2 性能向上を目的とした研究

NIDSの性能を向上する研究は，様々な観点から長年研究されている．特にNIDSの初期から研究されているのは，シグネチャとのマッチングアルゴリズムである．前節でも述べたとおり，初期のシグネチャ型NIDSはバイトパターンとの完全一致による検知が主流であったため，バイトパターンとの完全一致のマッチングアルゴリズムの高速化手法が提案された．例えば，Aho-Corasick [47]，Commentz-Walter [48], Wu-Manber [49] があり，現在でもこれらやこれらの改良アルゴリズムが広く利用されている [50, 51, 52, 53]．また，Bro [13] によって正規表現でシグネチャを記述することの有用性が提案されたため，正規表現マッチング [54] の高速化手法も提案されている．

ハードウェアでの実装による高速化手法も多く提案されている．用いられるハードウェアとしては，ネットワークプロセッサ [38,55,56] や FPGA [39,40,41,57,58,59,60,61,20]，マルチコアプロセッサ [62,63]，グラフィックプロセッサ [64,65,66] 等がある．ハードウェアの用い方としては，まずシグネチャとのマッチングアルゴリズムをハードウェア上で実装する手法がある．Sidhu と Prasanna はFPGAを用いることで正規表現マッチングを高速化する手法を提案している [57]．Gnort [65,66] はパターンマッチングや正規表現マッチングに GPU を利用することで，性能向上を達成している．Paxson らはマルチコアプロセッサと FPGA を組み合わせるためのアーキテクチャを提案している [62,63]．Clark ら [38] は，ネットワーク・プロセッサ (NP) と FPGA を用いて，エンド・ホスト上で動作するハードウェア・ベースのネットワーク・ノード IDS を提案している．また，マッチング自体はソフトウェアで行い，それ以外の部分をハードウェアで実装するシステムもある．Shunt [61,20] は，FPGA を用いて，監視対象の大部分のトラフィックについて，ソフトウェアでの検査を回避させることにより高速化する手法を提案している．TCP Splitter [39,40] は FPGA を用いて TCP ストリームを再構成することで高速化を行う．

以上で述べたアルゴリズムの高速化，ハードウェアによる高速化は，共に 1 台の NIDS の性能向上を目的とした研究である．本研究では，ネットワーク内に配置された複数の NIDS の協調による性能向上を提案しているため，これらの手法とは補完的となる．すなわち，各手法を用いた性能が向上した NIDS を協調させることでさらに性能を向上することができる．

一方，並列に動作する複数のマシンを用いて NIDS を構成することで，性能を向上させることを目的とした研究もある．これは，本研究で提案する Brownie のよ

うに独立に動作する複数のNIDSを協調させるというよりは，複数台のマシンを用いて1つの高性能なNIDSを構成し，組織ネットワーク内の1つの監視点に置くという手法である．例えば，NIDS Cluster [18], Active Splitter [19]，Kruegelらによるもの [67] などがある．これらは，基本的にセンサと呼ばれるトラフィックの検査を行う複数のマシンと，センサにトラフィックを割り当てるフロントエンドから構成されている．トラフィック検査を行うセンサを複数置き，処理を分散させることによって，一台のマシンよりも性能を向上させることができる．Kruegelら [67] は，高速かつ高精度な検査を可能にするために，三層構造のフロントエンドを提案している．Active Splitter [19] は，性能向上を目的としたフロントエンドの最適化手法を提案し，NIDS Cluster [18] は，センサ同士のやりとりを強化したシステムを提案している．しかし，これらのシステムを導入するためのコストは少なくない．管理者は，複数のマシンを購入し，設置や設定をし，管理しなければならない．多くのマシンを設置するのに必要な場所や電源の確保が容易ではない場合もある．本研究は，既に組織内のさまざまな場所に置かれているNIDSを利用し，それらを協調させるというアプローチを提案している．

2.3 耐障害性向上を目的とした研究

検知精度や性能に比べ，NIDSの耐障害性に関する研究はあまり行われていない [68]．Kuangらは，攻撃に対する耐性を考慮したNIDSを提案している [69]．複数の攻撃検知部とその動作を定期的に監視する監視用エージェントを用いた構成となっており，監視用のエージェントが攻撃検知部やマシンの障害を検知すると，攻撃検知部の複製を別のマシンで起動させることで，攻撃検知を継続させる．Siqueiraらは，エージェント型のNIDSにおいて，監視用エージェントが各NIDSの動作状態を監視し，停止したNIDSを復元する手法を提案している [70]．しかし，これらのシステムは，各提案手法に合致したアーキテクチャのNIDSを採用していなければならない．我々は，広く使われているNIDSを対象とし，対象とするNIDSのルール設定を変更することで，組織内ネットワークの木構造型のトポロジーを利用して，NIDS同士が監視し合い障害時の代替わりをするシステムを提案している．

第3章　レイヤ7 NIDS/NIPSのための TCPストリーム再構成機構

本章では，レイヤ7コンテキストを考慮した検知を行うのに不可欠である，TCPストリーム再構成機構について述べる．まず，レイヤ7コンテキストを考慮することにより精度の高い攻撃検知が可能となることを，実際の攻撃例を挙げながら説明する．次に，レイヤ7コンテキストを考慮した攻撃検知を行うために必要となるTCPストリーム再構成機構の要件を述べる．その後，TCPストリーム再構成機構の要件を全て満たす手法として，store-through方式を提案する．さらに実験を行い，store-through方式が要件を満たすことを示す.

3.1　レイヤ7コンテキストを考慮したNIDS/NIPS

3.1.1　レイヤ7コンテキストとは

レイヤ7コンテキストを考慮した攻撃の検知は，ネットワーク越しの不正攻撃を防御・検知する方法として非常に有用である．レイヤ7コンテキストとは，1) メッセージの送受信順序，2) 各メッセージの構文（メッセージ・フィールドがどのような順で並ぶか），そして3) 各フィールドのフォーマット（フィールドがどのような文字から成るか）のことをいう．メッセージの構文はBNF (Backus Naur Format) で，フィールドのフォーマットは正規表現で定義されることが多い.

ここで，レイヤ7コンテキストの例として，簡単なHTTPを挙げる．図3.1に示すように，HTTPでは，まず最初にクライアントが欲しいwebページのURIを含めたGETリクエストをサーバに送信する．サーバは指定されたwebページを含んだ返答をクライアントに返す．この簡単なHTTPのレイヤ7コンテキストは以下のようになる．この例で，1) で使われるRequestとResponseは2) で非終端記号として定義され，2) で終端記号として使われている*URI*や*HTTP-Version*は3) で定義されている.

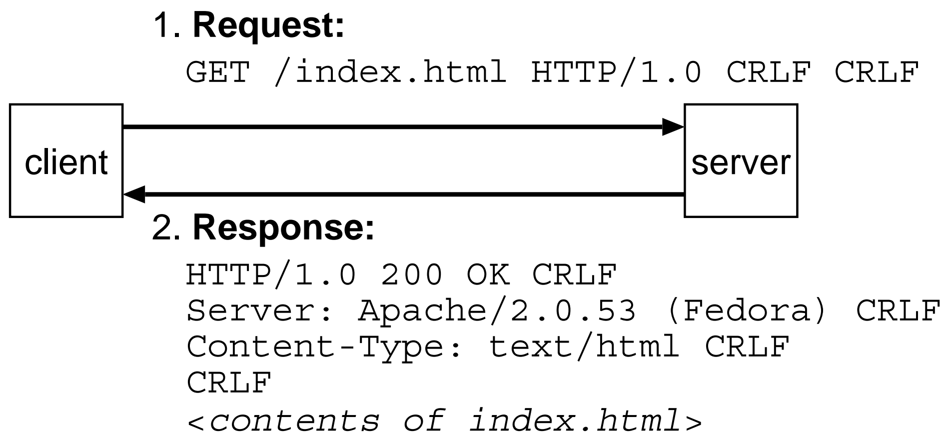


図 3.1: 簡単な HTTP の流れ

1) メッセージの送受信順序

1. クライアントが Request メッセージを送信する.
2. サーバは Response メッセージで返答をする.
3. 1 と 2 を繰り返す.

2) 各メッセージの構文

Request ::= "GET" SP *URI* SP *HTTP-Version* CRLF CRLF
/* SP は空白，CRLF は "\r\n" */

Response ::= *HTTP-Version* SP *Status-Code* SP *Reason-Phrase* CRLF
Header-List CRLF *Entity-Body*

Header-List ::= Header-List *Header*
| ϵ /* ϵ は空文字 */

3) 各フィールドのフォーマット

URI = "/" (*path*)? (";" *params*)? ("?" *query*)?
/* *path*, *params*, *query* は,
英数字や "."，"/" などの記号からなる文字列 */

HTTP-Version = "HTTP/" *digit*+ "." *digit*+ /* *digit* は数字 ("0"-"9") */

Status-Code = *digit digit digit*

Reason-Phrase = *alpha** /* *alpha* は英字 ("a"-"z" または "A"-"Z") */

Header = (*alpha* | *digit*)* ":" (*alpha* | *digit*)* CRLF

Entity-Body = .*

図 3.1 の例では，リクエストメッセージの *URI* は『/index.html』であり，

HTTP-Version は『HTTP/1.0』である．URI 内の *path* は『index.html』であり，*params* と *query* は空である．サーバからのレスポンスには，『`Server:...`』と『`Content-Type: ...`』の 2 つのヘッダと index.html の内容が含まれている．

3.1.2 レイヤ 7 コンテキストを考慮したネットワーク侵入検知

レイヤ 7 コンテキストを考慮することで，さまざまな攻撃を高い精度で検知することが可能となる．シグネチャ型の NIDS では単にメッセージとシグネチャと呼ばれるバイト列との比較を行うだけであるが，**レイヤ** *7 NIDS* ではレイヤ 7 のプロトコルの文脈に基づいてメッセージの検査を行う．そのため，従来のバイト列のシグネチャでは検知できない様々な攻撃を検知することができる．ここでは，レイヤ 7 コンテキストにより可能となる検知方法を 3 つ挙げ，それらによって検知できる攻撃の例を示す.

フィールドの長さを検査する

レイヤ 7 NIDS では到着したメッセージを各フィールドに分割することができるため，各フィールドの長さを検査することが可能となる．各フィールドの長さを検査できれば，バッファオーバーフロー攻撃を高い精度で検知することができる．バッファオーバーフロー攻撃は，バッファをあふれさせるという攻撃手法の本質的な特徴により，非常に長いフィールドを含むメッセージを送るためである．

例えば CodeRed [1] は，非常に長い URI を含むメッセージを，脆弱性を持つ Microsoft IIS web サーバに送りつけることで攻撃を行う．レイヤ 7 コンテキストを用いることにより，レイヤ 7 NIDS は送られてきたメッセージのどの部分に URI が含まれているかを知ることができる．そのため，メッセージから URI フィールドを取り出し，その長さをチェックすることができる．実際，レイヤ 7 コンテキストを利用したネットワーク侵入防御システムの一つである Shield [22] は，URI フィールドの長さを調べることで CodeRed を検知している．Shield では，HTTP クエリ（URI 内の ‘?’ 以降の文字列）が 239 文字よりも長い場合，そのメッセージを攻撃だと判断する．また，商用の NIDS である Web Intelligence [71] は，HTTP メッセージの様々なフィールドの最大長を設定することができる．もし設定した長さを超えるフィールドを含むメッセージがあれば，そのメッセージを破棄して，バッファオーバーフロー攻撃が成功する可能性を減らす.

別の攻撃例として，Apache の脆弱性 [72] に対する攻撃を挙げる．Apache では，1 つの HTTP リクエストに HOST ヘッダが 2 つ含まれていた場合，その HOST の値を 1 つに連結する．このとき，長さの検査は各 HOST ヘッダごとに行うため，2 つの HOST ヘッダを連結した時の長さがバッファの長さを超えていると，バッファオーバーフローが起こる．レイヤ 7 を考慮した NIDS では，2 つの HOST ヘッダの長さの和を検査することで，この脆弱性をつく攻撃を防御することができる．

レイヤ 7 コンテキストを用いずに，バッファオーバーフロー攻撃を検出することは困難である．CodeRed の例でいえば，レイヤ 7 コンテキストを利用しなければ HTTP リクエスト内に含まれる URI の長さを検査することはできない．HTTP リクエストには，URI 以外にヘッダやボディなどが含まれるため，HTTP リクエスト全体の長さが長いということのみで，バッファオーバーフローを狙った攻撃だと断定することは困難である．HTTP リクエスト内のヘッダの数や長さが一定以下であり，かつボディを含まないと仮定すれば，全体の長さが長い HTTP リクエストには，高い確率で長い URI が含まれると推測することもできるかもしれない．しかし，多くのヘッダや長い値を持つヘッダ，長いボディが含まれる HTTP リクエストも存在するため，レイヤ 7 コンテキストを用いて URI フィールドを特定し，その長さを検査するほうが，より正確に攻撃を検知できる．また，シグネチャ型 NIDS は，各攻撃に対して個別にシグネチャを用意しなければならないため，攻撃の亜種に弱い．亜種とは，同じ脆弱性を利用する複数の攻撃であり，バイト列とのマッチングのみで攻撃を検知するためには，それぞれの亜種ごとにシグネチャを作成する必要がある．実際，2001 年 7 月 18 日に最初の CodeRed が出た後，7 月 19 日，8 月 4 日，8 月 23 日と次々と亜種が出現したという報告がある．Apache の脆弱性の例でも，攻撃メッセージは，2 つの HOST ヘッダに攻撃コードを分割して含めることができるため，シグネチャを用いた場合の攻撃検知は困難である．

フィールドの内容を検査する

レイヤ 7 NIDS を用いることにより，各フィールドの内容を検査することが可能となる．フィールドの中身を検査することにより，レイヤ 7 NIDS では web サーバに対するコマンド・インジェクション攻撃，SQL インジェクション攻撃などを防御することができる．これらの攻撃では，攻撃者はコマンド・インタプリタやデータベースで特別な意味を持つメタ文字を含むメッセージを送る．レイヤ 7 NIDS では，本来メタ文字を含むべきでないフィールドにメタ文字が含まれていることを

検知できる．

例として，formmail CGI スクリプトが持つ，コマンド・インジェクション攻撃に対する脆弱性 [73] を挙げる．この脆弱性を持つ formmail CGI スクリプトは,『···;*shell-cmds*』（‘|’ の代わりに ‘;’ でも良い）という形式の文字列を，recipient という CGI パラメータの値として渡すことで，メッセージ内に含んだシェル・コマンドを実行してしまう．レイヤ 7 NIDS は，メッセージの中から recipient の値に対応する文字列を取り出し，メタ文字（この例では，‘;’ または ‘|’）が含まれていないか検査することで，このような攻撃を検知できる．Bro [13] では，HTTP リクエストから URI フィールドを取り出して，正規表現『.*formmail.*\?.*recipient=[^&]*[;|].*』と比較することで，この攻撃の検知を行っている．これはつまり，Bro は，‘;’ または ‘|’ が，formmail CGI スクリプトの recipient の値として含まれているとき，この攻撃を検知するということである．

また別の例として，Microsoft IIS web サーバの脆弱性 [74] がある．この脆弱性は，HTTP GET リクエストの最後に『Translate: F』というヘッダを含んだメッセージを受信すると，アクセス先のファイルのソースコードに攻撃者がアクセスできてしまう，という脆弱性である．レイヤ 7 NIDS は，メッセージからヘッダを取り出し，またその各ヘッダがどのような順で並んでいたかを知ることができる．そのため，もし受信したヘッダの中に『Translate: F』が含まれており，かつその後ろに他のヘッダが存在しない，すなわち『Translate: F』が最後のヘッダである場合には，攻撃だと判断できる．

レイヤ 7 コンテキストを考慮しない場合，このような種類の攻撃の検知は困難である．なぜなら，メタ文字がメッセージ中に存在するというだけで，コマンドインジェクション攻撃や SQL インジェクション攻撃であると断定できる訳ではないからである．formmail CGI スクリプトの例では,「‘;’ または ‘|’ が formmail CGI スクリプトの recipient の値として含まれる」ときのみ攻撃となる．このようなシグネチャを従来のシグネチャ型 NIDS で記述するのは不可能である．また，Microsoft IIS web サーバの脆弱性でも,「『Translate: F』を含む」というシグネチャは記述できても，これだけでは攻撃だと判断できない．この脆弱性では,「『Translate: F』が HTTP GET リクエストのヘッダの最後」に含まれていなければならず，このようなシグネチャをレイヤ 7 コンテキストを用いないで記述するのは困難である．

メッセージの順序を利用する

レイヤ 7 NIDS を用いることにより，メッセージ順序を考慮することで，より精度の高い攻撃の検知が行える．攻撃の中には，いくつかの手順を踏んで攻撃を行う攻撃もあり，そのような手順を追うことでより正確な検知ができる．

例えば，Slapper [75] ワームは二段階の手順を踏んで，Apache/mod_ssl にある脆弱性を攻撃する．まず第一段階として，Slapper は簡単な HTTP リクエストを送り，脆弱性が存在するかどうかのプローブを行う．その後第二段階として，OpenSSL の脆弱性を狙って攻撃を仕掛ける．レイヤ 7 NIDS ではこの手順を追うことで攻撃を検知する．まず，NIDS は Slapper からのプローブを検知した時点で，プローブがあったということを記憶しておく．その後，同じ送信元から SSL コネクションが確立すると，Slapper による攻撃だと判断する．Bro ではこのようにして Slapper を検知し，警告を出している．

Slapper を検知するためには，この，プローブとそれに続く SSL コネクションの確立という，二段階の手順を追うことが不可欠である．攻撃自体は SSL コネクション内に含まれており，全ての内容が暗号化されているため，メッセージの中身とバイト列とのマッチングによって検知するとは不可能である．また，プローブを検知しただけで攻撃だと判断すると，多くの誤検知が発生してしまう．プローブの結果，攻撃対象でないと判断された場合，実際の攻撃は行われないからである．

また，別の例として，Qualcomm qpopper POP3 サーバの `LIST` コマンドによるバッファオーバーフローの脆弱性を挙げる．この脆弱性では，POP アカウントのユーザ名とパスワードでログインしたユーザが長い引数を伴った `LIST` コマンドを送信してくると，バッファがオーバーフローし，不正なコマンド等が実行される．通常，`LIST` コマンドは POP3 におけるユーザ認証終了後にのみ受け付けられる．逆に言えば，ユーザ認証が行われなかったり失敗した場合には，バッファオーバーフローを狙って長い引数を伴った `LIST` コマンドが送信されてきても攻撃は成功しない．レイヤ 7 NIDS では，やりとりされるメッセージを追跡することができるため，ユーザ名やパスワードが送信されユーザ認証が正常に終了した後に，この脆弱性をつく `LIST` コマンドが送信されてきた場合にのみ攻撃を検知する，ということができる．通常のシグネチャ型の NIDS では，このようなメッセージの追跡が行えないため，`LIST` コマンドの検知のみとなり，攻撃とならないメッセージも検知する，という誤検知が発生する可能性がある．

3.2 レイヤ 7 NIDS のための TCP ストリーム再構成機構の要件

第 3.1 節で示したとおり，レイヤ 7 コンテキストを考慮することで，NIDS はより精度の高い攻撃の検知ができるようになる．しかしながら，レイヤ 7 コンテキストを考慮した NIDS/NIPS（レイヤ 7 NIDS/NIPS）を効率的に実装するのは容易ではない．レイヤ 7 コンテキストを考慮してメッセージの検査を行うためには，個々のパケットを完全なメッセージ・ストリームとして再構成しなければならない．すなわち，順番が入れ替わって届いたパケットは正しい順番に並べ直し，IP フラグメントされたパケットは再構成しなければならない．そのため，レイヤ 7 NIDS は *TCP* **ストリーム再構成機構**と**センサ**の 2 つから成る．TCP ストリーム再構成機構は届いたパケットを再構成してメッセージ・ストリームにし，センサに渡す．センサでは，そのメッセージが攻撃かどうかをレイヤ 7 コンテキストを用いて検査する．本研究では，TCP ストリーム再構成機構に注目し，NIDS だけでなく NIPS でも用いることができることを目指す.

レイヤ 7 コンテキストを考慮した NIDS のための TCP ストリーム再構成機構が満たさなければならない要件は以下の 4 つである．

- **完全な防御**：ネットワーク侵入防御システム（NIPS）は悪意のある攻撃メッセージを遅延なくブロックできなければならない．すなわち，NIPS があるメッセージを攻撃だと判断したら，そのメッセージが攻撃対象となっているアプリケーションに渡ることがあってはならない．この要件は，攻撃を防御するという観点から非常に重要である．攻撃メッセージが攻撃対象アプリケーションに渡った後に NIPS が行動を起こしても，そのアプリケーションは既に攻撃されている可能性があるからである．

- **性能**：NIDS による性能のオーバヘッドはできる限り少なくするべきである．NIDS を使用することによって性能が大きく低下する場合，管理者は NIDS の使用を躊躇したり使用をやめる可能性もある．理想的には，TCP ストリーム再構成機構の性能は，単に IP 層で転送を行う場合の性能にできるだけ近づけることができるとよい.

- **アプリケーション透過性**：NIDS はアプリケーション透過性を損なうべきではない．アプリケーション透過性とは，NIDS を新たに設置するときに，サーバ

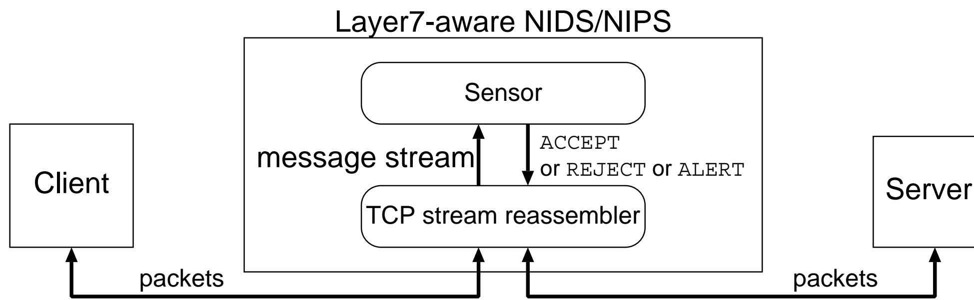


図 3.2: システム・アーキテクチャ

やクライアント・アプリケーションの変更や再設定が必要ないことをいう．アプリケーションの変更や再設定が必要であると，管理者は NIDS の導入を躊躇する可能性もあり，アプリケーション透過性を保持することは重要である．

- **トランスポート透過性**：NIDS は end-to-end の TCP/IP の振る舞いに与える影響を少なくするべきである．トランスポート透過性とは，NIDS が TCP フローや輻輳制御に与える影響が無視できるほどに小さいことをいう．単純な実装方法では，end-to-end の TCP/IP の振る舞いに影響を与えてしまい，エンドホストの TCP/IP プロトコル・スタックが通常と異なる振る舞いをするおそれがある．

3.3 Store-through 方式

本研究では，レイヤ 7 NIDS のための TCP ストリーム再構成機構として，*store-through* 方式を提案する．Store-through は，1) 完全な防御，2) 性能，3) アプリケーション透過性，4) トランスポート透過性の上記 4 つの要件を全て満たす方式である．この節の以降では，まず最初に第 3.3.1 節でレイヤ 7 NIDS 全体のアーキテクチャを示す．次に第 3.3.2 節で store-through の詳細な動作を説明する．第 3.3.3 節と第 3.3.4 節では，store-through の動作に必要なパケットとコネクションの管理について述べる．最後に，第 3.3.5 節で IP フラグメンテーションの扱いについて述べる．

3.3.1　システム・アーキテクチャ

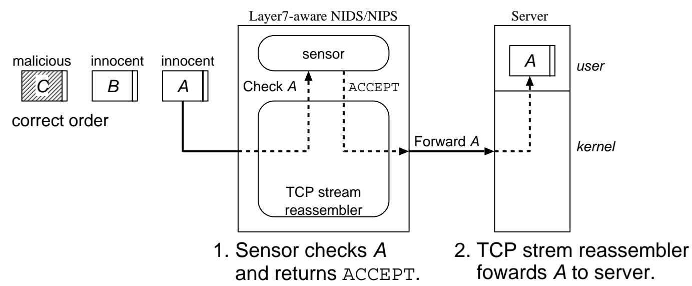
Store-through の詳細な動作の説明の前に，レイヤ 7 NIDS のアーキテクチャを簡単に説明する．ここでは，NIDS はカーネル内で動くことを仮定している．図 3.2 に示すように，レイヤ 7 NIDS は *TCP* **ストリーム再構成機構**（*TCP stream reassembler*）と**センサ**の 2 つから成る．TCP ストリーム再構成機構は，受信した個々のパケットをメッセージ・ストリームに変換する．つまり，順番が入れ替わったパケットを並べ直したり，IP フラグメントされたパケットを再構成したりする．その後，TCP ストリーム再構成機構はメッセージ・ストリームをセンサに渡す．センサは，渡されたメッセージ・ストリームを検査して，攻撃でないかどうかを判断し，攻撃でなければ `ACCEPT`, 攻撃であれば `REJECT` または `ALERT` を返す．もし，攻撃でないと判断されたら（`ACCEPT` が返されたら），TCP ストリーム再構成機構はそのパケットを最終的な宛先まで転送する．逆にもし攻撃だと判断されたら（`REJECT` または `ALERT` が返されたら），NIDS は警告を発し，NIPS の場合は転送せずにそのメッセージは破棄する．

