2005－OS－99（15）
2005／5／27

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

花岡 美幸† 河野 健二††

† 電気通信大学大学院 電気通信学研究科 情報工学専攻
†† 慶應義塾大学 理工学部 情報工学科
電子メール：hanayuki@zeus.cs.uec.ac.jp，kono@ics.keio.ac.jp

要旨

インターネット・サーバに対する不正攻撃メッセージは，アプリケーション層プロトコルの規約に反していることが多い．そこで，プロトコルの正しい振舞いを定義し，それに反したメッセージを破棄する TCP ストリーム・フィルタという手法が提案されている．本手法はアプリケーション層でやりとりされるバイト列を解釈しつつフィルタリングを行うため，従来のパケット単位でのフィルタリングでは実現できない．本論文では，TCP/IP のプロトコル・スタックによる処理を行わずに，パケットの入れ替わり，IP フラグメント等に対処できる TCP ストリーム・フィルタの実現法を示す．Linux カーネルを拡張して実装を行い，Apache ウェブサーバを用いた実験により，フィルタリングのオーバヘッドは高々 3% 程度と十分小さいことが分かった．

Packet-level Implementation of TCP Stream Filter

Miyuki Hanaoka† Kenji Kono††

†Department of Computer Science, Graduate School of Electro-Communications, University of Electro-Communications
††Department of Information and Computer Science, Keio University
E-mail: hanayuki@zeus.cs.uec.ac.jp, kono@ics.keio.ac.jp

Abstract

Malicious messages to Internet servers often violate the rule of application-layer protocols. To filter out those malicious messages, TCP Stream Filter has been proposed. It drops the message that violates the rule that defines correct behavior of the protocol. Because we target application-layer protocols, the existing packet-level management is insufficient. In this paper we propose the packet-level implementation of TCP Stream Filter that can deal with out-of-order arrival of packets and IP fragmentation, etc., without processing the protocol stack of TCP/IP. Experimental results with Apache web server suggest that the overhead is small enough.

1 はじめに

近年，インターネット・サーバを狙った不正攻撃が数多く報告されており，問題となっている．インターネットは開放性･匿名性がきわめて高いため，インターネット上のサーバは常に不正攻撃の脅威に晒されている．そうした不正攻撃の多くは，メッセージに攻撃用プログラムを埋め込み，インターネット･サーバの実装上の不具合を悪用して，攻撃用のプログラムを実行する．

こうした不正攻撃を狙うメッセージは，RFC などで定められた規約や運営上の慣例に反していることが多い．例えば，HTTP の GET メッセージでは，要求する HTML ファイルの URL を含むが，ここに攻撃用のバイナリ・コードを入れるという攻撃手法がある．GET メッセージの URL は通常英数字といくつかの記号 (スラッシュやドットなど) からなることが多く，バイナリ・コードを含む場合，攻撃メッセージである可能性が高い．

そこで，プロトコル規約や運営上の慣例に反したメッセージを破棄すれば，多くの不正攻撃を防止できることに着目した TCP ストリーム・フィルタ [1] という手法が提案されている．TCP ストリーム･フィルタでは，通信路を流れるデータ・ストリームを監視してバイト列を解釈し，アプリケーション層プロトコルの規約などに反していないかどうかを検査する．例えば，HTTP の GET の例であれば,「URL は英数字といくつかの記号 (スラッシュやドットなど) からなる」と定義しておけば，バイナリ・コードを含む攻撃メッセージはこの定義に反しているため破棄される．このように正しいプロトコルを定義することによって精度の高い検査が可能となり，さまざまな攻撃を未然に防ぐことができる．

文献 [1] では，TCP ストリーム･フィルタをユーザレベルのライブラリとして実装しているため，サーバと TCP ストリーム・フィルタが同一計算機上で動作することが前提となっていた．一方，本論文では大規模クラスタ上で動作するサーバへの適用のしやすさを考慮して，ルータ等のミドルボックス上で動作することを想定した仕組みを提案する．ルータ等で扱うデータはパケット単位であるが，TCP ストリーム・フィルタはアプリケーション層プロトコルを対象とするため，単体のパケットではなく，データ・ストリームすなわち一連のパケットがフィルタリングの対象となる．データ・ストリームを参照を可能とするために，TCP/IP プロトコル･スタックを通す方法があるが，これはオーバヘッドが大きい．本論文では，TCP/IP のプロトコル・スタックによる処理を行わずに，データ・ストリームを参照できる仕組みを示す．データ・ストリームを参照するために，コネクション管理やパケット順序の入れ替わり，IP フラグメントなどを考慮した設計を行った．

TCP ストリーム・フィルタを Linux2.4.22 カーネルを拡張して実装し，Apache ウェブサーバ (ver. 2.0.52) を対象に HTTP プロトコルのフィルタリングを行ったところ，オーバヘッドは 3%にも満たなかった．

以下，2 章では TCP ストリーム・フィルタについて述べる．3 章で本論文での TCP ストリーム・フィルタの設計と実装について述べ，4 章で実験結果を示す．5 章で関連研究をまとめ，6 章で本論文をまとめる．

2 TCP ストリーム・フィルタ

2.1 概要

TCP ストリーム・フィルタは，プロトコル規約や運営上の慣例に反したメッセージを破棄すれば，多くの不正攻撃を防止できることに着目した手法である．TCP ストリーム・フィルタでは通信路を流れるデータ・ストリームを監視してバイト列を解釈し，アプリケーション層プロトコルの規約などに反していないかどうかを検査することによって，様々な攻撃を防ぐことができる．