3.3.2　Store-through の動作

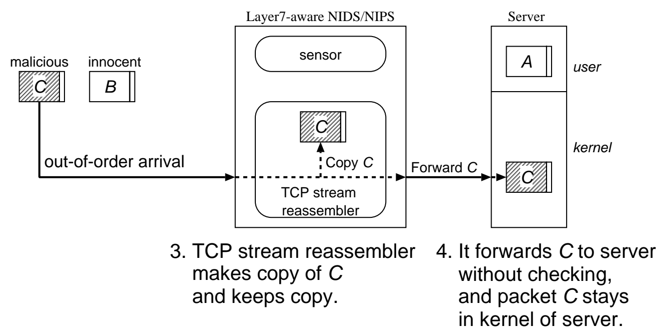
図 3.3 に store-through 方式の動作の流れを示す．パケットの正しい順序は，A, B，C の順である．このメッセージは攻撃メッセージであるが，パケット A と B は通常の内容と同じであるため，攻撃だと判断できない．しかし，パケット C には攻撃メッセージが含まれており，パケット C を検査した時点で攻撃であると判断できる，と仮定する．

まず，パケットが順番通りに到着した場合は，TCP ストリーム再構成機構は順番通りのパケットだということを確認した後，特に処理は行わずに TCP/IP ヘッダを除いたメッセージ部分をセンサに渡す．そして，センサでメッセージの検査をし，攻撃でないと判断された後に転送する．図 3.3(a) ではこの場合を示している．パケット A はメッセージの最初のパケットであり，それが一番最初に届いたため，順番通りに届いたということになる．このとき，TCP ストリーム再構成機構はパケット A をセンサに渡し，センサはパケット A を検査する．パケット A は攻撃ではないため，センサは `ACCEPT` を返し，TCP ストリーム再構成機構はパケット A をサーバに転送する．

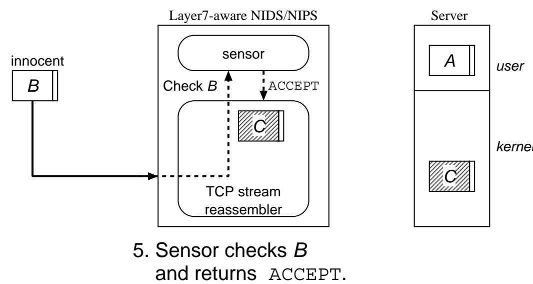
一方，パケットの順番が入れ替わっていた場合は，入れ替わったパケットを並



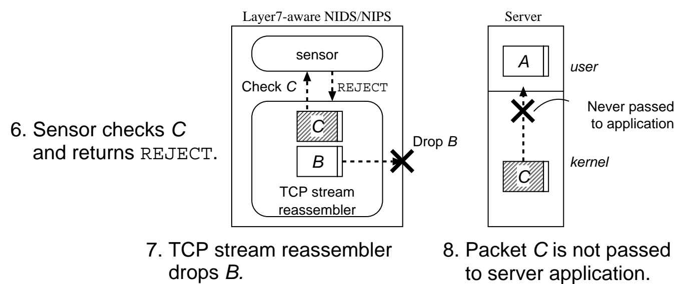
(a) 順番通りに届いたとき



(b) 順番が入れ替わって届いたとき



(c) 抜けていたパケットが届いたとき



(d) 転送したパケットが攻撃だと分かったとき

図 3.3: Store-through 方式の動作

び直さなければならない．そのため，順番が入れ替わったパケットが到着すると，TCP ストリーム再構成機構は，それよりも前に来るべき全てのパケットを待たなければならない．センサに渡して検査をするためには，個々のパケットを順番通りに並べ替えてメッセージ・ストリームに変換する必要があるためである．第 2.1.1 節で示したとおり stop-forward 方式では，順番が入れ替わったパケットは，前のパケットが全て届くまで転送されずに，NIDS で止められる．この場合，サーバにはパケットが 1 つも届かない状態となるため，サーバ側の TCP の輻輳制御機構がネットワーク負荷が大きくなっていると誤って判断する可能性がある．すなわち，トランスポート透過性は保持されない．しかし，store-through 方式では順番が入れ替わったパケットは，複製をつくりすぐに最終的な宛先に転送する．store-through 方式では，順番が入れ替わったパケットも転送を止めないため，stop-forward 方式に比べてパケットの振る舞いに与える影響が少ない．すなわち，トランスポート透過性が保持される．順番が入れ替わったパケットは，コピーをとった後，すぐに最終的な宛先に転送することで，stop-forward 方式のようにパケットが 1 つも届かないためにネットワークの負荷が大きくなっていると誤って判断する可能性を抑えることができる．パケットの順番が入れ替わっていたときの store-through の動作を図 3.3(b) に示す．図 3.3(b) では，パケット C が前にくるパケット B よりも早くに到着している．そのため，TCP ストリーム再構成機構はパケット C のコピーを作成した後，パケット B の到着を待つことなくサーバに転送する．

このとき，順番が入れ替わったパケットはセンサで検査されることなく転送され，実際の検査は前のパケットが全て届いた後となる．Store-through 方式では，後から検査するために順番が入れ替わったパケットは，コピーをとってから転送している．全てのパケットが届いたら，TCP ストリーム再構成機構はコピーしておいたパケットと届いたパケットを並べ直して，メッセージ・ストリームに変換し，センサに渡して検査する．センサが攻撃でないと判断した場合，届いたパケットを通常通りサーバに転送する．保存しておいたパケットは既に転送済みのため，破棄する．一方，攻撃であると判断した場合，最後に届いたパケットを転送せずに破棄し，サーバ・アプリケーションに攻撃メッセージが渡されるのを防ぐ[1]．図 3.3(c) と (d) に，この時の動作を示す．図 3.3(c) が示すように，パケット B が到着すると，パケット B をセンサに渡して検査する．その後，図 3.3(d) に示すように，コピーしてあったパケット C を検査する．ここでは，パケット C が攻撃だと判断された

[1]ただし，これは NIPS の場合．NIDS の場合には警告を発するのみで，破棄はしない．

ため，パケット B（パケット C ではなく）を転送せずに破棄する．実際に攻撃メッセージを含むパケット C は既にサーバ側に転送してしまっているため，パケット C を破棄することはできない．そこで，代わりにまだ転送していないパケット B を破棄する．また，このとき同時にコネクション情報を破棄し，サーバ側に RST パケットを送る．これにより，パケット B や C が再送されてきても，対応するコネクション情報が存在しないため，そのパケットは破棄される．

これは一見すると，store-through 方式では攻撃パケットがサーバ側に転送されることがあるため，完全な防御が達成できないように思えるかもしれない．しかしながら，実際には攻撃が成功することはない．攻撃を含んだパケットが，攻撃対象のアプリケーションに渡されることはないためである．攻撃パケットは，最終的な宛先のマシンまでには渡されるが，前に来るべきパケットは NIDS で破棄されて転送されないため，攻撃メッセージが宛先のアプリケーションに渡されることはない．図 3.3(d) で示すように，サーバの OS 内にあるプロトコル・スタックは，パケット C を保持したまま，抜けているパケット B が到着するのを待つ．しかし，パケット B は TCP ストリーム再構成機構によって破棄されているため，永遠にパケット B がサーバに届くことはない．そのため，サーバ・アプリケーションがパケット C を受け取ることはない．プロトコル・スタックはパケット B を渡さずにパケット C を渡す，ということはしないからである．このようにして，store-through 方式では先に攻撃を含んだパケットを転送してしまっていても，攻撃は成功せず，完全な防御が達成できている．もしクライアントがパケット B や C を再送しても，TCP ストリーム再構成機構はそれを破棄するため，攻撃は失敗する．

Store-through 方式では，トランスポート透過性と完全な防御は上記で述べたとおり達成できている．それに加えて，性能とアプリケーション透過性についても達成できている．性能については，stop-forward 方式と同様に，store-through 方式でもプロキシ方式よりも良いと考えられる．Store-through 方式は stop-forward 方式と同様にカーネル内で直接パケットを処理するため，プロキシ方式で必要となる 2 回の TCP/IP プロトコル・スタックの通過や 2 回のユーザ・カーネル空間間のデータ・コピーが必要ないからである．さらに store-through 方式では，stop-forward 方式のようにパケットを止めることがないため，stop-forward 方式よりも良い性能が期待できる．なぜなら，stop-forward 方式では，NIDS で全てのパケットを止めているため，宛先のプロトコル・スタックでは順番通りのパケットが届いた後に，順番が入れ替わったパケットも含めて一気に処理をしなければならない．それに対して，store-through 方式では，宛先のプロトコル・スタックは，順番通りのパケッ

トの到着を待つまでの間に，先に到着したパケットの処理を行うことができるからである．最後に，store-through 方式では，stop-forward 方式と同様にクライアントとサーバ間の TCP コネクションを分断しないため，アプリケーション透過性は保たれる．

3.3.3 パケット管理

保存サイズの制限

前節で説明したように，store-through 方式では，順番が入れ替わったパケットのコピーを保存する．多数のパケットのコピーを保存するのを防ぐため，やりとりされるパケットに含まれる TCP ヘッダのウィンドウ・サイズ・フィールドの値を元に，保存するパケットの合計サイズに最大値を設定する．もし何らかの最大値が設定されていなければ，攻撃者は送信するパケットを巧妙に操作することでメモリを枯渇させる攻撃が可能であるためである．ウィンドウ・サイズとは，受信側で用意されている受信用バッファの大きさなどによって変化し，送信側はウィンドウ・サイズ内であれば受信側からの ACK を待たずに，続けてパケットを送信できる．通常の送信者がウィンドウ・サイズを超えた量のパケットを送信することはないため，ウィンドウ・サイズを超えたパケットは破棄することができる．このようにして，通常の通信に悪影響を与えることなく，パケットのコピーの合計サイズに最大値を設けている．TCP ヘッダのウィンドウ・サイズ・フィールドの大きさは 16 ビットであるため，ウィンドウ・サイズの最大値は 65535 バイトである．TCP ヘッダオプションの window scale option を用いることで，最大 1GB まで設定可能であるが，受信側バッファの制約によりそこまで大きい値が設定されることは少ない．

パケットの順番整理

TCP ではパケットの順序が分かるように，TCP ヘッダにシーケンス番号を含んでいる．そのため，TCP ストリーム再構成機構でもシーケンス番号を利用することで，パケットの順番を知ることができる．TCP では，最初にコネクションを確立する際に送る SYN パケットに，シーケンス番号の初期値を含める．SYN パケットはシーケンス番号を 1 使うので，次のパケットのシーケンス番号は，初期値+1 となる．以後，到着したパケットのシーケンス番号とデータ長の和がその次に来

るべきパケットのシーケンス番号と予測でき，パケットの順序が分かるような仕組みを実現することができる．具体的には，以下のように次のシーケンス番号を予測できる．

- コネクション確立のためのパケット (SYN フラグがセットされたパケット) なら，次のパケットのシーケンス番号は,

$$\text{次のシーケンス番号} = \text{シーケンス番号} + 1. \tag{3.1}$$

- コネクション確立のためのパケットでなければ，次のパケットのシーケンス番号は,

$$\text{次のシーケンス番号} = \text{シーケンス番号} + \text{データ長}. \tag{3.2}$$

- データ長は，IP ヘッダに含まれる全データ長と IP ヘッダ長，TCP ヘッダに含まれる TCP ヘッダ長から求める．全データ長はバイト単位，IP，TCP のヘッダ長は 4 バイト単位であることに注意すると,

$$\text{データ長} = \text{全データ長} - (IP\,\text{のヘッダ長} + TCP\,\text{のヘッダ長}) * 4. \tag{3.3}$$

順番が入れ替わったパケットが到着した場合，パケットのコピーを TCP コネクションごとにリストとして保持しておく．このとき，パケットの順番通りになるようにしておく．シーケンス番号は，あるホストから送られてくるデータに関するものであるため，ひとつのコネクションに対するシーケンス番号の予測値，及びパケットのコピーを保持するリストは，それぞれ 2 つ用意する．1 つはクライアント，1 つはサーバのためのものである．

3.3.4 コネクション管理

TCP ストリーム再構成機構では TCP のコネクション状態を管理している．パケットのコピーや順番管理のための情報など，コネクションごとに必要な情報をここではまとめてコネクション情報という．まずコネクション情報の作成は，最初に SYN パケットを受信したときに行い，対応するコネクションの状態を保持するための構造体を作成する．もし SYN パケットにペイロードが含まれていたら，セン

サに渡して検査をする．以降，到着するパケットはこのコネクション情報に対応づけて処理される．対応するコネクション情報がなく，SYN フラグがセットされていないパケットが到着した場合，そのパケットは破棄する．

コネクション情報の破棄は，1) FIN，2) RST，3) タイムアウト，4) 攻撃，の 4 つのうちのいずれか起こったときに行われる．第一に，サーバ・クライアント双方から FIN を受け取った場合には，コネクションが通常の手順を踏んで切断されたとして，コネクション情報を破棄する．FIN パケットがペイロードを含んでいた場合は，コネクション情報を破棄する前にセンサに渡して検査をする．最後の FIN に対する ACK パケットは，対応するコネクション情報がすでに破棄されているために，対応するコネクション情報を見つけることができない．しかし，この ACK パケットはペイロードを含んでいないため，攻撃の可能性はなく，転送してよい．第二に，RST を受け取った時にはすぐにコネクション情報を破棄する．第三に，FIN や RST のやりとりがなく，何か障害によってコネクションが切断する状況があり得る．このとき，TCP ストリーム再構成機構ではパケットの到着を一定時間待ち，それが過ぎるとコネクション情報を破棄する．最後に，センサが攻撃を検知したときに，パケットを破棄すると同時にコネクション情報を破棄する．それ以後，再送されたそのコネクションのパケットは，対応するコネクション情報が存在しないため，自動的に破棄される．

コネクション情報は，1) 送信元 IP アドレス，2) 宛先 IP アドレス，3) 送信元ポート番号，4) 宛先ポート番号，5) プロトコル番号 の 5 つの要素で識別する．コネクション情報には，順番が入れ替わったパケットのコピーや次に来るべきパケットのシーケンス番号，またセンサでメッセージを検査するために必要な情報などが含まれる．

3.3.5 IP フラグメンテーションの扱い

順番が入れ替わったパケットと同様にして，IP フラグメントされたパケットも，転送を止めることはしない．IP フラグメントされたパケットが到着した場合，その時点で全てのフラグメントされたパケットが集まったならば，TCP ストリーム再構成機構はそれを再構成してセンサに渡す．一方，もしまだ抜けているパケットがあった場合，TCP ストリーム再構成機構はそのパケットをコピーした後，すぐに最終的な宛先に転送する．ここでも先ほどと同様に，メモリ枯渇を防ぐために，コピーするパケットの量の最大値を設定している．

表 3.1: MF フラグ，フラグメント・オフセット (FO) とフラグメントの関係

MF フラグ	FO	フラグメント	元の位置
0	0	No	–
	not 0	Yes	最後
1	0	Yes	最初
	not 0	Yes	途中

フラグメントされたパケットについて

IP フラグメントされたパケットの順番整理も，TCP のシーケンス番号と同様に，IP ヘッダに付加された情報を用いて行うことができる．IP ヘッダには IP フラグメントされたパケットが受信側で再構築できるように，識別子 (identification)，MF (more fragment) フラグ，フラグメント・オフセットの 3 つの情報が付加されている．フラグメントされたデータ全てに同じ識別子がつき，これによって元のデータを識別することができる．MF フラグは，それが最後のフラグメントではないことを示す．そして，フラグメント・オフセットは，そのフラグメントがもとのデータのどの位置にあったのかを示す．

パケットがフラグメントされたものかどうかは，MF フラグとフラグメント・オフセットの値を見れば分かる．その関係を表 3.1 に示す．もし，パケットがフラグメントされていなければ，前節で述べたコネクション管理とパケットの順序管理を行う．フラグメントされていた場合の管理について以下で述べる．

パケットの順序および再構築

IP フラグメントされたパケットの順番および再構築は，識別子，MF フラグ，フラグメント・オフセットの 3 つを利用して行う．MF フラグが立っていればまだフラグメントされたパケットがあるということを意味する．識別子は，フラグメントされたデータ全てに同じ値がつくので，これによって元のデータを識別できる．また，フラグメント・オフセットは，もとのデータのどの位置にあったかを示す．これより，次のように今届いたパケットから次のパケットの値を予測することで，パケットの順番を知ることができる．具体的には，以下のように次の識別子とフラグメント・オフセットを予測できる．ここで，フラグメント・オフセットは 8 バイト単位，データ長はバイト単位であることに注意する．

- フラグメントされた中の最初のパケットであれば，次のパケットの識別子とフラグメント・オフセット (FO) は,

$$\text{次の識別子} = \text{識別子}, \tag{3.4}$$

$$\text{次の} FO = \text{データ長}/8. \tag{3.5}$$

- 途中のパケットであれば,

$$\text{次の識別子} = \text{識別子}, \tag{3.6}$$

$$\text{次の} FO = FO + \text{データ長}/8. \tag{3.7}$$

- 最後のパケットであれば,

$$\text{次の識別子} \geq \text{識別子} + 1, \tag{3.8}$$

$$\text{次の} FO = 0. \tag{3.9}$$

- ここで，データ長は IP ヘッダに含まれる全データ長と IP ヘッダ長から求める．全データ長はバイト単位，ヘッダ長は 4 バイト単位であることに注意すると,

$$\text{データ長} = \text{全データ長} - \text{ヘッダ長} * 4 \tag{3.10}$$

コネクション管理

IP フラグメントされたパケットのうち, 2 番目以降のパケットには TCP ヘッダが含まれていないため，ポート番号を知ることができない．すなわち，第 3.3.4 節で述べたコネクション情報の識別ができない．そのため，フラグメントされたパケットが全てそろってから，それらを１つのパケットに構成しなおした後，対応するコネクション情報を求めて処理する．IP フラグメントされたパケットが全てそろうまでは，双方の IP アドレスとプロトコル番号のみの 3 つを用いて識別し，IP フラグメントされたパケットの管理等を行う．

3.4 実装

3.4.1 TCP ストリーム再構成機構の呼び出し

Store-through 方式のプロトタイプを Linux カーネル 2.4.30 に実装した．Linux の TCP/IP プロトコル・スタック内の IP 層内に実装してある．TCP ストリーム再構成機構は，IP 層の受信関数である，ip_rcv() から呼び出されるようになっている．呼び出し関数の定義は次のようになっている．

```
int tcp_stream_filter(struct sk_buff *skb);
```

ここで，引数 skb は sk_buff 構造体に入れられたパケットである．sk_buff 構造体は，通信パケットを管理するために Linux カーネルで定義されている構造体である．sk_buff 構造体は，データ領域の始点・終点，プロトコルデータの始点・終点 (どのプロトコル層にいるかに依存する)，各プロトコル用のヘッダを指すポインタなどを保有している．ip_rcv() が呼び出されたときにネットワークデバイスから渡されたデータを，そのまま TCP ストリーム再構成機構に渡す．TCP ストリーム再構成機構では，順番通りのパケットであればセンサに渡して中身の検査を行い，返り値として PASS または BLOCK を返す．順番通りでないパケットの場合は，パケットの複製をとった後，PASS を返す．

返り値が PASS であった場合，通常の ip_rcv() の処理として，ip_rcv() → ip_forward() → ip_send() と通ってパケットを転送する．返値が BLOCK であった場合は，転送せずにパケットを破棄する．

3.4.2 コネクション管理

コネクション管理では，ひとつのコネクションに対してひとつの管理情報を作成し，そこでそのコネクションに関する全ての情報を保持する．コネクション毎の情報は，connection 構造体で記憶する．コネクションが確立したとき connection 構造体を作成し，パケットが到着するごとに値を更新していき，コネクションを切断するときに破棄する．connection 構造体は次のようになっている．

```
struct connection {
        /* コネクション識別情報 */
```

```
        struct con_info         *connect_info;

        /* 次のパケットの予測値 */
        struct expect_value     s_exp_val;
        struct expect_value     c_exp_val;

        /* パケットのコピー */
        struct stored_data      *s_data;
        struct stored_data      *c_data;

        /* どちらから FIN パケットを受け取ったか */
        int                     fin_state;

        /* センサの状態 */
        sensor_info_t            *sensor_info;
};
```

コネクション管理情報の破棄は，サーバ・クライアント両方から FIN パケットを受け取ったときに行うので，FIN パケットを受け取ったということを記憶しておく必要がある．また，センサの状態をコネクションごとに保持しておく必要がある場合，sensor_info に保持する．

コネクション識別情報 connect_info は，どのコネクションかを識別できる情報を持つ．具体的には，第 3.3.4 節で述べたように，サーバ・クライアント双方の IP アドレスとポート番号，およびプロトコル番号の 5 つである．con_info 構造体は次のようになっている．

```
struct con_info {
        __u32 saddr;
        __u32 daddr;
        __u16 sport;
        __u16 dport;
        __u8  proto;
};
```

connection 構造体の要素のうち，パケットの順序に関するものは expect_value 構造体と stroed_data 構造体である．これらの構造体の定義は以下の通りである．

```
struct expect_value {
        __u32 seq_num;
        __u16 id;
        __u16 offset;
};

struct stored_data {
        struct expect_value my_val;
        struct expect_value next_val;
        unsigned char       *data;
        struct stored_data  *next;
};
```

expect_value 構造体は，シーケンス番号，識別子，フラグメント・オフセットの 3 つの要素からなる．s_exp_value, c_exp_value には，それぞれサーバ，クライアントからの次のパケットの予測値を保持する．この予測値どおりのパケットが到着したら，順番どおりのパケットが来たことを意味する．この場合，そのパケットをセンサに渡して検査し，ACCEPT が返されたら次の予測値を第3.3.3 節，第 3.3.5 節で示した式を用いて求め，値を更新する．

stored_data 構造体は，順番どおりでないパケットをコピーして記憶しておくためのリスト構造である．サーバ，クライアントから来た順番通りでないパケットは，それぞれ s_data と c_data を先頭とするリストに記憶しておく．

IP フラグメントの考慮

IP フラグメントを考慮したコネクション管理，パケットの順序管理についても前節に類似した情報を保持する．IP レベルではコネクションという概念はないが，ここでは便宜的にホストのペア毎にコネクション情報を作成し，IP フラグメントされたパケットの管理を行う．IP レベルでのコネクション管理のための構造体は，以下の通りである．

```
struct ip_connection {
        /* IP レベル コネクション識別情報 */
        struct ip_con_info        *connect_info;

        /* 次のパケットの予測値 */
```

```
        struct ip_expect_value    s_exp_val;
        struct ip_expect_value    c_exp_val;

        /* パケットのコピー */
        struct stored_data        *s_data;
        struct stored_data        *c_data;
};

struct ip_con_info {
        __u32 saddr;
        __u32 daddr;
        __u8  proto;
};

struct ip_expect_value {
        __u16 id;
        __u16 offset;
};
```

3.5 議論

この章では，store-through を導入したことで新たな脆弱性が発生していないということを議論する．最初に我々の store-through の実装を分析する．その後，store-through に対して想定される，DoS 攻撃（Dinal of Service attack）と Evasion 攻撃の 2 つの攻撃について議論する．

3.5.1 実装の考察

Store-through の実装自体がセキュリティ・ホールを持ち込まないように，設計および実装は注意深く行っている．TCP ストリーム再構成機構はシンプルな構造となっており，実際コード行数も 1000 行以下と少なくなっている．そのため，モデル・チェッキングやソースコード検証ツールを用いてソースコードを分析し，検証できるであると考えられる．現在の実装についてはまだ検証を行っていないが，今後，このような形式的な検証ツールを使って検証を行いたいと考えている．

また，TCP ストリーム再構成機構は C 言語で実装されているが，セキュリティ・ホールとなりやすい文字列処理は行っていない．TCP ストリーム再構成機構はパケット・ヘッダのみに基づいて処理を行っている．TCP/IP のヘッダ・フィールドは，固定長で各値の配置が決まっているため，バッファオーバーフローに対する脆弱性を持つ可能性は少ない．センサはペイロードを検査するために，文字列処理をやらなければならない可能性はあるが，センサの安全性については本研究の範囲外である．

3.5.2 Denial-of-Service Attack

Store-through 方式は，DoS 攻撃につながるような，新たな脆弱性を導入してはいない．想定される DoS 攻撃としては，メモリを枯渇させる攻撃と，コネクション状態を多く作成させる攻撃である．どちらの攻撃も，実際に行うには困難である．Store-through による NIDS は，他の NIDS に比べて，DoS 攻撃に対してより脆弱であるということはない．

Store-through 方式では，順番が入れ替わったパケットはすべてコピーをとって保存しておくため，一見するとメモリ枯渇を狙った攻撃に脆弱であるように思える．攻撃者が，非常に長いパケット列を，最初のパケットを除いて送りつけた場合，store-through では送られてきた多数のパケットを全て保存しておかなければならず，メモリが枯渇する可能性があるように思えるかもしれない．しかし，第 3.3.3 節で説明したように，保存するパケット・コピーの合計バイト数には上限を設けてある．そのため，この上限を超えた量のパケットを送られてきた場合には，そのパケットの保存はせずに破棄し，サーバにも転送しない．このようにすることで，無限量のパケット・コピーを保存するということはなく，メモリ枯渇を防ぐことができている．

別の DoS 攻撃として，大量のコネクションを確率する攻撃が考えられる．この攻撃では，大量にコネクションを確立することで，TCP ストリーム再構成機構が多くのコネクション情報を持たなければならないようにし向ける攻撃である．この攻撃は，コネクションを管理しなければならない他の NIDS に共通するものであり，store-through 方式が他の方式より脆弱であるということはない．コネクション状態を管理するために，他の NIDS よりも多くの処理や情報が必要となるわけではないため，store-through 方式は，他の方式と同等のスケーラビリティがあると考えられる．少なくとも，store-through 方式は，プロキシ方式よりはスケーラブル

である．プロキシ方式では，1つのクライアントとサーバの組に対して，2つのコネクションを確率しなければならない．そのため，1つのクライアントとサーバの組に対して，1つのコネクション情報を保持すればよいstore-through方式の倍の情報を，プロキシ方式は保持しなければならない．

3.5.3 Evasion攻撃

Store-through方式は，evasion攻撃[76]に対して，既存の防御対策をそのまま取り入れることで，対処することができる．Evasion攻撃は，故意に異常な状態のパケットを送信し，TCP/IPヘッダの解釈のあいまいさを利用して，監視を逃れる攻撃である．例えば，TCPヘッダのシーケンス番号を操作して，重なり合った部分に違うペイロードを含ませることで，攻撃であるにもかかわらず，NIDSに対して攻撃でないと誤って判断させる．Evasion攻撃は，store-through方式特有のものではなく，ネットワークを流れるパケットを監視するという方法をとるNIDS全てに共通している問題である．Store-through方式でも，evasion攻撃に対して何も対策をとっていない状態では，重なり合った部分のペイロードはチェックせずに転送する．このとき，重なり合った部分が攻撃であると，その攻撃は成功する可能性がある．

しかしながら，evasion攻撃に対する対策も提案されており，store-through方式もそれと組み合わせることによって，evasion攻撃に対応することができる．例えば，protocol scrubber [77]，Traffic Normalization [78]，Active Mapping [79] といった方法が提案されている．store-through方式でも，これらの方法と組み合わせることで，evasion攻撃に対応することができる．Protocol scrubberとTraffic Normalizationは同じアプローチをとっている．これらは，受信先からACKが返ってきてないパケットは保存しておき，新たに受信したパケットのヘッダが曖昧さを引きおこす可能性がある場合，そのパケット・ヘッダを書き換えてから転送する．また，TCPのシーケンス番号やIPフラグメントのフラグメント番号が重なりあっているパケットを受信した場合，保存してあるパケットのペイロードと再送されてきたパケットのペイロードを照合する．そして，異なるデータを含む場合，再送されてきたパケットのペイロードを，保存してあるパケットのペイロードと同じになるように書き換えてから転送する．Store-through方式では，これらprotocol scrubberやTraffic Normalizationと組み合わせることで，evasion攻撃対策をすることができる．パケット・ヘッダの曖昧さをなくすために，受信したパケットはまず最初に

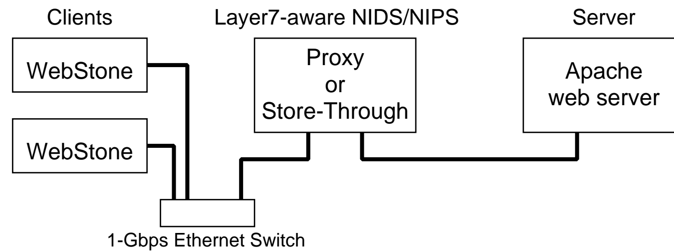


図 3.4: 実験のネットワーク設定

protocol scrubber や Traffic Normalization に渡してから TCP ストリーム再構成機構で処理を行えばよい．TCP ストリーム再構成機構でも，パケットのコピーを保存したり，コネクション状態を管理しているため，これらの処理を TCP ストリーム再構成機構の中に組み込むことも可能であると考えられる．

3.6 実験

Store-through 方式の性能を評価するために，Apache web サーバのスループットと応答時間を測定した．また，store-through 方式によって消費される計算機資源の量を知るために，CPU 使用率とメモリの使用量を測定した．

3.6.1 実験環境

図 3.4 に実験でのネットワーク設定を示す．この図に示すように実験では，サーバマシン 1 台，クライアントマシン 2 台，サーバとクライアントの間に置かれた NIDS マシン 1 台の，計 4 台のマシンを利用した．2 台のクライアントマシンと NIDS マシンは 1Gbps のイーサネット・スイッチを介して接続されており，サーバマシンと NIDS マシンは直接 1Gbps で接続されている．マシンは全て同じ構成であり，表 3.2 にマシンの性能をまとめた．OS は全て Linux 2.4.30 を用いており，カーネルのパラメータは変更していない．

web サーバとしては Apache 2.0.54 [80] を用い，ベンチマークは WebStone 2.5 [81] を用いた．WebStone は web サーバを対象とした標準的なベンチマークである．Apache，WebStone 双方とも設定はデフォルト値を利用した．

表 3.2: 実験環境

CPU	Pentium4 2.40 GHz
メモリ	512 MB
HDD	40 GB, 7200 rpm, UltraATA100
OS	Linux 2.4.30
ネットワーク	1000 Base-T

3.6.2 ベンチマーク

Store-through 方式による性能への影響を知るために，次の 4 つの場合について，web サーバのスループットと応答時間を測定し，比較した．また，store-through 方式により消費されるマシン資源の量を知るために，CPU 使用率とメモリ使用量についても測定した．

- **IP Forward**: パケットの転送に必要な最小限の性能を知るために，IP 転送のみの時の性能を計測した．この IP 転送のみの時の性能と，store-through 方式の性能を比較することで，store-through 方式によるオーバヘッドを知ることができる．なぜなら，store-through 方式は IP レベルに追加され，IP 転送時にパケットをコピーするなどの追加の処理を行うためである．

- **Store-through**: 提案方式である store-through の性能である．ここでは，store-through 方式単体のオーバヘッドを測定するために，空のセンサを使用している．すなわち，センサはメッセージを渡されると，中身を検査せずにすぐに `ACCEPT` を返す．

- **Store-through with Sensor**: センサでメッセージの検査を行う場合の store-through 方式の性能である．センサは，メッセージの最初から最後までを 1 バイトずつ，そのバイトの値がある範囲内にあるかどうかを検査していく．この実験では攻撃メッセージは含まないため，センサは全てのメッセージに必ず `ACCEPT` を返し，パケットが破棄されることはない．

- **Proxy**: Store-through 方式との比較のために，プロキシ方式による実装での性能を測定した．プロキシとしては，ウェブ・キャッシュ・プロキシとしてよく知られている Squid 2.5.13 を利用した．ただし，この実験では，他の場合と同様に全てのメッセージがサーバとクライアント間でやりとりされるようにするために，キャッシュはしないよう設定した．

クライアントの同時接続数を1から100まで変えて，スループットと応答時間を計測した．クライアントの同時接続数は，1から10までは1ずつ，10から100までは10ずつ増加させた．CPU使用率とメモリ使用量の計測は，topコマンドを用いた．topコマンドは，CPUやメモリなどのマシン資源の使用量を表示する標準的なUNIXコマンドである．

また，順番が入れ替わったりIPフラグメントされたパケットを処理するためにかかるオーバヘッドを計測するために，故意にパケットをIPフラグメントさせ順番を入れ替えた状態で，スループットと応答時間を測定する実験も行った．第3.3.2節と第3.3.5節で述べたとおり，store-through方式では，順番が入れ替わったりIPフラグメントされたパケットのコピーをとって保存する．そのため，順番が入れ替わったパケットが存在するときは，TCPストリーム再構成機構のオーバヘッドが大きくなると考えられるからである．クライアントとサーバのカーネルに手を加え，送信パケットを故意にIPフラグメントした上で，順番を入れ替えて送信するようにした．フラグメントのサイズは500バイトとし，順番の入れ替えは，フラグメントしたパケットを2つずつ入れ替えている．例えば，１ ２ ３ ４という順のパケットなら，２ １ ４ ３という順に送信する．IPフラグメントをおこす割合は，1%または10%とした．文献[82]によれば，実ネットワークでのIPフラグメントの割合は1%以下という調査結果があるため，実世界に近い1%と，それよりも高い割合の10%の状況下でのオーバヘッドを知ることができる．

3.6.3 実験結果：性能

フラグメンテーションがないとき

図3.5に，第3.6.2節で示した4つの場合について計測したスループットと応答時間の結果を示す．ここでは，故意にIPフラグメントをおこしたり順番を入れ替えたりはしていない．Store-through方式の性能は，全ての同時接続数について，IP転送のみを行うIP forwardの性能とほとんど変わりがなく，オーバヘッドは小さい．IP forwardと比較すると，store-through方式では，スループットが最大3.7%減少（同時接続数が40と70の時）し，応答時間が最大3.8%増加（同時接続数が70の時）した程度であった．一方，プロキシ方式では，同時接続数が1の時にさえ，スループットが7.8%減少し，応答時間が7.0%増加した．また，同時接続数が増加するにつれて，プロキシ方式でのオーバヘッドは大きくなっていった．同時接続

数が 10 を超えると，プロキシ方式のスループットは IP forward の約半分に，応答時間は約 2 倍になった．Store-through 方式では，センサを動作させメッセージの検査を行った場合にも，プロキシ方式よりも，オーバヘッドは小さかった．なお，プロキシ方式ではメッセージの検査は何も行わずに転送している．センサを動作させた store-through 方式のオーバヘッドは，プロキシ方式の半分程度であり，センサがメッセージの検査という重い処理を行った場合にも，性能の低下がプロキシ方式よりも小さいと言える．

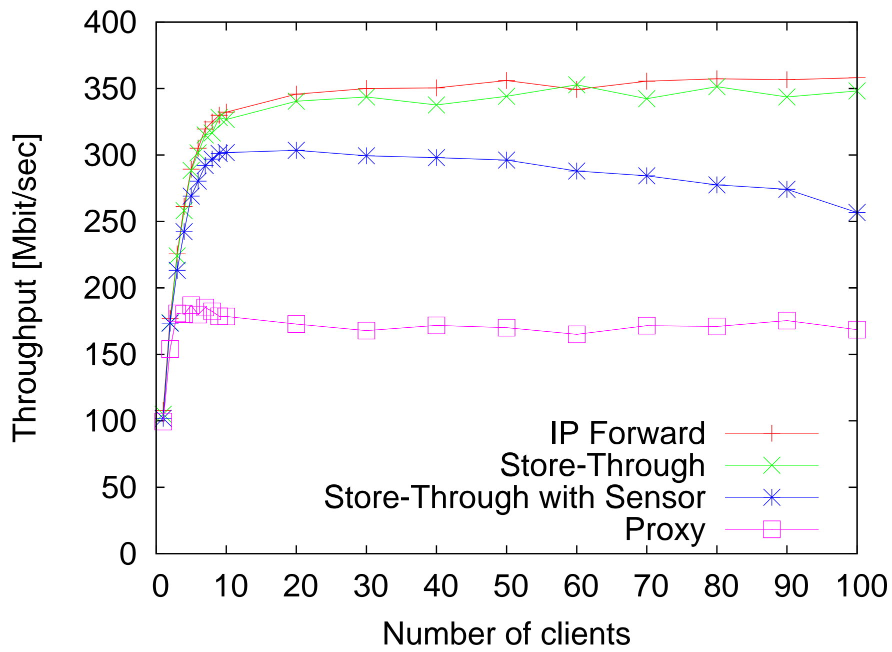
フラグメンテーションがあるとき

図 3.6 にパケットの 1%が IP フラグメントされ順番を入れ替えられた場合のスループットと応答時間を示す．また，図 3.7 に 10%が IP フラグメントされた場合を示す．実験結果より，IP フラグメントや順番が入れ替わったパケットがあった場合にも，store-through 方式の性能にそれほど影響を与えないということが分かった．フラグメントの割合が 1%の時は，store-through 方式では IP 転送に比べて，スループットが最大 2.6%減少し，応答時間が最大 1.4%増加した（ともに同時接続数が 30 の時）．フラグメントの割合が 10%と高いときも，スループットが最大 3.3%減少，応答時間が 3.3%増加（ともに同時接続数が 30）程度であった．これらのオーバヘッドは，前章で示したフラグメントがない状態でのオーバヘッドとほとんど変わらない．Store-through 方式では，IP フラグメントや順番が入れ替わったパケットがある場合，通常に加えて追加の処理が必要となるが，実験結果よりこれらの追加の処理による性能のオーバヘッドは小さいといえる．

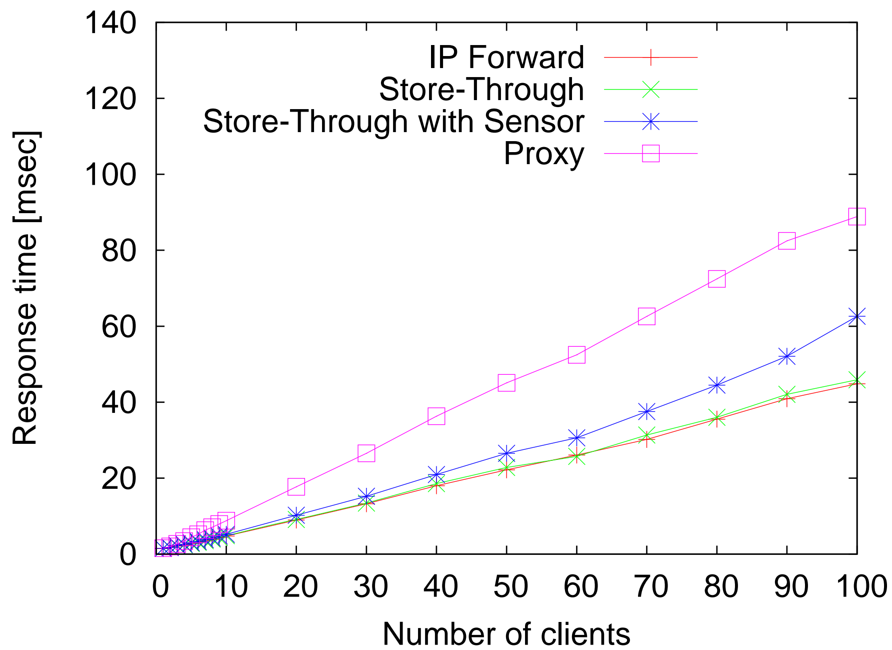
3.6.4　実験結果：CPU とメモリの使用率

フラグメンテーションがないとき

図 3.8 に，IP forward と store-through 方式の CPU 使用率とメモリ使用量を示す．ここでは，故意の IP フラグメントや順番の入れ替えはしていない．このとき，CPU 使用率は同時接続数にかかわらず IP forward の約 2 倍となっている．この CPU 使用率の増加は，コネクション情報の管理や，順番通りのパケットであるかを判定するためのシーケンス番号の計算等，store-through 方式の処理によるものである．しかし，同時接続数が 100 の場合にも，50%程度までの使用率にとどまっており，

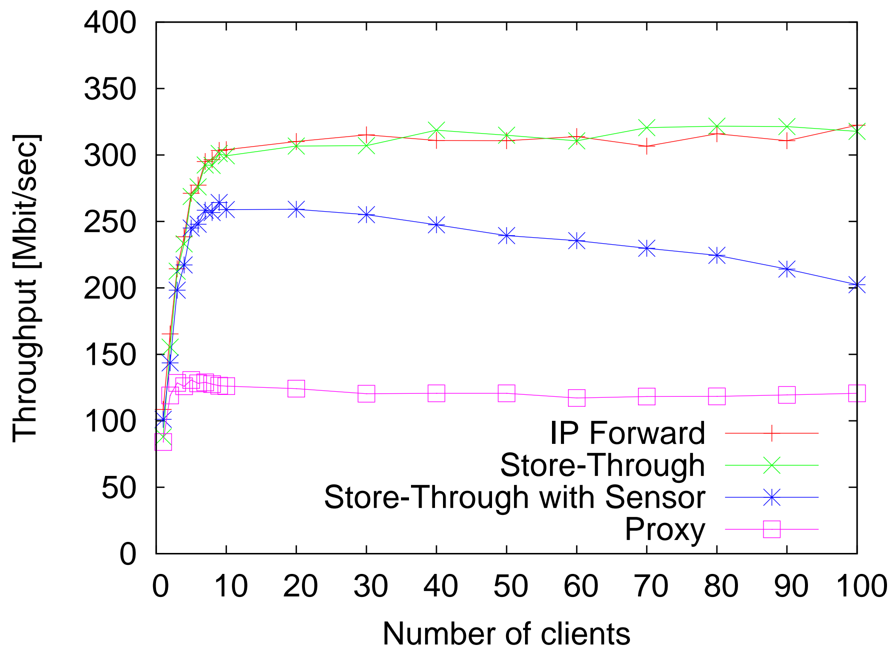


(a) スループット

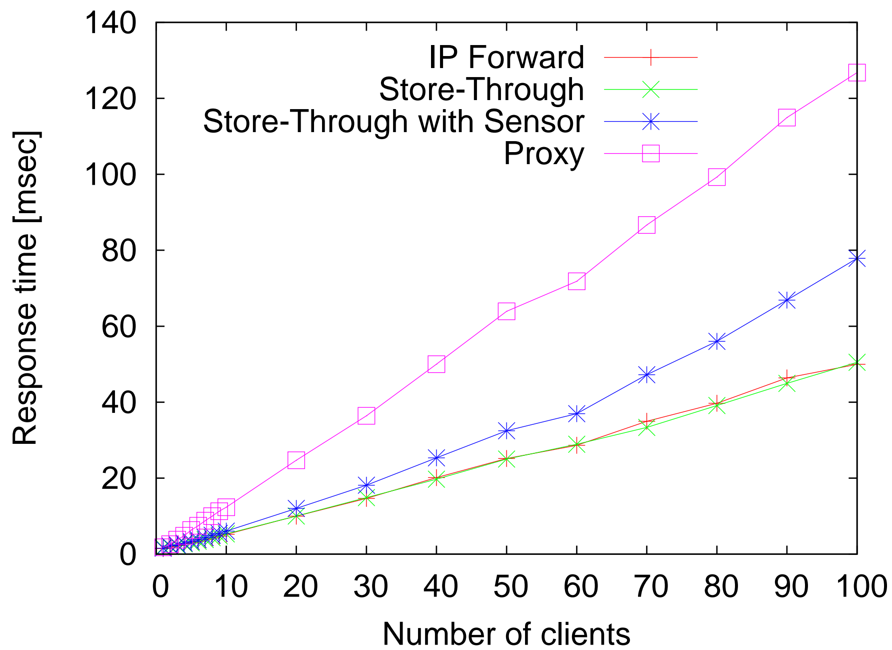


(b) 応答時間

図 3.5: 実験結果：性能（0% フラグメンテーション）

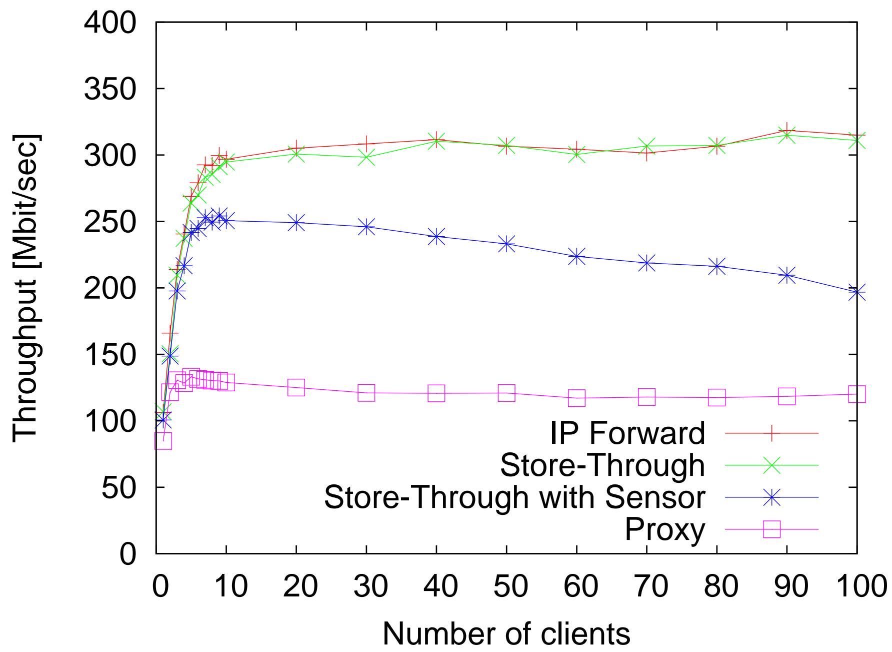


(a) スループット

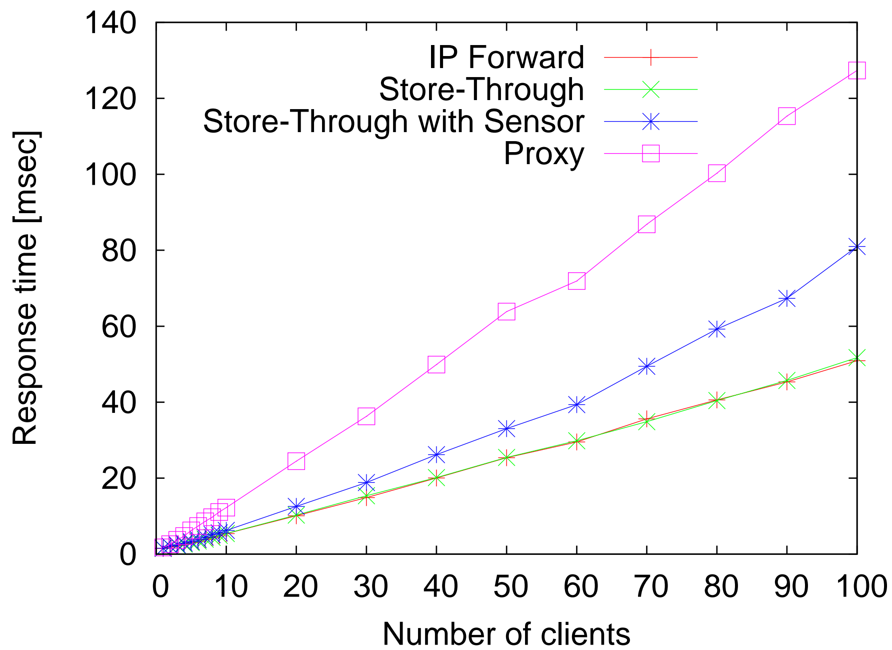


(b) 応答時間

図 3.6: 実験結果：性能（1% フラグメンテーション）



(a) スループット



(b) 応答時間

図 3.7: 実験結果：性能（10% フラグメンテーション）

センサで重い処理を伴うようなッセージの検査のための余地を残している．Store-through 方式によるメモリ使用量も，IP forward より少し多い程度であり，大きなオーバヘッドになるほどではなかった．Store-through 方式では，IP forward より最大 2.4%（同時接続数が 80 と 90 の時）メモリ消費量が多い程度であった．

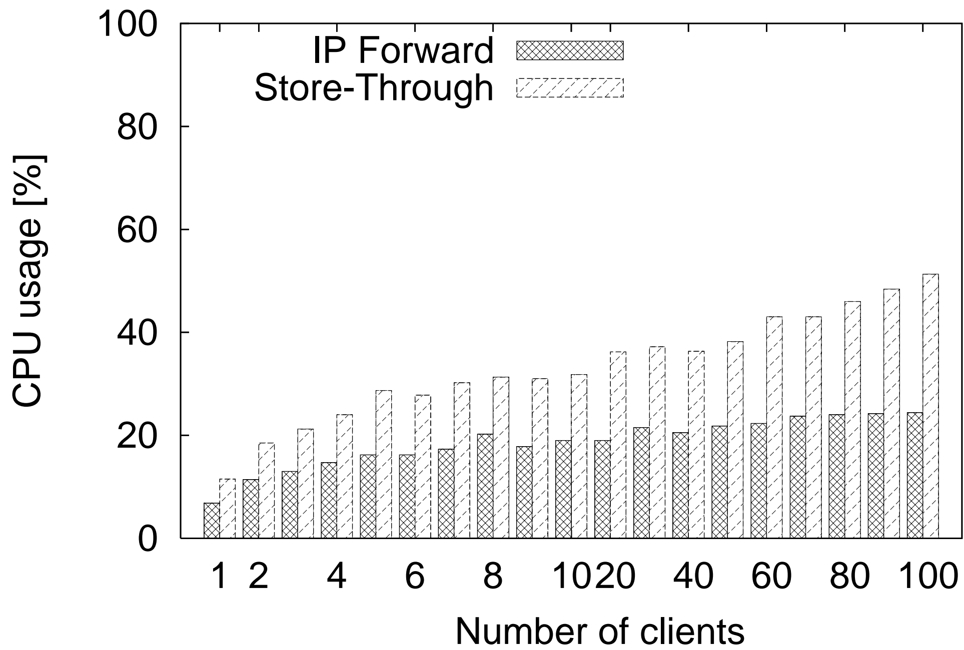
フラグメンテーションがあるとき

図 3.9 にパケットの 1% を，図 3.10 にパケットの 10% を，IP フラグメントし順番を入れ替えた場合の CPU 使用率とメモリ使用量を示す．フラグメンテーションがないときに比べて，IP forward と store-through ともに CPU 使用率は増加しているが，store-through の IP forward に対する割合は変化していない．すなわち，store-through による CPU 使用率は，IP フラグメントされたパケットの割合が 1%，10% の時ともに，同時接続数にかかわらず，IP forward の約 2 倍となっている．しかし，同時接続数が 100 と最悪の場合でも，CPU 使用率が 70% 程度であり，100% 使用率までには達していない．そのため，メッセージを検査するためにセンサが処理をする余地は残っている．

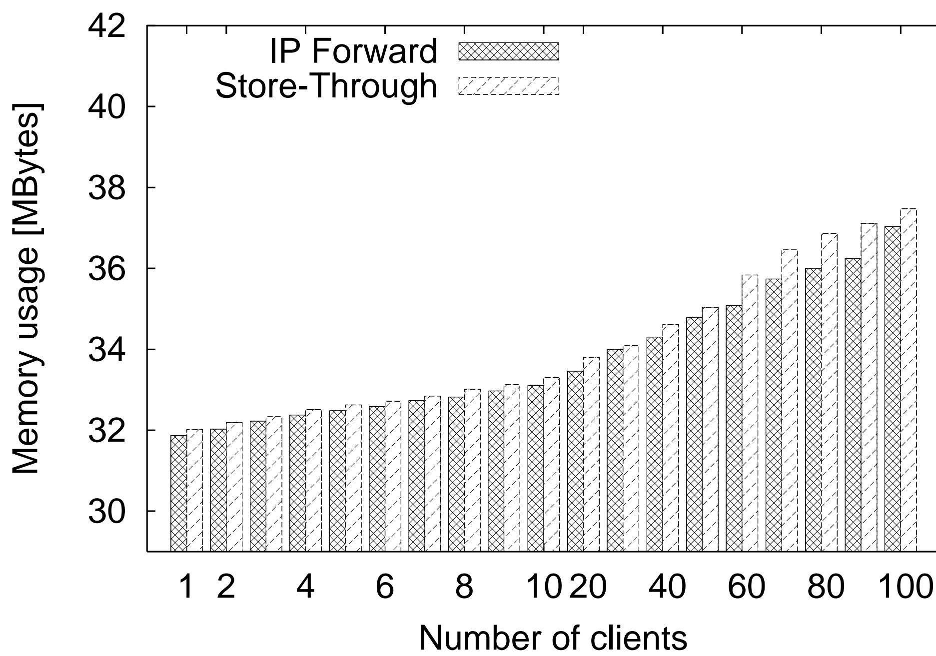
メモリ使用量も，IP フラグメンテーションや順番の入れ替えによって大きく増加する，ということはなかった．IP 転送時のメモリ消費量と比較すると，IP フラグメントされたパケットの割合が 1% の時には最大 3.3% 増加（同時接続数が 100 の時），IP フラグメントされたパケットの割合が 10% の時には最大 4.7% 増加（同時接続数が 90 の時）程度となっており，IP フラグメントがない場合の最大 2.4% 増加に比べてもそれほど大きくなっているわけではない．Store-through 方式では，順番が入れ替わったり IP フラグメントされたパケットはコピーを取って保存するが，順番通りのパケットが届いてメッセージの検査が終わった時点でメモリは解放されるため，メモリ使用量の増加はそれほど多くはならない．

3.6.5 Store-through と Stop-forward の比較

第 2.1.1 節で述べたように，store-through 方式は順番が入れ替わったパケットを止めないためトランスポート透過性を保つが，stop-forward 方式は順番が入れ替わったパケットを止めるため，トランスポート透過性が保たれない．この影響を確かめるために，パケット損失や順番の入れ替わりが起こる状況でクライアント側の輻輳ウィンドウサイズを測定する実験を行った

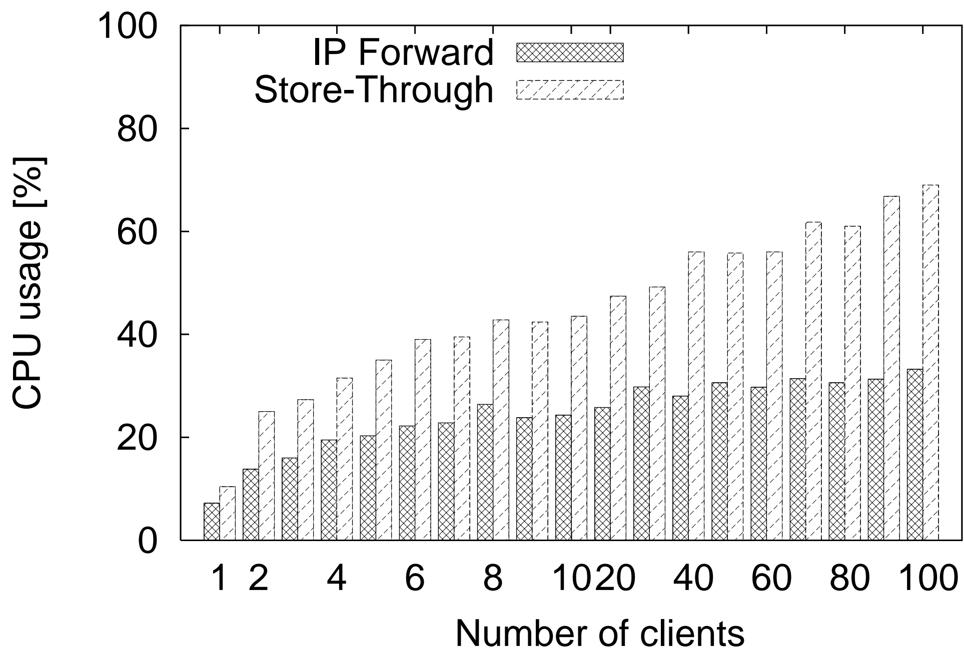


(a) CPU 使用率

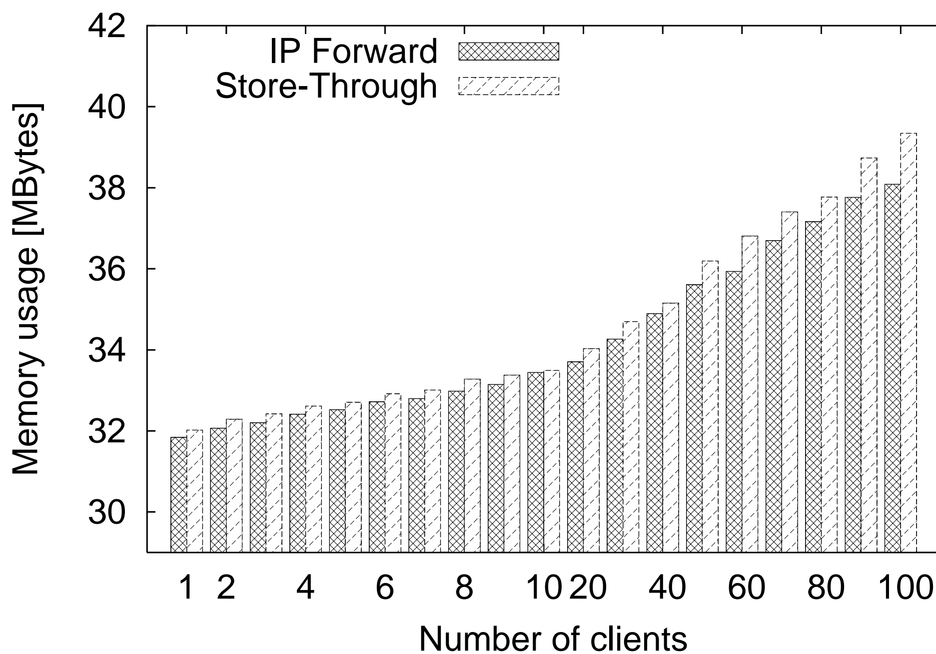


(b) メモリ使用量

図 3.8: 実験結果：CPU とメモリの使用率（0% フラグメンテーション）

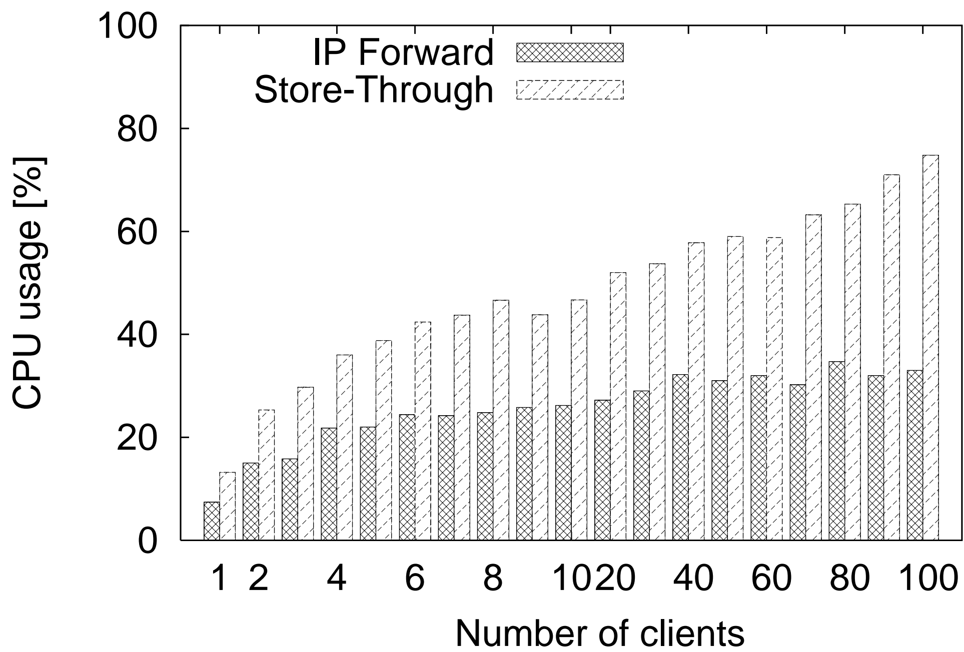


(a) CPU 使用率

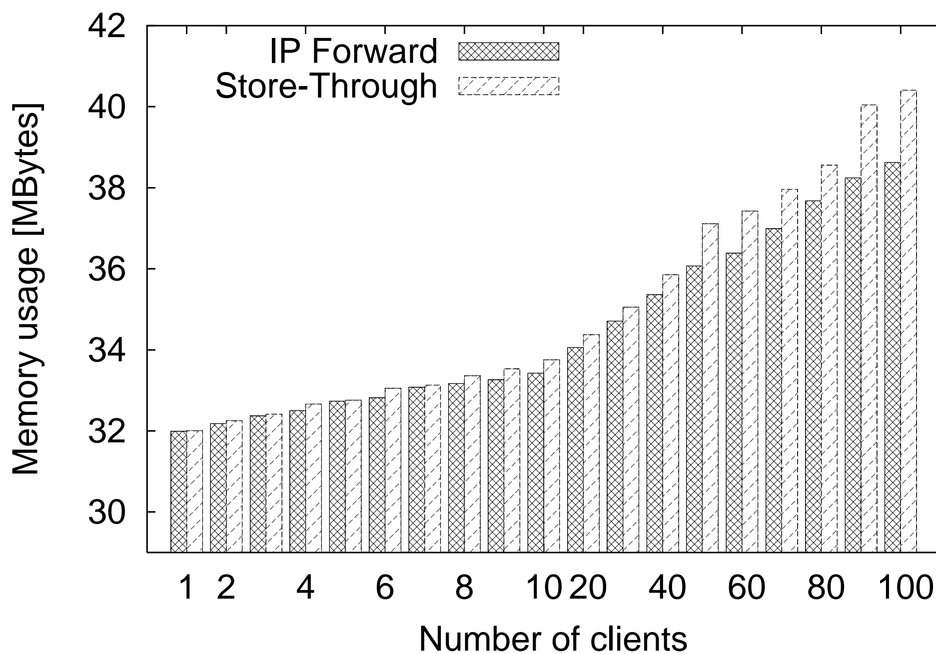


(b) メモリ使用量

図 3.9: 実験結果：CPU とメモリの使用率（1% フラグメンテーション）



(a) CPU 使用率



(b) メモリ使用量

図 3.10: 実験結果：CPU とメモリの使用率（10% フラグメンテーション）

実験環境

この実験では，クライアントマシン，ルータ，NIDS，サーバマシンをそれぞれ1台ずつ用い，この順番で1Gbpsのネットワークで接続した．全てのマシンは第3.6.1節と同じ性能である．OSはLinux 2.6.16を用いた．

クライアントとサーバは，ルータとNIDSを介してDiscard通信 [83] を行う．Discard通信とは，単にデータを送信するだけの通信であり，受信側では届いたデータに対して特別な処理はしない．また，ネットワークの輻輳とパケット損失を再現するため，ルータではtoken bucket filter (tbf) [84] を用いて，帯域幅2MB/s，遅延10ms，キューサイズ15,400bytesというボトルネックをエミュレートした．Store-through方式及びstop-forward方式はNIDS上で動作する．TCPストリーム再構成機構の2方式の比較を行うため，センサは無効とした．クライアントは200MBのデータをサーバに送信し，計測は5回行った．クライアントの輻輳制御アルゴリズムは，LinuxのデフォルトであるBinary Increase Congestion Control (bic) である．

また，トランスポート透過性の実ネットワーク上での影響を確認するため，同様の実験をインターネット経由でも行った．クライアントマシンとサーバマシンを，それぞれ慶應義塾大学と電気通信大学の2つの大学に置いて実験を行った．これらのネットワーク距離は22ホップであった．この実験のクライアントマシンは，Pentium 4 2.80 GHz CPU 2つ，512 MBメモリ，7,200 rpm HDDを持つ．この実験では故意にIPフラグメントや順番の入れ替えはしていない．

実験結果

図3.11にローカルネットワークでのクライアント側の輻輳ウィンドウサイズ，図3.12にインターネット環境での輻輳ウィンドウサイズの変化を示す．見やすさを考慮し，図3.11では5回計測したうちの1回分についてプロットしてある．図3.12では5回全ての計測結果をプロットした．

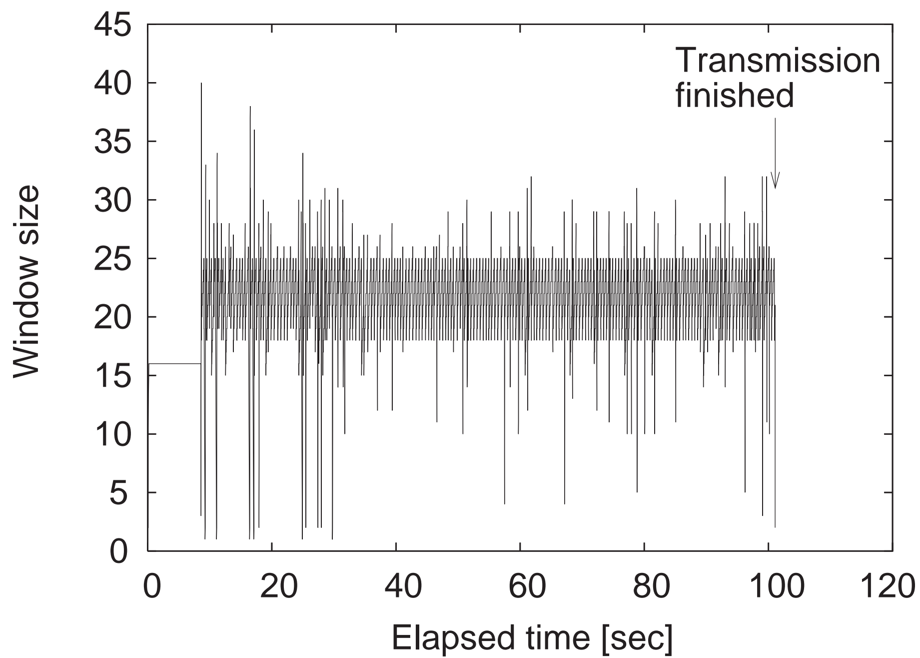
ローカルネットワーク（図3.11）におけるstore-through方式では，パケット損失が発生してウィンドウサイズが一度下がっても，すぐに大きくなる．一方，stop-forward方式では，一度ウィンドウサイズが下がると，しばらく下がった状態が続いてから大きくなる．この傾向は実インターネット環境でも同様である（図3.12）．Store-through方式でのウィンドウサイズは一度下がってもすぐに上昇するが，stop-forward方式では一度下がると，上昇する前にしばらく下がったままの状態が続く（例えば140秒付近）．

表 3.3: 送信所要時間の比較 (5 回測定の平均)

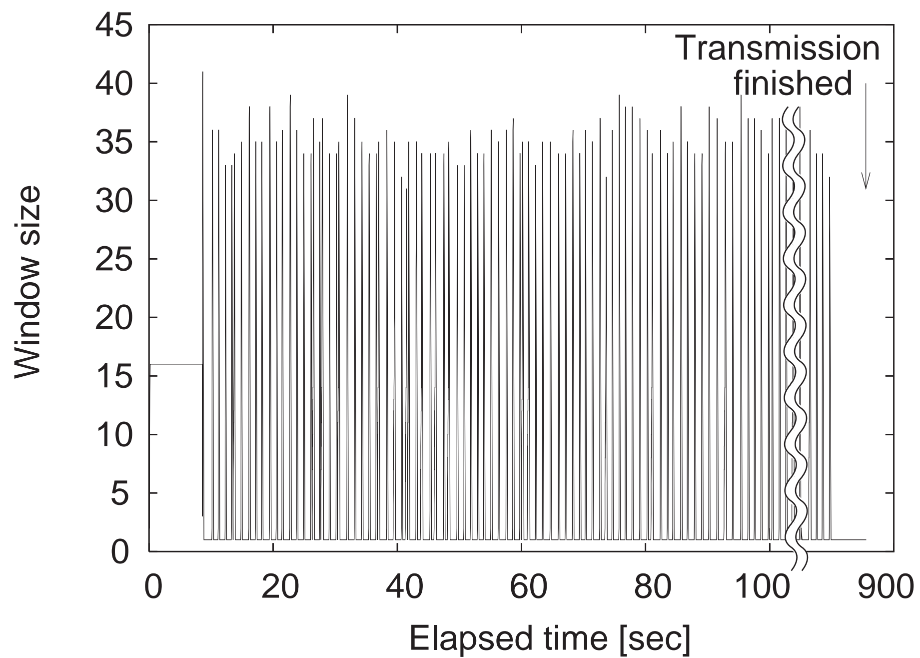
Time [sec.]	Store-through	Stop-forward	ratio
Local			
no packet drop (tbf off)	16.022	15.914	−0.7%
packet drop (tbf on)	119.326	1039.012	×8
Internet			
no packet drop	34.330	34.601	+0.7%
packet drop	92.840	188.828	×2

この結果より，store-through 方式はトランスポート透過性を保持するが，stop-forward 方式はトランスポート透過性を保持しないことが分かる．これは，stop-forward 方式ではパケットの転送を止めるからである．第 2.1.1 節で述べたように，stop-forward 方式では順番通りでないパケットを，その前のパケットが到着するまで，転送せずに止める．そのため，輻輳が起きて 1 つのパケットが損失すると，サーバにはパケットが 1 つも届かず，クライアントは損失したパケット以外にも全てのパケットを再送しなければならなくなる．これを確認するため，ローカルネットワークでの再送パケット数を計測した．5 回測定した平均再送パケット数は，store-through は 657 個であるのに対し，stop-forward は 38 倍の 21,231 個であった．

Stop-forward 方式では，トランスポート透過性が保持されないことで，性能にも影響を与えている．表 3.3 にデータ送信所要時間の平均値を示す．パケット損失がないときの，store-through 方式と stop-forward 方式の送信所要時間は，差が 1%以下とほとんど変わらない．しかしながら，ネットワークの輻輳によりパケット損失がおきたときの stop-forward 方式の送信所要時間は，store-through 方式に比べて，ローカルネットワークで 8 倍，インターネット環境で 2 倍となる．送信所要時間が増える理由としては 2 つ考えられる．まず 1 つ目は，再送パケットが増え，必要のないパケットが送信されているためであると考えられる．クライアントは，損失したパケットだけでなく NIDS で止められているパケットまで再送する．2 つ目は，ウィンドウサイズが小さいためであると考えられる．Stop-forward 方式では，一度ウィンドウサイズが小さくなると，しばらく小さいままとなりすぐに大きくならない．ウィンドウサイズが小さいと，クライアントは一度に多くのパケットを送信することができず，ACK が返ってくるのを待たなければならなくなる．

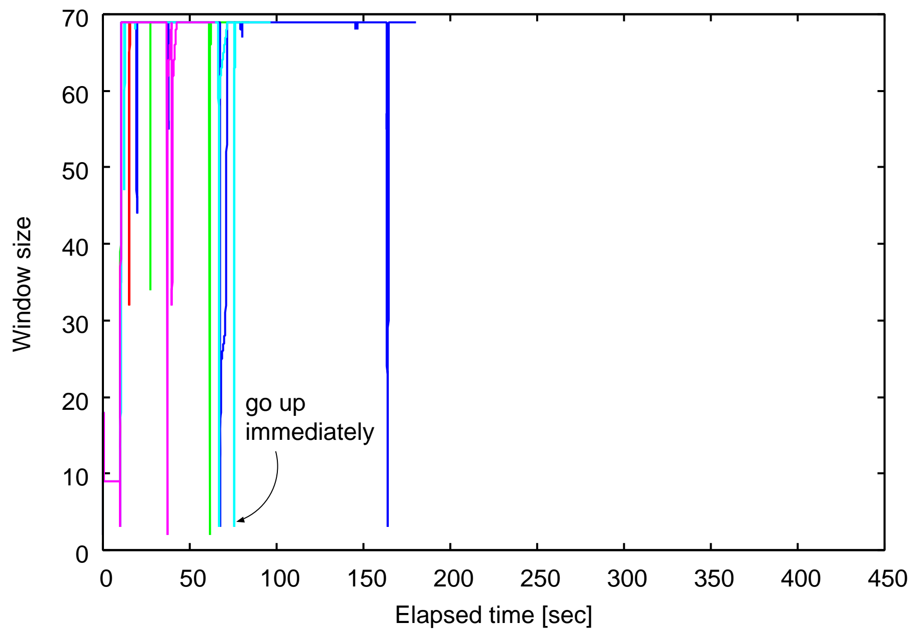


(a) Store-through

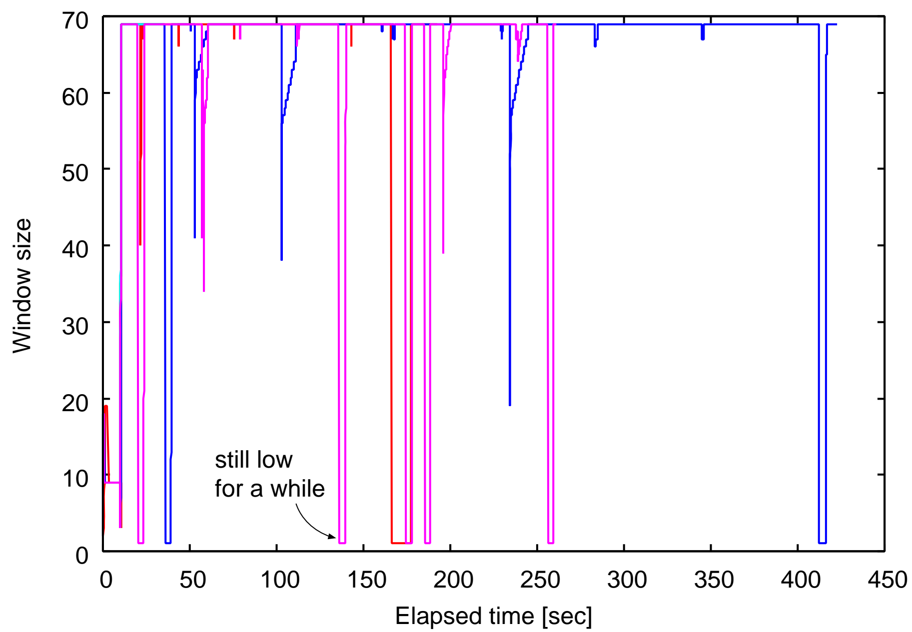


(b) Stop-forward

図 3.11: token bucket filter を用いたローカルネットワークでの輻輳ウィンドウサイズの変化



(a) Store-through



(b) Stop-forward

図 3.12: インターネット経由での輻輳ウィンドウサイズの変化

3.7 まとめ

本章では，レイヤ7コンテキストを考慮したNIDS/NIPSのためのTCPストリーム再構成機構について述べた．近年の攻撃の高度化・巧妙化に対応して，レイヤ7コンテキストを考慮した検知が，NIDSでの攻撃メッセージの検出方法として，注目されている．レイヤ7コンテキストを用いることで，単なるバイト列とのマッチングに比べて，高い精度の検知が可能となるからである．レイヤ7コンテキストを考慮した検知では，メッセージの検査を行い攻撃の検知を行うセンサの他に，個々のパケットを並べ直しメッセージに再構成するTCPストリーム再構成機構が必要となる．

本章では，レイヤ7コンテキストを考慮したNIDS/NIPS（レイヤ7 NIDS/NIPS）のためのTCPストリーム再構成機構の実装方式*store-through*を提案した．レイヤ7NIDSのためのTCPストリーム再構成機構は，1) 完全な防御，2) 性能，3) アプリケーション透過性，4) トランスポート透過性 の4つの要件を満たしていなければならない．完全な防御は，攻撃メッセージが攻撃対象のアプリケーションに届くのを防ぎ，攻撃が成功することがないことをいう．アプリケーション透過性は，NIDSの設置によって，クライアントやサーバの変更や再設定が必要ないことをいう．トランスポート透過性は，NIDSがend-to-endのTCP/IPセマンティクスを壊さないこと，すなわちTCPのフロー制御や輻輳制御に影響を与えることが少ないことをいう．

我々の提案手法store-throughはこれら4つの要件全てを満たす方式である．まず，プロキシ方式で必要な2回のTCP/IPプロトコル・スタックの通過や2回のユーザ・カーネル空間間のデータ・コピーが必要ないため，性能は単なるIP層における転送と同等のものが得られる．また，プロキシ方式のように，コネクションを分断しないため，アプリケーション透過性が保たれる．さらに，stop-forward方式のように順番が入れ替わったパケットも止めずに，コピーを取った後にすぐに転送するため，トランスポート透過性も保たれる．最後に，サーバマシン内で攻撃メッセージが全てそろうことがないように，パケットを適切に破棄するため，完全な防御が達成できている．

Store-throughをLinux2.4.30に実装し，実験を行ったところ，store-throughの性能は，単なるIP層での転送と同等の性能を得ることが分かった．また，CPUやメモリの消費量もさほど増加せず，センサが動作する余地が残る程度であった．また，パケットを止めない方式であるため，トランスポート透過性を保持している

ことを，故意にパケット損失がおこる環境や実ネットワークでの実験によって示した．

第4章　NIDSの協調による性能向上と耐障害性向上

本章では，組織内ネットワークに設置されたNIDS同士を協調させるシステムBrownieを提案し，これを用いることで性能向上や耐障害性向上ができることを示す．まず，複数のNIDSを協調させることによる，性能向上と耐障害性向上の各手法の概要を述べ，これらのトレードオフについて議論する．その後，設計と実装及び，実験を示す.

4.1　提案

4.1.1　概要

本論文では，組織内ネットワーク内の複数の場所に置かれたNIDS同士を連携させることで，性能向上や耐障害性向上を行う手法を提案する．本手法では，同じネットワーク内のNIDS同士がそれぞれの負荷状況やルール設定を交換し合い，適切に再設定していく．同じネットワーク内に設置されたNIDSであるため，NIDS同士は相互に信頼して連携を行うことができる．通常時はNIDS同士の負荷を分散させることで性能向上を，障害時は障害が発生したNIDSの攻撃検知を他のNIDSで代替することで耐障害性向上を行う．従来のように，1カ所に並列に動作する複数のNIDSを置くのではなく，既にネットワーク内にあるNIDSを活用することでネットワーク内全体の性能向上と耐障害性向上を目指す.

我々の着目点は，大学や会社など多くの組織では，インターネットと組織のネットワークの間に置かれるNIDS以外にも，組織内ネットワークの様々な場所にNIDSが置かれていることが多くある，ということである．すなわち図4.1のように，組織内ネットワークの入り口に置かれたNIDS A以外にも，その下流に，各部門や課によって独自のNIDSが置かれることがある．例えば大学では，学科や研究室でそれぞれNIDSを設置するといったことが考えられる．図4.1の例では，NIDS B1

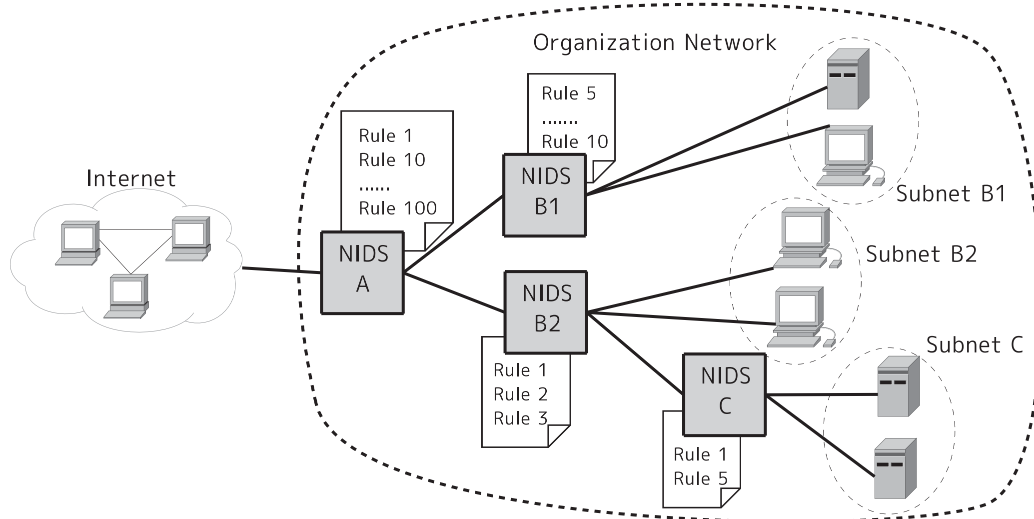


図 4.1: NIDS の設置例

と B2 は学科によって，NIDS C は研究室によって設置された，と考えられる．本手法では，各 NIDS はそれぞれの一つ上流の NIDS (親 NIDS) とすぐ下流の NIDS (子 NIDS) と通信をする．例えば，NIDS A は NIDS B1 および B2 と，NIDS B2 は NIDS A および C とやりとりをする．

なお，本提案では，NIDS のネットワーク・トポロジが木構造であることを仮定する．通常，組織内のネットワークは組織の構造に従うことが多いと考えられるため，NIDS も木構造となっている場合が多いと考えることができる．また，組織内の NIDS 同士は信頼できると仮定する．同じ組織内に属する，それぞれの部署の管理者が設置する NIDS であるため，悪意を持った NIDS はないと仮定できる．ただし，組織内に存在する全てのクライアントやサーバ同士が信頼できるものと仮定する訳ではない．

以下，性能向上と耐障害性向上の各手法について述べた後，性能向上と耐障害性の間に本質的に存在するトレードオフについて議論する．

4.1.2 性能向上

提案手法では，他の NIDS と協調し，1) 過負荷になった NIDS の負荷を減少させ，2) NIDS 間の冗長なルール設定を削除する，という 2 つのアプローチをとることにより性能向上を目指す．この 2 つのアプローチにより，負荷が NIDS 間で分散され，効能向上を達成することができる．以下，それぞれのアプローチについて

述べる．

過負荷 NIDS の負荷の減少

あるNIDSが過負荷になると，過負荷になったNIDSがネットワーク全体のボトルネックとなり性能低下の要因となる．NIDSが過負荷になる要因としては，大量のトラフィックを受け取った，多くのルールを検査しなければならない，マシンが低性能であるなどが挙げられる．組織内に置かれた複数のNIDSのうち，全てが過負荷になるという状況はほとんどないが，1つのNIDSが過負荷になると，そのNIDSが全体のボトルネックになる．そこで，過負荷になったNIDSの負荷の一部を，ネットワーク経路上にある他の低負荷であるNIDSに分散することで，過負荷になったNIDSの負荷を減らす．これにより，過負荷NIDSがボトルネックになることを防ぐことができ，結果としてスループットの向上や通信遅延の減少が期待できる．

負荷を分散させるために，提案システムでは，過負荷なNIDSから低負荷なNIDSにルールを移譲する．すなわち，過負荷なNIDSでいくつかのルールを無効にし，代わりに低負荷なNIDSで同じルールを有効にする．トラフィックの量やマシンの性能をすぐに変更することは難しいため，ルールの数を減らすことで負荷を減らす．例えば図4.1では，NIDS Aに多くのルール設定がされており，過負荷であるとする．一方，その下流にあるNIDS B1, B2は設定されているルール数が少なく，低負荷である状況を考える．このとき，NIDS AはNIDS B1, B2に，例えばルール70〜100を移譲する．つまり，ルール70〜100をNIDS Aで無効にし，代わりにNIDS B1, B2で有効にする．こうすることで，NIDS Aの負荷の一部をNIDS B1, B2に移すことができる．

ここで，ルールを移譲した後も，元のセキュリティレベルは維持していることに注意してほしい．すなわち，ルールの移譲前と移譲後で，各パケットに適用されるルールは変わらない．例えば，組織外からのネットワークについて考えると，提案手法では，元々ルールを有効にしていた過負荷NIDSを通る組織外からの全てのトラフィックは，ルールを移譲した先のNIDSのどれかを必ず通る．そのため，組織外からのトラフィックは，宛先マシンに届く前に，移譲されたルールに対して必ずチェックされることが保証されている．例えば，前述の例のようにルールが上流から下流のNIDSに移譲される場合を考える．これは，図4.1においてNIDS AからNIDS B1とB2にルールを移譲したときである．このとき，上流NIDS (NIDS

A) を通る組織外からの全てのパケットは，下流 NIDS のどれか 1 つ (NIDS B1 か B2) を通ることになる．すなわち，組織外からの全てのパケットは下流 NIDS のどれか 1 つ (NIDS B1 か B2) で，移譲されたルールに対してチェックされることになる．ここでは，組織外からのネットワークトラフィックについて述べたが，本システムでは組織内のネットワークトラフィックも考慮して，全てのパケットが移譲前と同じルールに適用されることを保証することにより，セキュリティレベルの維持を行っている．セキュリティレベル維持についてのさらに詳しい議論は，第 4.2.1 節で述べる．

重複ルールの除去

あるネットワーク経路上に置かれた複数の NIDS が，同じルールを重複して有効にしていることは，珍しいことではない．これは，全ての NIDS が同じ管理者によって管理されているわけではなく，各 NIDS がそれぞれ別々の管理者によって管理されているためである．例えば前述の大学の例では，大学の入り口に置かれた NIDS は大学のネットワークの管理者によって管理されるが，学科の NIDS は学科のネットワーク管理者によって管理される．ネットワーク経路上に置かれた複数の NIDS に同じルールが設定されていると，その経路を通るパケットは同じルールに対して何度もチェックされることになる．重複しているルールをなくし，ネットワーク経路上のどこか一カ所だけで検査すれば，通信遅延を少なくすることができると考えられる．ここで，前節と同様に，重複したルールをなくすことで，セキュリティレベルが低下することはない．ルールの無効化は，ネットワーク経路上の他の NIDS でも同じルールが有効にしており，その NIDS でチェックされる時のみ行うからである．

例えば図 4.1 では，NIDS A, B2, C の 3 つの NIDS でルール 1 を有効にしている．そのため，NIDS C 以下にあるマシン宛のパケットは，ルール 1 に対して 3 回検査されることになる．そこで，例えば NIDS B2 と C ではルール 1 の設定を無効にして，NIDS A のみでルール 1 について検査するようにすれば，その分通信遅延やマシンの負荷を減らすことができる．また，NIDS B2 と C でルール 1 を無効にすることで，セキュリティが低下することはない．なぜなら，NIDS B2 と C に届く組織外からのトラフィックは，NIDS A ですでにルール 1 に対してチェックされているからである．

4.1.3　耐障害性

NIDS に障害が発生すると，通常，障害が発生した NIDS が検査していたトラフィックは素通りとなり，攻撃の検知ができなくなる．NIDS の障害の原因としては，通常のサーバと同様にハードウェア故障等が考えられる．そこで，本手法では障害が起こった NIDS で有効にしていたルールを別の NIDS で有効にすることで攻撃検知を代替する．障害の検知は，定期的に送り合っている負荷情報をハートビートとして用いることで行うことができる．一定時間ハートビートが送信されて来なければ，何らかの原因によりその NIDS に障害が起こったと考えられる．そして，障害が起こった NIDS が有効にしていたルールを，自 NIDS で有効にする．各 NIDS は，自分の上流・下流の NIDS のルール設定を交換しているため，障害が起こった NIDS でどのルールを有効にしていたかは容易に知ることができる．

例えば図 4.1 で NIDS C が何らかの障害により動作を停止した場合を考える[1]．障害が起こる前は，サブネット C のトラフィックについて，NIDS C でルール 1 と 5 の検査を行い，攻撃検知をすることができた．NIDS C に障害が起こった場合，NIDS C で検査をしていたルール 5 について，トラフィックの検査はされなくなる．そのため，ルール 5 で検知できる攻撃の検知・防御ができなくなる．そこで，上流 NIDS である NIDS B2 でルール 5 を有効にし，サブネット C へのトラフィックを検査することで，NIDS C の代わりに攻撃の検知を行う．ここで，NIDS B2 と C は互いに有効にしているルール設定を交換しているため，NIDS B2 は NIDS C で有効にしていたルールを容易に知ることができる．また，NIDS B2 と C は定期的に負荷情報を交換しているため，これをハートビートとして用いることで，NIDS B2 は NIDS C の障害を検知することができる．

4.1.4　性能向上と耐障害性向上のトレードオフ

本論文では，NIDS の協調による性能向上と耐障害性向上の手法を提案している．しかし，性能と耐障害性は簡単には両立できるものではない．なぜなら，これらは基本的に相反するものだからである．高性能を得るためには，できるだけ処理が分散されるようにしなければならない．一方，耐障害性を得るためには，処理を冗長にしてどれかに障害が起きたときも他のマシンで代替できるようにしておかなければならない．

[1]なお，ここでは分かりやすさのために，第 4.1.2 節で述べたルールの移譲や削除を考えず，図 4.1 のルール設定の状態で障害が起こったと仮定する．

そのため，本機構では，管理者が性能重視や耐障害性重視の設定を柔軟に選択できるようにした．管理者は，性能と耐障害性のトレードオフを考慮し，自組織に合致した設定を行う．例えば，耐障害性を最優先した設定をした場合，全てのNIDS で同じルールを有効にすることで完全冗長とする．これは，前述した性能向上や耐障害性向上の手法を動作させることはせず，NIDS 同士がルール設定を交換した時点で，相手の NIDS が有効にしているルール全てを自 NIDS でも有効にすることで行う．これにより，ある NIDS が障害で停止しても，すぐに他の NIDS で同じルールに対して検査され，攻撃を検知することができる．しかし，全ての NIDS で完全冗長のルール設定を行うため，性能は低下する恐れがある．

一方，性能を最優先した設定をした場合，本機構の性能向上手法により，ルールの再配置と冗長ルールの削除が行われる．その結果，負荷分散され性能向上ができる．しかし，ある NIDS が障害で停止すると，一時的に障害が起こった NIDS で有効になっていたルールについて検査されなくなる．本機構では，耐障害性向上の手法により，このルールを他の NIDS で検査することで耐障害性向上を行う．しかし，耐障害性を最優先にしたときの完全冗長に比べると，検査されない期間ができる．ただし，本機構がない場合，管理者が障害を検知して対策を行うまでの時間，検査されないルールがあるということに注意して欲しい．本機構の耐障害性向上手法による検査復帰までの時間は，NIDS 同士のハートビートの間隔と障害検知の閾値によって変わるが，通常管理者が対策を終了するまでの時間と比べれば十分短いと言える．

また，これらの中間の設定として，冗長度を制限したり，冗長化するルールを重要度が高いものだけに限定する等で，性能と耐障害性のトレードオフを考慮した設定をすることができる．

4.1.5　本手法導入による影響の可能性

この節では，本手法導入によるシステムへの悪影響の可能性について議論する．例えば，可用性の問題である． 一般に分散システムなどでは，システムが連携し複雑になると，システムの可用性が低くなる可能性がある．しかし，本手法を導入した場合においても，ルール設定を参照しながら，ネットワークトラフィックを検査する部分は個々に動作する．そのため，NIDS の動作に障害が起こった場合においても，他の NIDS にその障害が伝搬することはない．一方，ルール設定に関しては，互いに連携してルール設定を行うため，他の NIDS のルール設定の影響を受

ける．例えば，悪意のある NIDS が存在した場合には，負荷の高いルールを多数有効にしておくことで，他の NIDS が過負荷になる可能性がある．しかし本システムは，組織内での NIDS 同士の協調であるため，互いの NIDS 同士は信頼してよく，悪意のある NIDS は存在しないと仮定できる．ネットワークや NIDS の管理者は，直接接続してる上流・下流のネットワーク構成については熟知していると考えられ，認証等の方法により相互に信頼できる NIDS とのみ接続することは難しくない．

4.2　設計と実装

NIDS 協調によって性能向上や耐障害性向上ができることを示すため，NIDS 協調システム Brownie を実装した．各 Brownie は各 NIDS を管理し，他の Brownie と通信をおこない，それぞれが管理する NIDS を適切に設定することによって，NIDS 同士の協調を実現させる．以降，特に紛らわしい場合を除いて，Brownie はシステム全体及び各 NIDS を管理する各インスタンス双方を指すのに用いる．

NIDS を Brownie と共に動作させるには，管理者は親 NIDS の IP アドレスなどの情報を Brownie に与える．NIDS の起動時，対応する Brownie は親 NIDS と共に動作している Brownie とコネクションを確立し，以降それぞれの負荷状況やルール設定を定期的に交換する．障害検出にはこれをハートビートとして用いる．また，コネクション確立時に，それぞれの NIDS で有効または無効にしているルール設定を交換する．

NIDS としては，広く普及しているオープンソース NIDS である Snort [11] を用いた．現在の実装では全ての NIDS が Snort であるが，将来的には様々な NIDS が動作している環境にも適用できるようにしたいと考えている．もう 1 つのオープンソース NIDS である Bro [12] は，動的ルール書き換えやセンサ間の通信機構を備えているという点で，より本システムに向いていると考えられ，今後用いる NIDS の候補として考えられる．

4.2.1　性能向上

提案手法では，他の NIDS と協調し，1) 過負荷になった NIDS の負荷を減少させるオフローディング，2) NIDS 間の冗長なルール設定を削除する，の 2 つのアプローチをとることにより性能向上を行う．

過負荷 NIDS の負荷を減少

NIDS の負荷を減少させるオフロード手順の基本的な考え方は，自分と子 NIDS の負荷を比較し，負荷が高い方から低い方にルールを移譲する，ということである．つまり，自分の NIDS の負荷が，どの子 NIDS の負荷よりも高い場合，いくつかのルールを子 NIDS に移譲する．逆に自分の NIDS の負荷がどの子 NIDS の負荷よりも低い場合，いくつかのルールを子 NIDS から自分の NIDS に移譲する．これを，過負荷 NIDS の負荷が十分に減少するまで行う．ここで，1) いつオフロードを行うか，2) どのルールを移譲するかについて説明する．

オフロードの開始・終了判断　まず，Brownie はオフロードが必要かを判断するために，NIDS の負荷を測る．NIDS の負荷は資源使用量，NIDS では特に CPU 使用率で知ることができる．これは NIDS の処理のほとんどは CPU インテンシブなパターンマッチングであるためであり，CPU 使用率を用いることでマシンの性能が異なる場合でも適切に負荷を知ることができる．

その後，Brownie は測定した CPU 使用率を元にオフロードが必要かを判断する．過負荷の NIDS の負荷を減少させるという観点から，NIDS が過負荷になったら（CPU 使用率が 100% 近くなったら）オフロードを開始し，CPU 使用率がある設定値を下回ったら終了する，という方法が考えられる．しかしこの方法にはいくつかの問題点がある．まず第一に，終了させるための CPU 使用率を決定するのが困難である．負荷を適切に減少させるためには小さい値がよいが，小さすぎると今度は逆にルールを移譲された NIDS が過負荷に陥る．第二に，もし全ての NIDS が過負荷だった場合，互いにルールを移譲しあうにも関わらず過負荷は解消されないという状態が続く．最後に，もし他の NIDS の負荷が軽いのであれば，NIDS が過負荷に陥る前にオフロードを開始した方が良いと考えられる．

そこで，過負荷の NIDS の負荷を減少させるというよりむしろ，負荷分散を目指す手法を用いた．すなわち，Brownie は自分が管理している NIDS の CPU 使用率と子 NIDS の CPU 使用率が同じくらいになるように調整していく．Brownie は，自分が管理している NIDS の CPU 使用率と子 NIDS の CPU 使用率との間にある程度の差があれば，ルールを移譲し，CPU 使用率の差が小さくなればルール移譲を終了する．上記の方法と違い，全ての NIDS が過負荷であるときは，ルールの移譲は起こらない．また，Brownie は自分が管理する NIDS が過負荷に陥る前に，負荷の軽い他の NIDS にルールを移譲することで負荷を割り振ることができる．NIDS

間の許容CPU使用率差をパラメータ *Diff* として定め，実験では5と設定した．

移譲ルール選択　効率よくルールを移譲できれば，それだけ早く過負荷NIDSの負荷が減る．もちろん，Brownieでは負荷分散されるまでオフロードを繰り返すため，一度のルール移譲でオフロードが完了できる必要はない．しかし，何度もルール移譲を繰り返して長い時間がかかるよりは，できるだけ短い時間で負荷分散が完了できる方が好ましい．

最も単純なルールの選択方法は，一定数のルールをランダムに選ぶことである．しかし，この「一定数」を設定するのは困難である．もしこの値が小さすぎれば（例えば，1）オフロードが完了するまでの時間が長くなりすぎる．逆にこの値が大きすぎれば，負荷の軽かったNIDSが過負荷になり，同じルールを元々過負荷だったNIDSに移譲し戻すということが起きる可能性がある．

そこで，Brownieは移譲するルールの数を，自分と子NIDSのCPU使用率の差に基づいて決め，この数のルールをランダムに選ぶ．負荷の差が大きいときはより多くのルールを移譲することで，早く負荷を分散させる狙いである．もちろん1つのルールが与える負荷は同じであるとは限らないが，有効にされているルールが増えれば，負荷が増える可能性は高いと考えることができる．

処理手順　以上より，過負荷NIDSの負荷を減少させるためのルール移譲の具体的な手順は，以下の通りとなる．

1. 自分が管理するNIDSと子NIDSのCPU使用率を収集する．自分のCPU使用率を c_{my}，子NIDSのCPU使用率を c_i $(0 \leq i \leq n$, n は子NIDSの数) とする．
2. c_i のうち最大のものを c_{max}，最小のものを c_{min} とする．
3. 必要があればルールを移譲する．
   - $c_{my} - max(c_i) > \mathit{Diff}$ (*Diff* は正の数なので，$c_{my} > max(c_i)$) であれば，Brownieは自分の管理するNIDSから全ての子NIDSにルールを移譲する．
     移譲するルール数は，*Factor* $\times(c_{my} - max(c_i))$．
   - $min(c_i) - c_{my} > \mathit{Diff}$ (同様に，$min(c_i) > c_{my}$) であれば，Brownieは全ての子NIDSから自分の管理するNIDSにルールを移譲する．
     移譲するルール数は，*Factor* $\times(min(c_i) - c_{my})$．

表 4.1: セキュリティ確保を考察する 4 つのケース

		ルール移譲の方向	
		上流から下流	下流から上流
下流マシン	全て NIDS	ケース 1	ケース 2
	一部 NIDS 以外	ケース 3	ケース 4

- どちらでもない場合，すなわち $min(c_i) - Diff \leq c_{my} \leq max(c_i) + Diff$ であれば，負荷は分散していると考え Brownie はルールの移譲を行わない．

4. 1〜3 を *T* 秒ごとに繰り返す．

パラメータ *T* は負荷の変化に対する感度と Brownie による CPU 使用率収集のオーバヘッド等を考慮して設定する．また，この値は，耐障害性向上機能において負荷情報の収集をハートビートとして用いるため，障害の検知速度にも影響する．実験では，30 秒と設定した．また，パラメータ *Factor* は一回で移譲するルールの数を決定する．実験では 10 とした．

セキュリティの確保　第 4.1.2 節で簡単に述べたとおり，Brownie がルールを移譲することによって，セキュリティを低下させることはないようにしている．すなわち，Brownie 導入前に検知できた攻撃は，Brownie 導入後でも検知できることを保証している．なお，Brownie 導入前に検知できなかった攻撃は Brownie 導入後でも検知できない．この節では，組織外及び組織内からの脅威を考慮したときの，セキュリティ確保の問題について詳細を議論する．表 4.1 に示すように，次の 2 軸にそった 4 つのケースに分けて，ルールの移譲後のセキュリティの確保を考える．1 つ目の軸は，ある NIDS に直接接続された下流のマシンが，全て NIDS であるか，一部は NIDS でないか，である．2 つ目の軸は，ルールの移譲方向が下流 NIDS から上流 NIDS であるか，上流 NIDS から下流 NIDS であるか，である．

ケース 1 では，上流 NIDS の直下にあるマシンは全て NIDS であり，ルールは上流 NIDS から下流 NIDS に移譲する．図 4.1 の例で考えると，NIDS A の直下には NIDS のみ (NIDS B1 と B2) が接続されており，NIDS A から NIDS B1 と B2 にルールが移譲する場合である．この場合，上流 NIDS (NIDS A) を通過する全てのトラフィックは，下流 NIDS のいずれか (NIDS B1 か B2) を通過する．そのため，全てのトラフィックは移譲されたルールに対して下流 NIDS で検査される．

ケース 2 では，ケース 1 と同様に上流 NIDS の直下にあるマシンは全て NIDS であるが，ケース 1 とは逆にルールが下流 NIDS から上流 NIDS に移譲する．図 4.1

の例では，NIDS B2 から A にルールが移譲する場合である．この場合，組織外からの下流NIDS (NIDS B2) を通るトラフィックは，必ず上流NIDS (NIDS A) を通過し検査される．しかし，下流NIDS の下にあるサブネット間の通信については別に考慮しなければならない．例えば，ルール 3 が NIDS B2 から A に移譲すると，サブネット B2 と C 間のトラフィックは NIDS A では検査できない．このトラフィックを検査するために，下流 NIDS B2 でルール 3 を保持し，送信元・宛先 IP アドレスに基づきサブネット B2 と C 間のトラフィックに対してのみ，検査をする．通常組織外からのトラフィックは，サブネット間トラフィックに比べて多いと考えられるため，ルール 3 をサブネット間トラフィックに対して有効にしていても NIDS B2 のオフロードはできると考えられる．

ケース 3 では，NIDS の下流に通常のマシンと下流NIDS が共存しており，ルールが上流 NIDS から下流 NIDS に移譲する．図 4.1 の例では，サブネット B2 内のホストと NIDS C が NIDS B2 の直下に接続されており，ルールが NIDS B2 から C に移譲する場合である．この場合，ルール 3 が NIDS B2 から C に移譲されたときに，NIDS B2 でルール 3 を無効にすると，サブネット B2 内にあるマシン宛のトラフィックはチェックされなくなる．これを防ぐため，ルール 3 について上流 NIDS B2 でサブネット B2 宛のトラフィックをチェックする．NIDS B2 はサブネット C 宛のトラフィックはチェックしないため，ルール 3 が一部トラフィックについて有効となっていても，NIDS B2 のオフロードは可能だと考えられる．

ケース 4 では，ケース 3 と同様に NIDS の下流に通常のマシンと下流 NIDS が共存しているが，ルールが下流 NIDS から上流 NIDS に移譲する．例えば，NIDS C から B2 にルールを移譲した場合である．このケースは，ケース 2 と同様に扱うことができる．下流 NIDS C は NIDS C の下にあるマシン間のトラフィック（サブネット C 内のトラフィック）のみをチェックするためにルールを有効しておき，その他のトラフィックは NIDS B2 で検査する．

以上をまとめると，下流NIDS から上流NIDS にルールを移譲する際 (ケース 2 及び 4) には，移譲元の下流NIDS でおいてサブネット間のトラフィック検査を継続する必要がある．また，直下に NIDS でない通常のマシンが接続されている上流NIDS から下流 NIDS にルールを移譲する際 (ケース 3) には直下の通常マシン宛のトラフィック検査を継続する必要がある．このとき，検査をする必要のあるトラフィックの検出は，トラフィックの IP アドレスに基づいて行う．通常，NIDS の設定においては，組織内及び組織外の IP アドレスを指定する．例えば，Snort では，`HOME_NET` に監視対象となる組織内ネットワークの IP アドレスを，`EXTERNAL_NET` に組織

外ネットワークの IP アドレスを設定する．そのため，送信元・送信先 IP アドレスが共に HOME_NET 内にあれば，サブネット間トラフィックと判断できる．また，自 NIDS の HOME_NET から下流 NIDS の HOME_NET を除いたアドレスが直下にある通常マシンの IP アドレスとなるので，この IP アドレスと組織外ネットワーク EXTERNAL_NET 間がそれぞれ送信先・送信元 IP アドレスとなっていれば，直下の通常マシン宛のトラフィックと判断できる．現時点では IP アドレスに基づく検査は手動での設定変更で行うようになっているものの，上記で述べた組織内及び組織外の IP アドレスに基づいてルールを書き換えることで自動化は可能である．

冗長なルール設定を削除

冗長なルールの削除はシンプルである．Brownie は，自分が管理している NIDS と下流の NIDS の両方で有効にされているルールがあると，下流の NIDS でそのルールを無効にする．これによって，冗長なルールは全て自分が管理している NIDS でのみ有効となる．上流と下流の NIDS は，それぞれ有効・無効にしているルールを交換しているため，冗長なルールは容易に見つけることができる．また，冗長なルールを下流の NIDS で無効にすることにより，セキュリティが低下することがないのは，第 4.1.2 節で述べたとおりである．組織外からのトラフィックは，ルールを無効にした下流 NIDS に届く前に，ルールを有効にしてある上流 NIDS によってチェックされている．また，組織内トラフィックは，第 4.2.1 節で述べたのと同様に，組織内トラフィックのみを検査するようにしておけばよい．

冗長なルールを削除するために，下流 NIDS ではなく上流 NIDS のルールを無効にし，下流 NIDS のみで有効にする，という選択も可能である．しかしながら，下流 NIDS のみで有効にする場合，上流 NIDS のみで有効にする場合と比べて，セキュリティを保つための処理が複雑になる．上流 NIDS でルールを無効にすることによって，組織外からのトラフィックが検査されずに届くことになるサブネットができるためである．例えば，図 4.1 で，NIDS B2 でのみルール 1 を有効にし，NIDS A と C では無効にして，冗長なルールを削除したときを考える．冗長なルール 1 が削除される前は，サブネット B1 と B2 に向けたトラフィックを NIDS A において検査していた．しかし，NIDS A でルール 1 を無効にし，NIDS B2 のみでルール 1 を有効にすると，サブネット B1 へのトラフィックはルール 1 に対していずれの場所でも検査されない．ルール 1 に対して検査を行うためには，NIDS B2 でルール 1 を有効にしなければならない．一方，上流 NIDS でのみ有効にする場合，他の

NIDS でルールを有効にする必要はなく，よりシンプルな処理でできる．

なお，この方法では，全ての冗長なルールが上流NIDS でのみ有効となるが，上流 NIDS が突然過負荷になるということは考えにくい．なぜなら Brownie が動作していない場合でも，これらのルールは上流 NIDS で有効にされているからである．それでももし上流 NIDS の負荷が高くなった場合には，第 4.2.1 節で述べた手順にしたがって，上流 NIDS の負荷を下流 NIDS に移譲すればよい．下流 NIDS では，冗長なルールが無効とされたために負荷が軽減されている可能性が高く，上流 NIDS の負荷を引き受けやすくなっていると考えられる．

移譲元 NIDS におけるルール設定変更に対する対応

Brownie では，負荷状況などによってルールを自動的に移譲及び削除する．そのため，各 NIDS を管理している組織のポリシー変更等により，管理者がその NIDS のルール設定を変更しルールを追加･削除した場合には，Brownie は他の NIDS の設定状況によりルール設定を再調整する必要がある．そこで，Brownie では，各 NIDS の現在のルール設定の他に，Brownie による設定変更が行われる前の本来のルール設定を保持しておく．NIDS の管理者は本来のルール設定を変更を加え，Brownie は追加または削除されたルールについて，下流・上流 NIDS での本来及び現在の設定状況を考慮し，必要があれば再設定を行う．以下，ルールが追加･削除された場合について，それぞれの処理を述べる．ルールの変更については，変更前のルールが削除され，変更後のルールが追加されたとして扱う．

ルールが追加された場合の処理は，比較的シンプルである．下流または上流NIDS の現在のルール設定で追加されたルールが有効になっていた場合，冗長なルール設定となるため，4.1.2 節と同様の処理により冗長なルールの削除を行う．すなわち，下流 NIDS でルールが有効になっていた場合には下流 NIDS で，上流 NIDS でルールが有効になっていた場合には変更された NIDS で，当該ルールを無効にする．下流及び上流 NIDS の双方で，該当ルールが現在無効になっている場合には，そのまま当該ルールを有効にすればよい．ルールが追加されたことにより，負荷状況が大きく変化した場合には，4.1.1 節で述べたオフロード手順に従って，ルールの移譲がおこる可能性がある．

一方，ルールが削除された場合には，上流・下流 NIDS における本来の及び現在の設定状況を考慮して，ルール削除を反映する必要がある．表 4.2 に各設定状況における変更内容を示す．各設定状況において，上段にルールが削除された NIDS の

表 4.2: ルールが削除された場合の処理：各設定において，上段にルールを削除した NIDS の実際の設定変更を，下段に上流・下流の NIDS の設定変更を示す.

		上流・下流 NIDS の現在の設定	
		有効	無効
上流・下流 NIDS の本来の設定	無効	変更無し（無効）	有効 → 無効
		変更無し（有効）	無効 → 有効
	有効	変更無し（無効）	有効 → 無効
		有効 → 無効	変更無し（無効）

設定変更を，下段に上流・下流の NIDS の設定変更を示している．例えば，右上は，上流・下流 NIDS の本来の設定では有効になっており，現在の設定では無効になっている場合である．このとき，ルール変更が行われた NIDS での変更前の設定は，本来の設定は有効（削除するためには，有効になっている必要がある），実際の設定でも有効（上流・下流 NIDS では無効であるため）となっている．このルールが削除された場合，変更された NIDS でこのルールを無効にし，上流・下流 NIDS で有効にすることで，本来の設定と矛盾のない設定となる．設定内容が他の場合も同様に，表 4.2 の通りに処理することでルール削除に対応することができる．

4.2.2 耐障害性向上

障害が起こった NIDS で有効にしていたルールを別の NIDS で有効にすることで攻撃検知を肩代わりする．障害の検知は，定期的に送り合っている負荷情報をハートビートとして用いることで行う．これは第 4.2.1 節で述べたもので，過負荷 NIDS の検出および負荷分散のために送り合っているものである．一定時間ハートビートが送信されて来なければ，何らかの原因によりその NIDS に障害が起こったと判断する．そして，障害が起こった NIDS が有効にしていたルールを，自分が管理している NIDS で有効にする．各 NIDS は，自分の上流・下流の NIDS のルール設定を交換しているため，障害が起こった NIDS でどのルールを有効にしていたかは容易に知ることができる．

障害の検知速度は，ハートビートの間隔と，障害と判断するまでにハートビートを待つ時間によって決まる．本手法では，定期的に交換している負荷情報をハートビートとして用いているため，負荷情報の交換間隔がハートビートの間隔となる．第 4.2.1 節で述べたとおり，実験ではこの間隔を 30 秒と設定した．障害検知速度により影響するのは，ハートビートを待つ時間である．この時間が短いほど

障害を即座に検知することができるが，短すぎるとハートビートが少し遅延しただけで障害と誤検知する可能性がある．NIDS やネットワークの負荷により，ハートビートが遅れる可能性は十分考えられる．ハートビートを待つ時間はこれらを考慮して設定する必要があり，実験では 60 秒とした．なお，ハートビートは負荷情報とは別に送り合うことも可能である．この場合，ハートビートのための通信を別にしなければならないため通信のオーバヘッドが増加するが，ハートビートの間隔を負荷分散の間隔とは別に設定できるというメリットがある．

障害が起こった NIDS が再び動作した時は，まず初期動作時と同様にルール設定等を互いに交換する．その後，障害が起こった NIDS の回復だと判明したら，有効にしておいたルールを再び無効に戻す．むろん，障害発生前後のルールやマシンの設定が全く同じとは限らないため，障害前のルール設定も参照して一部のルールを有効にしたままにする場合や，負荷分散や冗長ルールの削除を再び行う場合もある．

4.2.3 警告ログの収集

Brownie は自動的にルールを有効・無効にするため，元々ルールを有効にしていた NIDS とは別の NIDS で攻撃検知が警告されることがある．そこで，元来ルールを有効にしていた NIDS の管理者に別の NIDS で出た警告を知らせるため，Brownie ではどのルールがどの NIDS から NIDS に移譲されたかを記憶しておく．NIDS が警告を出したら，その NIDS を管理している Brownie は警告を元々ルールを有効にしていた NIDS の Brownie に転送する．警告を受け取った Brownie は，NIDS の代わりに警告を出す．例えば，NIDS の警告ログに警告を書き出す．これによって，管理者は実際にはその NIDS で有効にされていない警告についても，警告に気付くことができる．

4.3 実験

4.3.1 実験環境

Brownie により，性能や耐障害性が向上することを示すため，実験を行った．3 台の NIDS，2 台のクライアント，2 台のサーバの計 7 台のマシンを用い，図 4.2 に示すように 1Gbps イーサネットで接続した．上流 NIDS マシンは，2 つの Intel

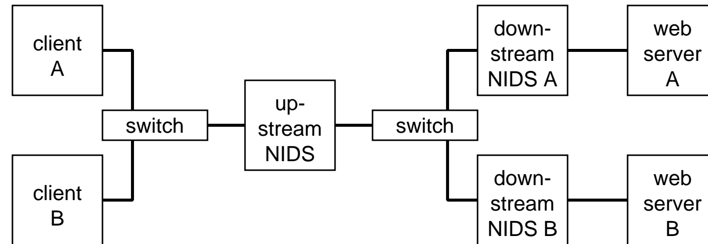


図 4.2: 実験環境

Dual-Core Xeon 2.33GHz CPU（1 コアのみ有効），2GB メモリ，250GB 7200rpm HDD で構成され，他のマシンは，Pentium 4 2.8GHz CPU，512MB メモリ，36GB 7200rpm HDD で構成されている．これは，下流 NIDS より上流 NIDS の方が高性能な構成となっている．オペレーティングシステムには Fedora 8 (Linux 2.6.24)，web サーバには Apache2.2.8 を使用した．NIDS としては，Snort 2.8.0.1 [11] を inline モードで用い，ルールセットは 2008 年 1 月 28 日現在のものをデフォルトのまま用いた．

4.3.2 性能向上：ベンチマーク

Brownie による性能向上を示すため，web サーバベンチマークを用いた実験を行った．上下流 NIDS のルールの初期設定として，1) 上流 NIDS で全て無効，下流 NIDS でデフォルト全て有効という設定 (DOWN) と，2) 上下流双方の NIDS でデフォルト全て有効という設定 (BOTH)，の 2 つの設定で実験を行った．前者は過負荷 NIDS をオフロードすることによる性能向上，後者は冗長ルールを削除することによる性能向上を測定することを目的とした初期設定である．実験では，各 NIDS での有効ルール数，CPU 使用率，及びベンチマークのスループットを 10 秒ごとに計測した．最初の 30 分間は，初期状態の性能を測定するため Brownie は動作させないようにした．ワークロードには WebStone2.5 を各クライアント上で実行し，それぞれ同時接続数は 10 とした．Apache，WebStone，Snort のその他の設定はデフォルトのままとした．

実験結果：過負荷 NIDS の負荷軽減の効果

過負荷 NIDS の負荷を削減することによる効果を示すため，NIDS の初期ルール設定を，下流 NIDS はデフォルトルールを全て有効 (ルール数 8676)，上流 NIDS

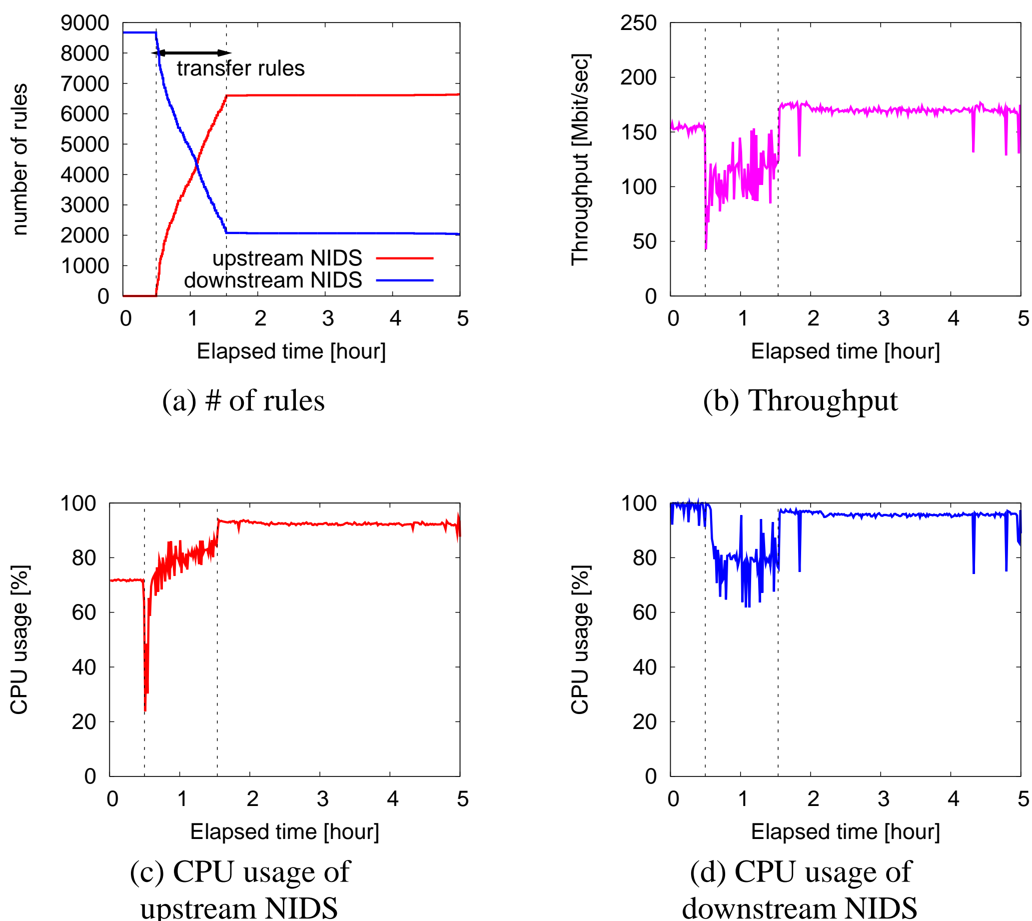


図 4.3: 実験結果：過負荷 NIDS の負荷軽減 (ベンチマーク，初期設定: DOWN)

は全てのルールを無効とした．この設定では，下流 NIDS が過負荷となり，ネットワーク全体のボトルネックとなる．

図 4.3 に実験結果を示す．グラフ内の 2 つの縦線は，それぞれ Brownie がルール移譲を開始及び終了した時刻を示す．図 4.3 (a) は，各 NIDS で有効にしているルール数を示している．開始 30 分後から Brownie が動作し始め，負荷分散のためのルールの移譲が開始する．下流 NIDS のルール数が減少し，上流 NIDS のルール数が増加していることがわかる．図 4.3 (c) と (d) に，それぞれ上流 NIDS と下流 NIDS の CPU 使用率を示す．2 つの下流 NIDS の CPU 使用率はほぼ同じであるため，片方のみを載せた．初期ルール設定では，下流 NIDS の CPU 使用率がほぼ 100% に達している一方，上流 NIDS の CPU 使用率は 80% 未満となっている．Brownie 動作開始後約 1 時間で，双方の CPU 使用率はほぼ同じとなり，Brownie はルール移譲を停止した．その後の CPU 使用率は全ての NIDS で 90〜95% となり，負荷が分散

されたことがわかる．

なお，この実験では4.1.1.3節で述べたIPアドレス制限つきのルール移譲については考慮せず，移譲元では移譲したルールを完全に無効としているが，IPアドレス制限つきのルール移譲を行った場合には，移譲時間が少し長くなる可能性がある．一部のトラフィックに対しての監視を続けることで，完全に無効にしたときに比べて，CPU使用率の低下が小さい可能性があるからである．その結果，より多くのルールを移譲する必要が出てくる可能性がある．しかし，5.2.3節で示すように，トラフィックの半分について監視を続けても十分に負荷を下げることができる．通常サブネット間トラフィック等IPアドレス制限での監視をするトラフィックは，その他のトラフィックに比べて少ないため，このトラフィックに対する監視の負荷は小さく，大きな影響はないと考えられる．

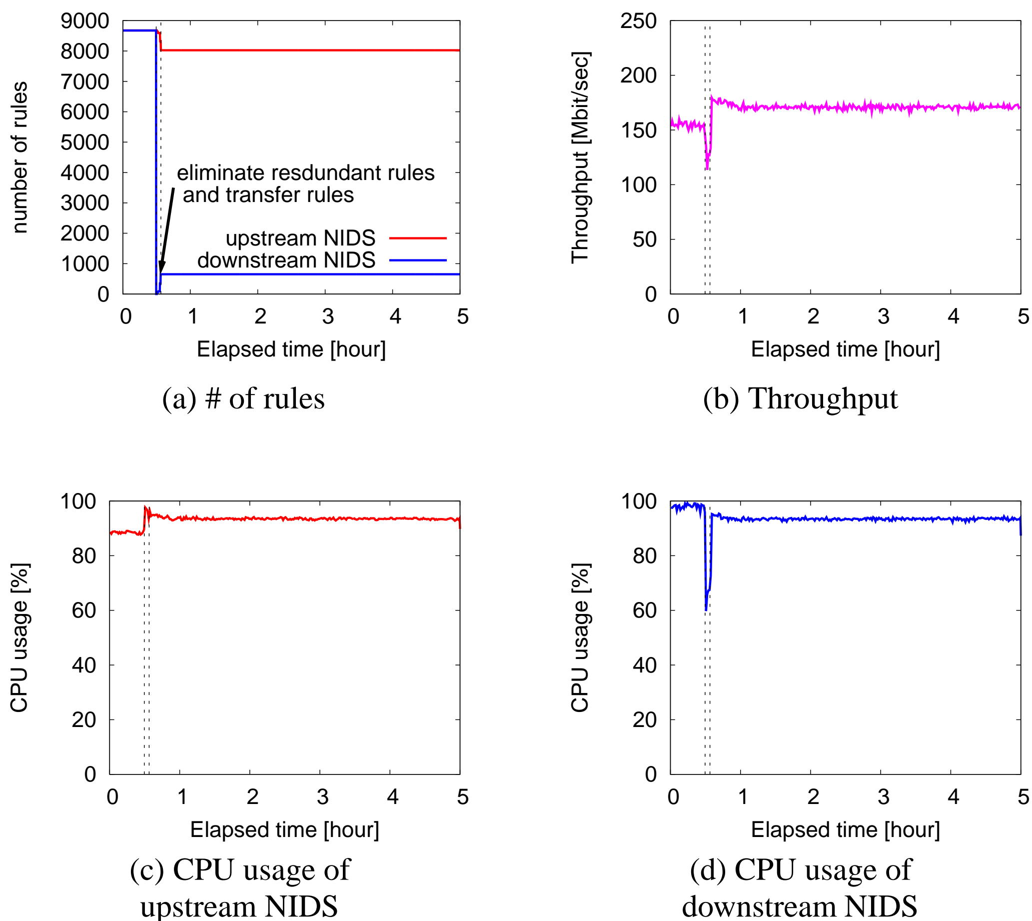
図4.3 (b)にwebサーバベンチマークのスループットを示す．初期設定でのスループットは154 Mbit/sec，負荷分散後のスループットは174 Mbit/secとなり，13%増加した．現在の実装ではルールの再設定のためにNIDSを再起動させなければならないため，ルール移譲中スループットは一時的に減少する．これは，Snortの設定再ロード処理を変更することで再ロード時間を20%削減したElephant [85]を用いることで改善できると考えられる．

ルール移譲後のルール数は，上流NIDSで6600，下流NIDSで2076となった．上流NIDSは下流NIDSより高性能であるため，負荷が同程度であるとき，より多くのルールが上流NIDSで有効となっている．

実験結果：重複ルールの削除の効果

重複ルールの削除による性能向上の効果を示すため，NIDSの初期ルール設定を，上流・下流双方ともデフォルトルールを全て有効として，実験を行った．全てのNIDSが同じルール設定となっているため，トラフィックは必ず同じルールに対して2回チェックされる．

図4.4に実験結果を示す．図4.4 (a)に各NIDSでの有効ルール数の推移を示す．実験開始30分後にBrownieが動作を開始すると，まず冗長ルールを全て削除する．初期設定BOTHでは，全てのデフォルトルールが全てのNIDSで有効にされているため，下流NIDSのルール全てが無効にされる．その結果，全てのルールが上流NIDSでのみ有効となる．その後，一部のルールが上流NIDSから下流NIDSに移譲され，約5分でオフロードが終了した．図4.4 (b)に示すように，ベンチマー



(a) # of rules

(b) Throughput

(c) CPU usage of upstream NIDS

(d) CPU usage of downstream NIDS

図 4.4: 実験結果：冗長ルール削除 (ベンチマーク，初期設定: BOTH)

クスループットは，初期設定時 155 Mbit/sec から 173 Mbit/sec に，12%増加した．この性能向上は，同じルールに対して複数検査する必要がなくなったためであると考えられる．

図 4.4 (c) と (d) に，それぞれ上流と下流 NIDS の CPU 使用率を示す．初期設定では，全てのルールが有効となっているため，下流 NIDS の CPU 使用率は 100%近くになっている．Brownie が冗長ルールを削除すると，下流 NIDS の CPU 使用率は下がり，上流・下流 NIDS で同程度の CPU 使用率となった．

IP アドレス制限つき移譲

ルールが IP アドレス制限つきで移譲元NIDS に残った場合においても，NIDS の負荷が軽減することを示すため，全てのトラフィックを検査した場合と一部のトラ

フィックを検査した場合とを比較する実験を行った．ここでは，4.1.1.3 節で述べたうち，NIDS の下流に通常マシンと下流 NIDS が共存している場合に，上流 NIDS から下流 NIDS にルールが移譲される場合 (ケース 3) を想定した．この場合，上流 NIDS には，ルールが IP アドレス制限つきで残ることとなる．

実験では，図 2 において，下流 NIDS B のマシンが NIDS 機能を持たない場合を想定し，上流 NIDS の CPU 使用率とスループットを計測した．下流 NIDS B が存在しないため，上流 NIDS から下流 NIDS A にルールを移譲した場合，上流 NIDS に web サーバ B 向けのトラフィックを検査するためにルールが残ることになる．全てのトラフィックを検査する場合の設定では，上流 NIDS で全てのルールを有効にし，全てのトラフィック（web サーバ A とクライアント A 間のトラフィック及び web サーバ B とクライアント B 間のトラフィック）を検査する．一方，一部のトラフィックを検査する場合の設定では，下流 NIDS A に全てのルールを移譲し，web サーバ B とクライアント B 間のトラフィックのみを検査する．web サーバ A とクライアント A 間のトラフィックは，下流 NIDS A で検査するため，検査しなければならないトラフィックは半分となる．

実験の結果，CPU 使用率は，全てのトラフィックを検査する場合で 100%，半分のトラフィックを検査する場合で 80% となった．また，スループットは，全てのトラフィックを検査する場合で 186 Mbit/sec，半分のトラフィックを検査する場合で 263 MBit/sec となった．検査するトラフィックが半分になったことにより，CPU 使用率は 20 減少し，スループットは 41% 増加した．

4.3.3　性能向上：実トラフィック

実ネットワーク・トラフィック下での Brownie による性能向上を示すため，Brownie に対してネットワーク・キャプチャしたトラフィックを流す実験を行った．トラフィックは，豊橋技術科学大学（/16 ネットワーク）のネットワーク・エントリ・ポイントにおいて，2008 年 3 月 23 日から 4 日間のネットワークトラフィックを収集したものを用いた．収集したデータは，データ量 673GB で，220,919,190 個の内向きパケットと，168,454,019 個の外向きパケットからなっている．

実験では，3 台の NIDS と，収集したパケットを送受信する 1 台のマシンの計 4 台のマシンを用いた．NIDS は図 4.2 で示した図と同様に接続し，パケットを送受信するマシンを全ての NIDS に接続した．全てのマシンは，1Gbps イーサネットで接続している．上流 NIDS は Intel Quad-Core Xeon 2.33GHz CPU，4GB メモリ，下

流 NIDS は Intel Pentium Dual-Core 2GHz CPU，1GB メモリ，パケット送受信マシンは Intel Core2 Duo 2.4GHz CPU，2GB メモリで構成されている．ソフトウェアの設定は，第 4.3.2 節と同様である．キャプチャしたパケットの送信は，`tcpreplay` を用い，NIDS に負荷をかけるため，再生時間間隔を無視したトップスピードで送信した．

上下流の NIDS のルールの初期設定は，1) 上流 NIDS でデフォルト全て有効，下流 NIDS で全て無効という設定 (UP) と，2) 上下流双方の NIDS でデフォルト全て有効という設定 (BOTH)，の 2 つの設定で実験を行った．ベンチマークによる実験と同様に，前者は過負荷 NIDS をオフロードすることによる性能向上，後者は冗長ルールを削除することによる性能向上を測定することを目的とした初期設定である．

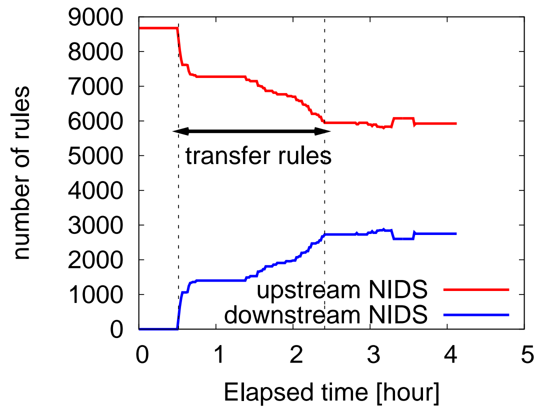
この実験では，各 NIDS の有効ルール数と CPU 使用率を計測した．キャプチャしたパケットを送信する実験であるため，スループットや応答時間は測定ができない．ただし，比較のために Brownie がない状態での CPU 使用率を測定した．Brownie がない場合，全ての NIDS において，ルール設定は初期設定のままとなる．

実験結果：過負荷 NIDS の負荷軽減の効果

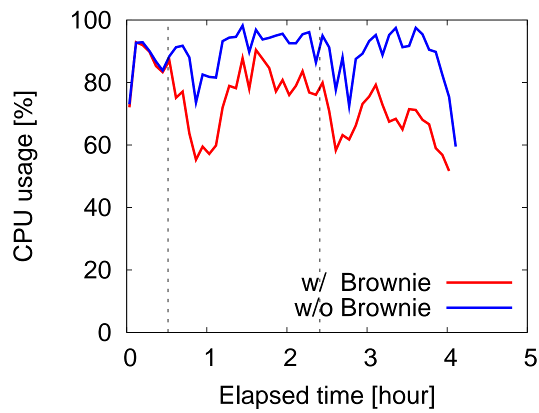
過負荷 NIDS の負荷を軽減することによる効果を示すため，NIDS の初期ルール設定を，上流 NIDS はデフォルトルールを全て有効 (ルール数 8676)，下流 NIDS は全てのルールを無効とした．この設定では，上流 NIDS が過負荷となり，ネットワーク全体のボトルネックとなる．なお，実験マシンの性能がベンチマークを利用した実験とは異なることから，どちらかの NIDS を過負荷にするために，NIDS の初期ルール設定も第 4.3.2 節とは異なる設定となっている．

図 4.5 に実験結果を示す．図 4.5 (a) は，Brownie がある場合の各 NIDS で有効にしているルール数の推移を示している．開始 30 分後から Brownie が動作し始め，負荷分散のためのルールの移譲を開始する．上流 NIDS のルールが減少し，下流 NIDS のルールが増加していることが分かる．Brownie 動作開始後約 1.9 時間で，ルール移譲が停止した．

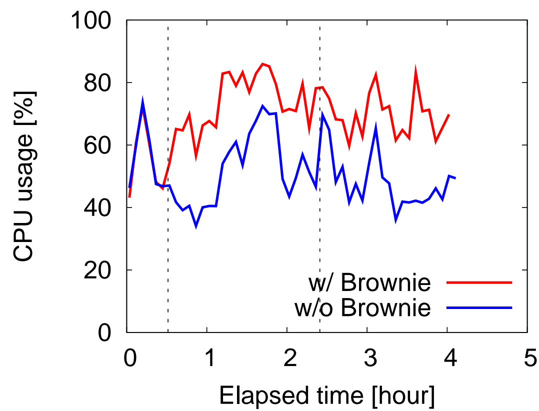
図 4.5 (b) と (c) にそれぞれ上流 NIDS と下流 NIDS での 5 分ごとの CPU 使用率を示す．Brownie なしの場合，全てのルールが上流 NIDS で有効にされたままのため，上流 NIDS の CPU 使用率は 100% になる場合がある．一方，Brownie ありの場合，ルールを移譲し始めるとすぐに，上流 NIDS の CPU 使用率は減少し，下流 NIDS



(a) # of rules with Brownie



(b) CPU usage of upstream NIDS



(c) CPU usage of downstream NIDS

図 4.5: 実験結果：過負荷 NIDS の負荷軽減 (実トラフィック，初期設定: UP)

の CPU 使用率が増加した．ルール移譲が停止すると，全ての NIDS の CPU 使用率はほぼ同じ程度となった．

実験結果：重複ルールの削除の効果

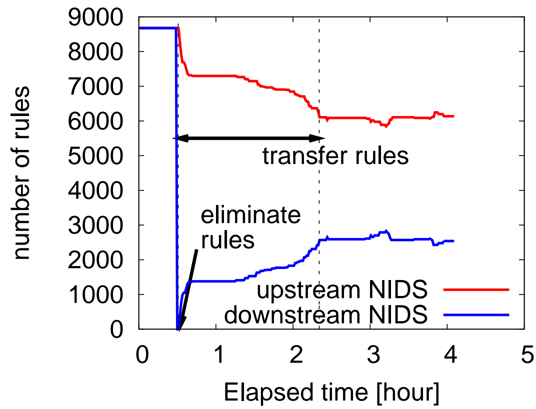
重複ルールの削除による性能向上の効果を示すため，NIDS の初期ルール設定を，上流・下流双方ともデフォルトルールを全て有効として，実験を行った．全ての NIDS が同じルール設定となっているため，トラフィックは必ず同じルールに対して 2 回チェックされる．

図 4.6 に実験結果を示す．図 4.6 (a) に各 NIDS での有効ルール数の水位を示す．この図が示すように，実験開始 30 分後に Brownie が動作を開始すると，まず冗長ルールを全て削除する．その後，一部のルールが上流 NIDS から下流 NIDS に移譲される．冗長ルールを削除したあとの設定は，第 4.3.3 節での初期設定 UP と同一であるため，この移譲は第 4.3.3 節とほぼ同様となる．図 4.6 (b) と (c) にそれぞれ上流 NIDS と下流 NIDS の CPU 使用率を示す．Brownie なしの場合，上流 NIDS の CPU 使用率は 100% になる場合がある．一方，Brownie ありの場合，ルールを移譲するにつれて，上流 NIDS の CPU 使用率は減少する．また，下流 NIDS の CPU 使用率は，Brownie なしの場合とほとんど変化はなかった．

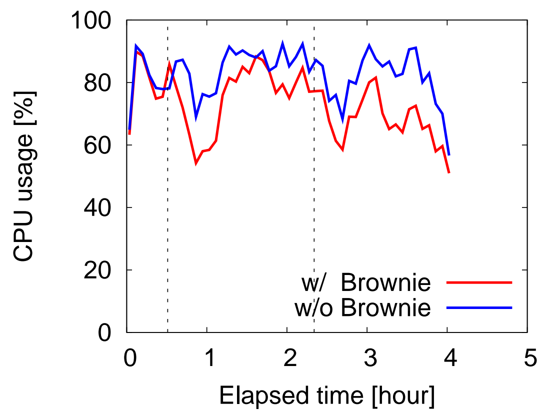
4.3.4 耐障害性向上

Brownie による耐障害性向上を示すため，攻撃生成ツールと web サーバベンチマークを用いた実験を行った．ルールの初期設定として，第 4.3.2 節でのオフロード終了後のルール設定 (BALANCED: ルール数は上流 NIDS が 6600，下流 NIDS が 2076) とした．攻撃生成ツールと web サーバベンチマークを動作させ，開始 5 分後に 1 つの NIDS を停止させた時の，有効ルール数，警告数，ベンチマークスループットを測定した．停止させる NIDS は上流 NIDS を停止させる場合と，下流 NIDS を停止させる場合の実験とした．攻撃生成ツールとして，Nikto2.1.2 [86] を用いた．

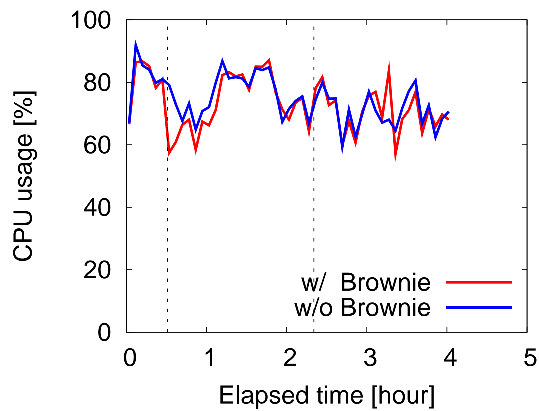
初期設定を負荷分散された後のルール設定で，上流 NIDS を停止させた時の実験結果を図 4.7 に示す．図 4.7 (a) に示すように，実験開始 5 分後に上流 NIDS が停止したため，上流 NIDS の有効ルール数が実質 0 になった．その後 100 秒後に障害を検知し，上流 NIDS で有効にされていたルールを下流 NIDS で有効にしたことにより，下流 NIDS での有効ルール数が増加した．このときの警告の数を図 4.7 (b)



(a) # of rules with Brownie



(b) CPU usage of upstream NIDS



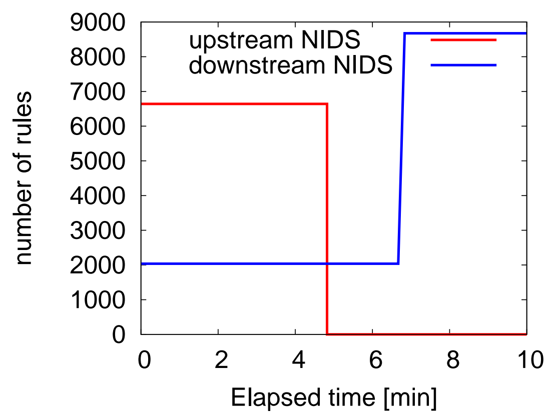
(c) CPU usage of downstream NIDS

図 4.6: 実験結果：冗長ルール削除 (実トラフィック，初期設定: BOTH)

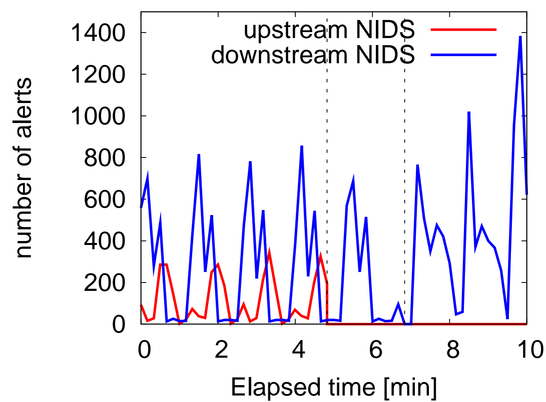
に示す．縦線は，1つ目が上流NIDSが停止した時刻，2つめが下流NIDSがルール設定を変更した時刻を表す．実験開始時は上流・下流NIDS双方で攻撃を検知し警告を発しているが，上流NIDSを停止させると下流NIDSのみで警告が発生するようになる．停止後しばらくの間は下流NIDSで発生する警告数は変化しないが，障害を検知し上流NIDSでのルールも有効にすると，警告数が増加し，上流NIDSでの障害をカバーできていることがわかる．また，ベンチマークスループットを図4.7 (c) に示す．前節の実験でも示した通り，デフォルトルールを全て有効にすることは，下流NIDSには過負荷となるため，障害発生前に比べてスループットは減少している．

初期設定を同じく負荷分散した後のルール設定で，下流NIDSを停止させた時の実験結果は，図4.8の通りである．上記の場合と下流NIDSと下流NIDSが逆になった以外は大きな違いはない．なお，下流NIDSが停止した後のスループットが，停止する前のスループットよりも増加している原因として，下流NIDSではSnortが停止しているためと考えられる．NIDSでSnortが稼働している場合，パケットを検査するためにユーザレベルに一度上げる必要があるが，NIDSが停止している場合そのまま転送するからである．

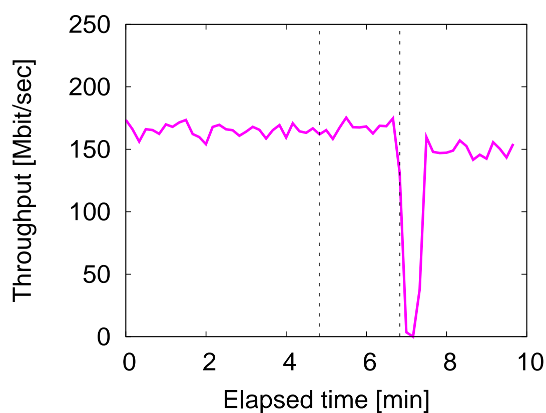
この実験では，ハートビート30秒，障害検知までのハートビート待機時間を60秒としたため，障害発生からルール再設定まで1分半程度要した．この期間，障害が発生したNIDSで有効になっていたルールに対する攻撃は検知されない．ハートビートの間隔及びハートビート待機時間を短くすることで，この期間を短くすることができる．もちろん，完全冗長のルール設定を行っていれば，この期間はない．しかし，第4.3.2節の実験で示した通り，完全冗長は性能が低下する．また，今回は性能をスループットで計測するためにNIPSとしてinline-modeで動作させたが，NIDSの場合過負荷になるとパケットの取りこぼしがおきるため，ルールを有効にしていても攻撃を検知できない可能性がある．また，Brownieがない場合は，障害が起こっていないNIDSが元々有効にしていたルールのみを検知する期間が非常に長くなる．これは，例えば管理者がNIDSの障害に気づき対処するまで，という時間となり，Brownieによる障害検知及び他NIDSによる肩代わりよりも長い時間になることが予想される．



(a) # of rules

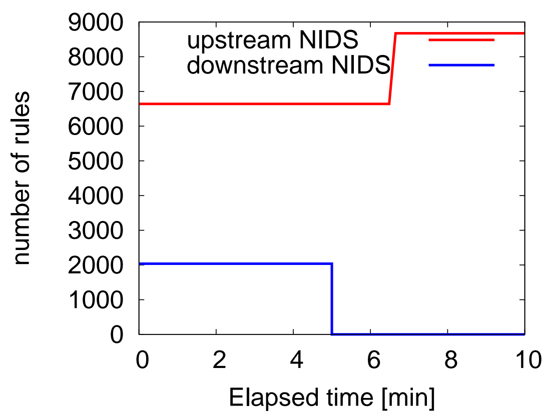


(b) # of alerts

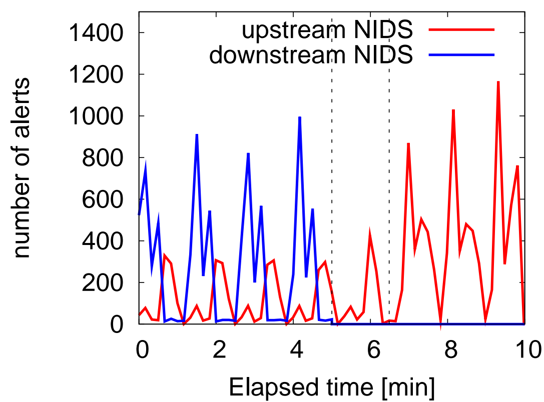


(c) Throughput

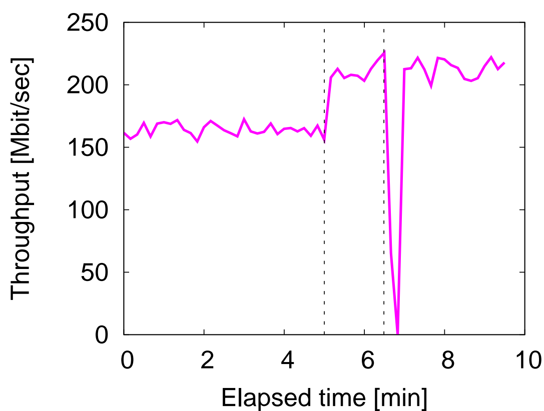
図 4.7: 実験結果：耐障害性 (初期設定: BALANCED，上流 NIDS を停止)



(a) # of rules



(b) # of alerts



(c) Throughput

図 4.8: 実験結果：耐障害性 (初期設定: BALANCED，下流 NIDS を停止)

4.4 まとめ

ネットワーク経由の攻撃の検知及び防御にネットワーク侵入検知・防御システム (NIDS/NIPS) が広く用いられているが，汎用 PC を用いた NIDS が多い中，ネットワークの高速化や攻撃の巧妙化に伴う性能低下や，NIDS 障害によって攻撃を見逃す耐障害性の問題がある．本論文では，組織内に複数設置された NIDS を相互に協調させることで，性能向上及び耐障害性向上をする手法を提案する．1 カ所に高価なマシンや並列に動作する複数のマシンを置くのではなく，他の NIDS の負荷やルール設定を元に，通常時は過負荷のマシンや冗長なルール設定を減らすように再設定をすることで性能向上を，障害時は他の NIDS が検査を肩代わりすることで耐障害性向上を目指す．また，性能向上と耐障害性はトレードオフの関係となるため，管理者がこれらを柔軟に選択できるようにした．実験では，冗長なルール設定が除去され負荷が分散されることで性能向上を，障害時に攻撃検知を他の NIDS が行うことで耐障害性向上ができることを確認した．

今後は，NIDS を協調させることによる他の応用を検討する予定である．例えば，ルールをある NIDS に集めることで，NIDS の省電力化ができると考えられる．上流 NIDS のみで処理できる程度の負荷であれば，全てのルールを上流 NIDS で処理し，下流 NIDS を止めることで，下流 NIDS の電力を削減できると考えている．

第5章　結論

5.1　本研究のまとめ

インターネットが我々の生活に不可欠な社会基盤となっている一方，インターネット上のサーバを狙ったリモート攻撃は後を絶たない．ネットワークを介したリモート攻撃を検出・防止する手段の一つとして，ネットワーク侵入検知・防御システム（NIDS/NIPS: Network Intrusion Detection/Prevention Systems）が広く利用されている．リモート攻撃とは，脆弱性のあるサーバに対して，インターネットを通じて攻撃メッセージを送り，被害を発生させる攻撃である．NIDS/NIPS はサーバに送信されてくるメッセージを検査することによって攻撃を検知し，管理者に警告を発することで，サーバを攻撃から防御する．

リモート攻撃が巧妙になりインターネット上のトラフィックが増大するなど，NIDS を取り巻く環境の変化により，3 つの課題が出てきている．第一に，攻撃検知の精度向上が求められている．攻撃が高度化・巧妙化したことにより，従来の単純なシグネチャ・マッチングでは検知できない攻撃が増えているためである．第二に，NIDS の性能向上が求められている．インターネット・トラフィックの増大や検知手法の高度化に伴い，NIDS の負荷が増加しているためである．第三に，NIDS の障害発生時にも継続して攻撃検知が可能となる，耐障害性が求められている．汎用 PC を元にした構成が多い NIDS が，ハードウェア故障などにより NIDS に障害が起こった場合にも，継続して攻撃検知が行える必要がある．

本論文では，NIDS/NIPS の実装技術として，検知精度・性能向上・耐障害性向上を解決する手法を提案した．まず，性能を低下させずに検知精度を向上するために，レイヤ 7 NIDS/NIPS のための TCP ストリーム再構成機構の方式として，store-through 方式を提案した．近年，単純なバイトパターンのマッチングでなく，メッセージの順番やフォーマットなどのレイヤ 7 コンテキストを考慮することにより検知精度を高める NIDS が提案されている．レイヤ 7 コンテキストを考慮した攻撃検知を行うためには，ネットワーク上を流れる個別のパケットをメッセージに

再構成する，TCP ストリーム再構成機構が必要となる．レイヤ 7 NIDS/NIPS のための TCP ストリーム再構成機構は次の 4 つの要件を満たさなければならない．すなわち，1) 攻撃メッセージが攻撃対象アプリケーションに届くことがない完全な防御，2) 性能低下が少ないこと，3) NIDS の設置に伴って，サーバやクライアントのアプリケーションに変更や再設定が必要ないアプリケーション透過性，4) 監視する通信の振る舞いを乱さない，すなわち TCP フローや輻輳制御に与える影響が少ないトランスポート透過性，の 4 つである．本論文では，これらを全て満たす TCP ストリーム再構成機構として，store-through 方式を提案した．Store-through 方式では，個別のパケットからメッセージに再構築する際，順番が入れ替わったパケットは，止めずにコピーをとって転送することで，トランスポート透過性を保持する．また，攻撃だと判断された時点で，後から到着したパケットを破棄することで，攻撃の成功を防ぎ，完全な防御を達成する．また，IP レベルで実装することで性能とアプリケーション透過性を達成する．プロトタイプを Linux 2.4.30 上に実装し，実験によって store-through 方式によるオーバヘッドは 3.8% 以下であることを示した．また，実際のネットワークを用いた実験により，トランスポート透過性を保持できていることを示した．

次に，組織ネットワーク内に複数の場所に置かれた NIDS 同士を協調させることで，性能向上と耐障害性向上を行う手法を提案した．大学や企業など多くの組織では，組織内ネットワークとインターネットの境界のみでなく，内側のネットワークの様々な階層に複数の NIDS を設置している場合が多い．本論文で提案する NIDS 協調システム Brownie では，これらの組織ネットワーク内に置かれた NIDS 間でルール設定を交換し合い，定期的に負荷情報をやりとりすることで，NIDS 同士のルール設定を連携させる．そして，性能を向上するため，過負荷になった NIDS の負荷を減少させ，NIDS 間の冗長なルール設定を削除するようにルールを再設定する．また，耐障害性を向上するため，NIDS の障害時には，障害が起こった NIDS で有効にしていたルールを別の NIDS で有効にして攻撃検知を代替えする．Brownie のプロトタイプを実装し，実験によって，ルールを再設定することで，web サーバベンチマークのスループットが 10% 以上向上することを示した．また，NIDS を意図的に停止させ障害を起こした実験では，100 秒程度で別の NIDS でルールが有効になり攻撃が検知されるようになることを示した．

5.2 今後の展望

本研究によって，NIDS の現在の課題である検知精度・性能・耐障害性に対する 2 つの手法を提案した．本節では，本研究の今後の展望について述べる．

レイヤ 7 コンテキストによる攻撃検知は，研究段階から実際の製品でも採用されるようになってきており，今後多くの NIDS は標準的にレイヤ 7 コンテキストまでを考慮した攻撃検知を行うようになると考えられる．すなわち，NIDS 協調システム Brownie が協調させる NIDS もレイヤ 7 NIDS である可能性が高くなってくる．パケットレベルの NIDS は状態を持たないため，ルールの再配置を自由におこなうことができる．しかし，メッセージを監視するレイヤ 7 NIDS では，ルールを移譲する際は，メッセージのどの部分までを検査したか状態も含めて移譲しなければならない．センサの状態のみでなく，TCP ストリーム再構成機構で保持している，順番通りでないパケットのコピーも同様である．

また，レイヤ 7 NIDS の協調においては，状態の考慮という課題だけでなく，TCP ストリーム再構成機構を考慮したルールの配置を行うことが可能であると考えられる．レイヤ 7 NIDS では，複数のルールが同じメッセージの再構成を必要とする場合がある．移譲するルールを選ぶ場合には，なるべく同じメッセージに対する検査が 1 つの NIDS に集まるようにすると良いと考えられる．すなわち，本研究で提案したルールの重複を削除するだけではなく，メッセージの再構成の重複もできるだけ少なくするようにルールの再配置を行うことで TCP ストリーム再構築の重複がなくなり，性能向上を行うことが可能となると考えられる．

謝辞

本論文は著者が慶應義塾大学大学院理工学研究科開放環境科学専攻の後期博士課程に在籍中の研究成果をまとめたものです．本研究を行うにあたって，また本論文をまとめるにあたって，多くの方々からご指導とご協力を賜りました．お世話になった全ての方々にこの場を借りて御礼申し上げます．

まず，本論文の主査であり，著者の指導教員である慶應義塾大学理工学部情報工学科 河野健二准教授に深く感謝いたします．河野健二准教授には，著者が電気通信大学学部 4 年生時から，慶應義塾大学大学院修士課程，慶應義塾大学大学院後期博士課程，そして所定単位取得満期退学後本論文をまとめる 2 年間まで，8 年間もの長きにわたって，研究活動の進め方から，論文のまとめ方，口頭発表におけるプレゼンテーションの要点，そして研究活動の面白さをご教示いただきました．その親身なご指導は大変ありがたいものでした．今現在研究活動に従事できているのは，ひとえに河野健二准教授と出会えたからであると思っております．また高度な研究活動を可能とする，国内有数の潤沢な研究設備を利用させていただきました．心より感謝いたします．

次に，本論文の副査を担当していただいた，慶應義塾大学理工学部情報工学科 寺岡文男教授，重野寛准教授，システムデザイン工学科 西宏章准教授に感謝いたします．副査の皆様には貴重なお時間を割いていただき，本論文を丁寧に査読していただきました．副査の皆様からいただいたコメントと，皆様との有意義な議論によって，本論文の完成度が向上したと実感しております．深く感謝いたします．

電気通信大学情報理工学部情報・通信工学科 岩崎英哉教授には，著者が電気通信大学電気通信学部情報工学科ならびに電気通信大学大学院電気通信学研究科情報工学専攻に在籍していた際，大変お世話になりました．岩崎英哉教授には初めて研究活動というものに触れる大切な時期に，研究を進める上で重要な助言をいくつもいただきました．また，本論文で行った実験に必要な環境をご提供いただきました．心より感謝いたします．

法政大学情報科学部コンピュータ科学科 廣津登志夫教授ならびに大阪大学サイ

バーメディアセンター 阿部洋丈助教には，本研究に必要な実験を行う環境をご提供いただきました．また，実験方法や論文に関しても丁寧なご指導をいただきました．感謝いたします．また，豊橋技術科学大学情報工学系廣津研究室の皆様には，実験のために豊橋技術科学大学に訪問した際，大変お世話になりました．感謝いたします．

筑波大学システム情報工学研究科コンピュータサイエンス専攻 杉木章義助教ならびに慶應義塾大学理工学部情報工学科 山田浩史特任助教に感謝いたします．両氏には，著者が電気通信大学在籍時から，著者の研究活動に対し多くの有意義な助言をいただきました．また，両氏の研究に取り組む姿勢に多くのことを学ばせていただきました．深く感謝いたします．

慶應義塾大学理工学部情報工学科 河野研究室の皆様ならびに電気通信大学情報理工学部情報・通信工学科 岩崎研究室の皆様に感謝いたします．特に共著論文を執筆した小菅祐史氏，田上歩氏，横山敏博氏，北川貴久氏，石川豊氏には，共に刺激しあうことで，著者の成長を助けていただきました．優秀な後輩に囲まれて過ごすことができたのは大変幸運であったと感じております．深く感謝いたします．また，慶應義塾大学大学院へ共に入学した浅原理人氏，嶋村誠氏，宮地大輝氏，山口聖司氏には，勝手分からぬ新しい学舎での講義受講や研究活動の際に，多くの助けをいただきました．感謝いたします．

日立製作所中央研究所 助川直伸氏，西澤格氏，川本真一氏，ならびに日立製作所ソフトウェア事業部 松林忠孝氏に感謝いたします．仕事のかたわら本論文をまとめることについてご理解いただき，応援していただきました．皆様のご支援に深く感謝いたします．

本論文の研究を進めるにあたって使用した実験機材の一部，および本研究の原著論文の発表にあたっては，科学技術振興機構 CREST による支援をいただきました．また，慶應義塾先端科学技術研究センターの KLL 後期博士課程研究助成金と慶應義塾大学の大学院高度化推進研究費助成金から本研究に対する支援をいただきました．さらに，日本学生支援機構奨学金や慶應義塾大学大学院奨学金は，著者の研究活動の大きな支えとなりました．ここに感謝いたします．

最後に，経済的支援を惜しまず，著者の博士課程進学を支持し，現在まで暖かく見守っていただきました両親に心より感謝いたします．

論文目録

定期刊行誌掲載論文

- 花岡 美幸, 河野 健二: “ネットワーク侵入検知システムの協調による性能と耐障害性の向上”, 情報処理学会論文誌：コンピューティングシステム, Vol.5, No.1, pp.13–26, January 2012.

- Miyuki Hanaoka, Kenji Kono, Toshio Hirotsu, and Hirotake Abe: “Performance Improvement by Coordinating Configurations of Independently-managed NIDS”, *IJCSNS International Journal of Computer Science and Network Security*, Vol.11, No.5, pp.1–11, May 2011.

- Miyuki Hanaoka, Makoto Shimamura, and Kenji Kono: “TCP Reassembler for Layer7-aware Network Intrusion Detection/Prevention Systems”, *IEICE Transactions on Information and Systems*, Vol.E90-D, No.12, pp.2019–2032, December 2007.

国際会議論文

- *Miyuki Hanaoka, Kenji Kono, Toshio Hirotsu: “Performance Improvement by means of Collaboration between Network Intrusion Detection Systems”, In *Proceedings of the 7th Annual Conference on Communication Networks and Services Research (CNSR ’09)*, pp.262–269, May 2009.

- *Miyuki Hanaoka, Kenji Kono, Makoto Shimamura, and Satoshi Yamaguchi: “An Efficient TCP Reassembler Mechanism for Layer7-aware Network Intrusion Detection/Prevention Systems”, In *Proceedings of the 12th IEEE Symposium on Computers and Communications (ISCC ’07)*, pp.79–86, July 2007.

国内学会発表

- *花岡美幸，河野健二，廣津登志夫: “協調型ネットワーク侵入検知システム Brownie の提案と評価”, 情報処理学会 OS 研究会報告 (2009-OS-111), April 2009.
- *花岡 美幸，河野 健二: “ネットワーク侵入検知システムの協調による性能向上”, 情報処理学会コンピュータセキュリティシンポジウム 2008 (CSS 2008), pp.677–682, October 2008.
- *花岡 美幸, 河野 健二: “アプリケーション層プロトコルに対するパケット・レベルでのフィルタリング”, 情報処理学会 OS 研究会報告 (2005-OS-99), pp.91–98, May 2005.

参考文献

[1] Microsoft Security Bulletin MS01-033, November 2003. http://technet.microsoft.com/en-us/security/bulletin/ms01-033.

[2] David Moore, Colleen Shannon, and K Claffy. Code-Red: A Case Study on the Spread and Victims of an Internet Worm. In *Proceedings of the 2002 ACM SIGCOMM Internet Measurement Workshop (IMW '02)*, pp. 273–284, November 2002.

[3] Microsoft Security Bulletin MS02-039, January 2003. http://technet.microsoft.com/en-us/security/bulletin/ms02-039.

[4] David Moore, Vern Paxson, Stefan Savage, Colleen Shannon, Stuart Staniford, and Nicholas Weaver. Inside the Slammer Worm. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P '03)*, pp. 33–39, July 2003.

[5] Microsoft. Microsoft Security Bulletin MS08-067 - Critical: Vulnerability in Server Service Could Allow Remote Code Execution (958644). http://www.microsoft.com/technet/security/bulletin/MS08-067.mspx, October 2008.

[6] Phillip Porras, Hassen Saidi, and Vinod Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. http://mtc.sri.com/Conficker/, February 2009.

[7] UPI.com. Virus strikes 15 million PCs. http://www.upi.com/Top_News/2009/01/26/Virus-strikes-15-million-PCs/UPI-19421232924206/, January 2009.

[8] Aharon Etengoff. Nefarious Conficker worm racks up $9.1 billion bill. http://www.tgdaily.com/security-features/

42101-nefarious-conficker-worm-racks-up-91-billion-billz, April 2009.

[9] 特定非営利活動法人日本ネットワークセキュリティ協会. IT セキュリティ対策施策の導入・実施状況とその満足度調査集計結果報告書, January 2005. http://www.jnsa.org/houkoku2004/market_research/report2004.pdf.

[10] Gartner Inc. 平成 22 年度コンピュータセキュリティ早期警戒体制の整備事業（各国情報セキュリティ政策等の動向に関する調査研究）報告書（市場調査編）, March 2011. 経済産業省委託調査. http://www.meti.go.jp/policy/netsecurity/downloadfiles/22FY_ISmarket_research_report.pdf.

[11] Martin Roesch. Snort - Lightweight Intrusion Detection for Networks. In *Proceedings of the 13th USENIX Systems Administration Conference (LISA '99)*, pp. 229–238, November 1999.

[12] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, Vol. 31, No. 23–24, pp. 2435–2463, December 1999.

[13] Robin Sommer and Vern Paxson. Enhancing Byte-Level Network Intrusion Detection Signatures with Context. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*, pp. 262–271, October 2003.

[14] Christopher Kruegel and Giovanni Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS '03)*, pp. 251–261, October 2003.

[15] Ke Wang and Salvatore J. Stolfo. Anomalous Payload-Based Network Intrusion Detection. In *Proceedings of the 7th International Symposium on Recent Advances in Intrusion Detection (RAID '04)*, Vol. 3224/2004 of *Lecture Notes in Computer Science*, pp. 203–222, September 2004.

[16] 渡辺勝弘, 鶴岡信彦. 侵入検知システム. 電子情報通信学会「知識ベース」. 電子情報通信学会, 3 群, 7 編, 5 章, June 2010. http://www.ieice-hbkb.org/files/03/03gun_07hen_05.pdf.

[17] 伊藤良孝. ネットワーク型侵入検知システムの歴史と進化. http://www.snort.gr.jp/docs/N+I2005SnortBOF.pdf.

[18] Matthias Vallentin, Robin Sommer, Jason Lee, Craig Leres, Vern Paxson, and Brian Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID '07)*, Vol. 4637/2007 of *Lecture Notes in Computer Science*, pp. 107–126, September 2007.

[19] Konstantinos Xinidis, Ioannis Charitakis, Spiros Antonatos, Kostas G. Anagnostakis, and Evangelos P. Markatos. An Active Splitter Architecture for Intrusion Detection and Prevention. *IEEE Transactions on Dependable and Secure Computing*, Vol. 3, No. 1, pp. 31–44, January 2006.

[20] Jose M. Gonzalez, Vern Paxson, and Nicholas Weaver. Shunting: A Hardware/Software Architecture for Flexible, High-Performance Network Intrusion Prevention. In *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS '07)*, pp. 139–149, October 2007.

[21] Kathleen A. Jackson. Intrusion Detection System Product Survey. Technical Report LA-UR-99-3883, Los Alamos National Laboratory, 1999.

[22] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits. In *Proceedings of the ACM SIGCOMM 2004*, pp. 193–204, August 2004.

[23] Giovanni Vigna, William Robertson, Vishal Kher, and Richard A. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC '03)*, pp. 34–43, December 2003.

[24] Check Point Software Technologies Ltd. *Application Intelligence*. http://www.checkpoint.com/products/downloads/applicationintelligence_whitepaper.pdf.

[25] McAfee, Inc. *IntruShield Network IPS Appliances*. `http://www.mcafee.com/japan/products/pdf/IntruVert-NextGenerationIDSWhitePaper_en.pdf`.

[26] William Robertson, Giovanni Vigna, Christopher Kruegel, and Richard A. Kemmerer. Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks. In *Proceedings of the 13th Annual Network and Distributed System Security Symposium (NDSS '06)*, February 2006.

[27] Elvis Tombini, Hervé Debar, Ludovic Mé, and Mireille Ducassé. A Serial Combination of Anomaly and Misuse IDSes Applied to HTTP Traffic. In *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC '04)*, pp. 428–437, December 2004.

[28] Xinran Wang, Chi-Chun Pan, Peng Liu, and Sencun Zhu. SigFree: A Signature-free Buffer Overflow Attack Blocker. In *Proceedings of the 15th USENIX Security Symposium*, pp. 225–240, August 2006.

[29] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-level polymorphic shellcode detection using emulation. In *Proceedings of the 3rd Conference on Detection of Intrusions, Malware, and Vulnerability Assessment (DIMVA '06)*, pp. 54–73, July 2006.

[30] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Emulation-based Detection of Non-self-contained Polymorphic Shellcode. In *Proceedings of the 10th International Symposium on Recent Advances in Intrusion Detection (RAID '07)*, pp. 87–106, September 2007.

[31] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. An Empirical Study of Real-world Polymorphic Code Injection Attacks. In *Proceedings of the 2nd USENIX Workshop on Large-Scale Exploits and Emergent Threats (LEET '09)*, April 2009.

[32] Chenfeng Vincent Zhou, Christopher Leckie, and Shanika Karunasekera. A Survey of Coordinated Attacks and Collaborative Intrusion Detection. *Elsevier Computers & Security*, Vol. 29, No. 1, pp. 124–140, 2010.

[33] Vinod Yegneswaran, Paul Barford, and Somesh Jha. Global Intrusion Detection in the DOMINO Overlay System. In *Proceedings of the 11th Annual Network and Distributed System Security Symposium (NDSS '04)*, February 2004.

[34] Dshield.org. `http://www.dshield.org/`.

[35] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-End Containment of Internet Worms. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pp. 133–147, October 2005.

[36] Christian Kreibich and Jon Crowcroft. Honeycomb – Creating Intrusion Detection Signatures Using Honeypots. In *Proceedings of the 2nd Workshop on Hot Topics in Networks (HotNets-II)*, 2003.

[37] Sumeet Singh, Cristian Estan, George Varghese, and Stefan Savage. Automated Worm Fingerprinting. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp. 45–60, December 2004.

[38] Chris Clark, Wenke Lee, David Schimmel, Didier Contis, Mohamed Koné, and Ashley Thomas. A Hardware Platform for Network Intrusion Detection and Prevention. In *Proceedings of the 3rd Workshop on Network Processors & Applications (NP3)*, February 2004.

[39] David V. Schuehler and John W. Lockwood. TCP-Splitter: A TCP/IP Flow Monitor in Reconfigurable Hardware. In *Proceedings of the 10th on High Performance Interconnects Hot Interconnects (Hotl '02)*, pp. 127–131, August 2002.

[40] David V. Schuehler and John W. Lockwood. TCP Splitter: A TCP/IP Flow Monitor in Reconfigurable Hardware. *IEEE Micro*, Vol. 23, No. 1, pp. 54–59, January 2003.

[41] Yutaka Sugawara, Mary Inaba, and Kei Hiraki. High-speed and Memory Efficient TCP Stream Scanning using FPGA. In *Proceedings of the 15th International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 45–50, August 2005.

[42] The NSS Group. http://www.nss.co.uk/.

[43] Symantec Corporation. http://www.symantec.com/.

[44] Cisco Systems, Inc. http://www.cisco.com/.

[45] Juniper Networks, Inc. http://www.juniper.net/.

[46] Westline Security Ltd. http://www.west-line.net/.

[47] Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, Vol. 18, No. 6, pp. 333–340, June 1975.

[48] Beate Commentz-Walter. A String Matching Algorithm Fast on the Average. In *Proceedings of the 6th Colloquium on Automata, Languages and Programming*, Vol. 71/1979 of *Lecture Notes in Computer Science*, pp. 118–132, 1979.

[49] Sun Wu and Udi Manber. A Fast Algorithm for Multi-pattern Searching. Technical report, TR-94-17, May 1994.

[50] C. Jason Coit, Stuart Staniford, and Joseph McAlerney. Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort. In *Proceedings of DARPA Information Survivability Conference & Exposition II (DISCEX '01)*, pp. 367–373, June 2001.

[51] Mike Fisk and George Varghese. Applying Fast String Matching to Intrusion Detection. Technical Report In Preparation, successor to UCSD TR CS2001-0670, 2002.

[52] Nathan Tuck, Timothy Sherwood, Brad Calder, and George Varghese. Deterministic Memory-Efficient String Matching Algorithms for Intrusion Detection. In *Proceedings of the IEEE Infocom Conference 2004*, pp. 2628–2639, March 2004.

[53] Lin Tan and Timothy Sherwood. A High Throughput String Matching Architecture for Intrusion Detection and Prevention. In *Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA '05)*, pp. 112–122, June 2005.

[54] Liu Yang, Rezwana Karim, Vinod Ganapathy, and Randy Smith. Improving NFA-Based Signature Matching Using Ordered Binary Decision Diagrams. In *Proceedings of the 13th International Symposium on Recent Advances in Intrusion Detection (RAID '10)*, September 2010.

[55] Herbert Bos and Kaiming Huang. Towards Software-Based Signature Detection for Intrusion Prevention on the Network Card. In *Proceedings of the 8th International Symposium on Recent Advances in Intrusion Detection (RAID '05)*, pp. 102–123, September 2005.

[56] Willem de Bruijn, Asia Slowinska, Kees van Reeuwijk, Tomas Hruby, Li Xu, and Herbert Bos. SafeCard: A Gigabit IPS on the Network Card. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection (RAID '06)*, pp. 311–330, September 2006.

[57] Reetinder Sidhu and Viktor K. Prasanna. Fast Regular Expression Matching using FPGAs. In *Proceedings of the 9th Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, pp. 227–238, March 2001.

[58] Shaomeng Li, Jim Torresen, and Oddvar Søråsen. Exploiting Reconfigurable Hardware for Network Security. In *Proceedings of the 11th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '03)*, pp. 292–293, April 2003.

[59] Zachary K. Baker and Viktor K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *Proceedings of the 12th International Symposium on Field Programmable Gate Arrays (FPGA '04)*, pp. 223–232, February 2004.

[60] Haoyu Song, Todd Sproull, Mike Attig, and John Lockwood. Snort Offloader: A Reconfigurable Hardware NIDS Filter. In *Proceedings of the 15th International Conference on Field Programmable Logic and Applications (FPL '05)*, pp. 493–498, August 2005.

[61] Nicholas Weaver, Vern Paxson, and Jose M. Gonzalez. The Shunt: An FPGA-Based Accelerator for Network Intrusion Prevention. In *Proceedings of the 15th International Symposium on Field Programmable Gate Arrays*, pp. 199–206, 2007.

[62] Vern Paxson, Robin Sommer, and Nicholas Weaver. An Architecture for Exploiting Multi-Core Processors to Parallelize Network Intrusion Prevention. In *Proceedings of the 2007 IEEE Sarnoff Symposium*, pp. 1–7, 2007.

[63] Robin Sommer, Vern Paxson, and Nicholas Weaver. An Architecture for Exploiting Multi-Core Processors to Parallelize Network Intrusion Prevention. *Conurrency and Computation: Practice and Experience*, Vol. 21, No. 10, pp. 1255–1279, July 2009.

[64] Nigel Jacob and Carla Brodley. Offloading IDS computation to the GPU. In *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC '06)*, pp. 371–380, December 2006.

[65] Giorgos Vasiliadis, Spiros Antonatos, Michalis Polychronakis, Evangelos P. Markatos, and Sotiris Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID '08)*, pp. 116–134, September 2008.

[66] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P. Markatos, and Sotiris Ioannidis. Regular Expression Matching on Graphics Hardware for Intrusion Detection. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection (RAID '09)*, September 2009.

[67] Christopher Kruegel, Fredrik Valeur, Giovanni, Vigna, and Richard Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P '02)*, pp. 285–293, May 2002.

[68] Perminder Kaur, Dhavleesh Rattan, and Amit Kumar Bhardwaj. An Analysis of Mechanisms for Making IDS Fault Tolerant. *International Journal of Computer Applications*, Vol. 1, No. 24, pp. 22–25, 2010.

[69] Liwei Kuang and Mohammad Zulkernine. An Intrusion-Tolerant Mechanism for Intrusion Detection Systems. In *Proceedings of the 3rd International Conference on Availability, Reliability and Security (ARES '08)*, pp. 319–326, 2008.

[70] Lindonete Siqueira and Zair Abdelouahab. A Fault Tolerance Mechanism for Network Intrusion Detection System based on Intelligent Agents (NIDIA). In *Proceedings of the 4th IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems and 2nd International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA '06)*, pp. 319–326, 2008.

[71] Check Point Software Technologies Ltd. *Web Intelligence Tech Note*. http://www.checkpoint.com/products/downloads/web_intelligence_technote.pdf.

[72] Apache APR_PSPrintf Memory Corruption Vulnerability, May 2003. http://www.securityfocus.com/bid/7723/.

[73] CVE-1999-0172. Common Vulnerabilities and Exposures, September 1999. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-1999-0172.

[74] Microsoft IIS 5.0 "Translate: f" Source Disclosure Vulnerability, August 2000. http://www.securityfocus.com/bid/1578/.

[75] CERT Advisory CA-2002-27 Apache/mod_ssl Worm, October 2002. http://www.cert.org/advisories/CA-2002-27.html.

[76] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[77] G. Robert Malan, David Watson, Farnam Jahanian, and Paul Howell. Transport and Application Protocol Scrubbing. In *Proceedings of IEEE INFOCOM 2000*, pp. 1381–1390, March 2000.

[78] Mark Handley, Vern Paxson, and Christian Kreibich. Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics. In *Proceedings of the 10th USENIX Security Symposium*, August 2001.

[79] Umesh Shankar and Vern Paxson. Active Mapping: Resisting NIDS Evasion Without Altering Traffic. In *Proceedings of the 2003 IEEE Symposium on Security and Privacy (S&P '03)*, pp. 44–61, May 2003.

[80] The Apache Software Foundation. Apache http server. http://httpd.apache.org/.

[81] Mindcraft, Inc. WebStone. http://www.mindcraft.com/webstone/.

[82] Colleen Shannon, David Moore, and k claffy. Characteristics of Fragmented IP Traffic on Internet Links. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, pp. 83–97, November 2001.

[83] J. Postel. *RFC863: Discard Protocol*, 1983. http://www.ietf.org/rfc/rfc863.txt.

[84] S. Shenker and J. Wroclawaki. *RFC2216: Network Element Service Specification Template*, 1997. http://www.ietf.org/rfc/rfc2216.txt.

[85] Michael G. Merideth and Priya Narasimhan. Elephant: Network Intrusion Detection Systems that Don't Forget. In *Proceedings of the 38th Annual Hawaii International Conference on System Science (HICSS '05)*, pp. 309c–309c, January 2005.

[86] Nikto. http://cirt.net/nikto2.