例えば，SecurityFocus[2] で報告されているバッファ溢れ攻撃では，wu-ftpd という ftp サーバに攻撃用バイナリ・コードを埋め込んだパス名を送信する．ftp で用いるパス名は英数字といくつかの記号 (スラッシュやドットなど) からなることが多く，バイナリ・コードを含むパス名は攻撃メッセージである可能性が高い．TCP ストリーム・フィルタでは，パス名は英数字といくつかの記号から構成されるという制限を記述することができ，それに反したメッセージを破棄することができる．そのため，パス名中にバイナリ・コードを埋め込むことはできなくなり，こうした攻撃を未然に防ぐことができる．

2.2 tsfrule 言語

TCP ストリーム・フィルタでは，監視対象となるアプリケーション層プロトコルの振舞いを定義する

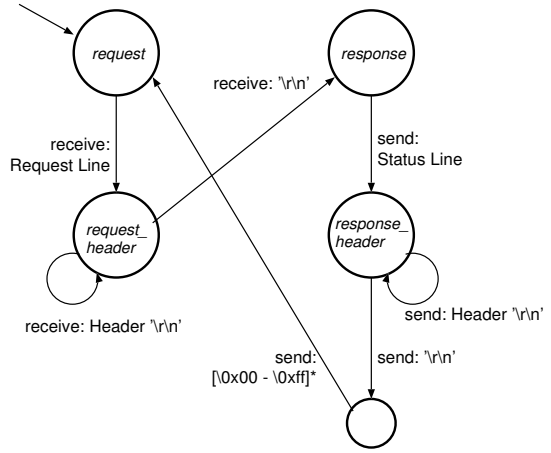

図 1: HTTP のプロトコル・オートマトン

ため，tsfrule (Tcp Stream Filtering RULE) 言語という専用言語が用意されている．

アプリケーション層プロトコルの多くは，メッセージの送受信をトリガとして状態遷移を行うオートマトンとして記述できる．tsfrule 言語ではこのオートマトンの定義を行う．オートマトンの各状態について，どのようなメッセージを送信・受信し，次にどの状態に遷移するのかを記述する．状態遷移のトリガとなるメッセージの定義には，正規表現を用いる．

2.2.1 HTTP プロトコルの記述例

tsfrule 言語の記述例として HTTP/1.0 の例を示す．HTTP/1.0 では，まずクライアントからサーバにリクエスト・メッセージを送る．リクエスト・メッセージは，`GET` などから始まるリクエスト・ラインと複数のヘッダからなる．その後，サーバはクライアントにレスポンス・メッセージを返す．レスポンス・メッセージは，ステータス・ラインと複数のヘッダ，要求されたファイルの内容からなる．これに対応する簡単なプロトコル・オートマトンを図 1 に示す．

図 1 に対応する tsfrule 言語による HTTP/1.0 のプロトコル記述例を図 2 に示す．プロトコル定義の冒頭でプロトコル名とポート番号を宣言し，3〜9 行目で正規表現の定義を行っている．ここでは，例えば半角空白に `SP`, 数字に `digit`, 英字またはピリオドまたはスラッシュの並びに `URL` という名前を付けている．

11〜14 行目は初期状態 *request* の定義である．12, 13 行目に記述された，

```
 1  protocol http(80);
 2
 3  regexpr SP    = ' ';
 4  regexpr digit = ['0'-'9'];
 5  regexpr URI   = ['a'-'z' 'A'-'Z' '.' '/']+;
 6  regexpr Version  = 'HTTP/' digit* '.' digit*;
 7  regexpr Header = ['a'-'z' 'A'-'Z' '0'-'9' ':']+;
 8  regexpr Status = digit digit digit;
 9  regexpr ReasonPhrase = ['a'-'z' 'A'-'Z']+;
10
11  state request {
12      <: 'GET' SP URI SP Version '\r\n'
13            -> request_header;
14  }
15
16  state request_header {
17      <: Header '\r\n' -> request_header;
18       | '\r\n' -> response;
19  }
20
21  state response {
22      :> Version SP Status SP ReasonPhrase '\r\n'
23            -> response_header;
24  }
25
26  state response_header {
27      :> Header '\r\n' -> response_header;
28       | '\r\n'
29            -> { :> .* -> request; }
30  }
```

図 2: HTTP/1.0 の記述例

```
<:  'GET' SP URI SP Version '\r\n'
      -> request_header;
```

は，`'GET' SP URI SP Version '\r\n'` にマッチするメッセージを受信したら，状態 *request_header* に遷移するということを示す．12 行目の冒頭の `<:` はメッセージを受信することを意味し，13 行目の `-> request_header` は状態 *request_header* に遷移することを表している．

16〜19 行目は，状態 *request_header* の定義である．18 行目冒頭の記号 `|` はメッセージによる遷移先の分岐を意味する記号であり，`Header` にマッチするメッセージを受信したら状態 *response_header* に遷移し，`'\r\n'` を受信したら状態 *response* に遷移することを表している．

同様に，21〜24 行目で状態 *response* の定義を，26〜30 行目で状態 *response_header* の定義をしている．29 行目の {} で囲まれた部分は名前無しの状態であり，`'.'` は全てにマッチすることを表す．

2.2.2 メッセージのフィルタリング

tsfrule 言語による記述はプロトコル・オートマトンに対応する内部表現に変換される．フィルタリングを行う際，オートマトン上の現在の状態を保持しつつ，メッセージ・ストリームの内容とマッチングを行いながら，状態遷移を行う．もし遷移できる状

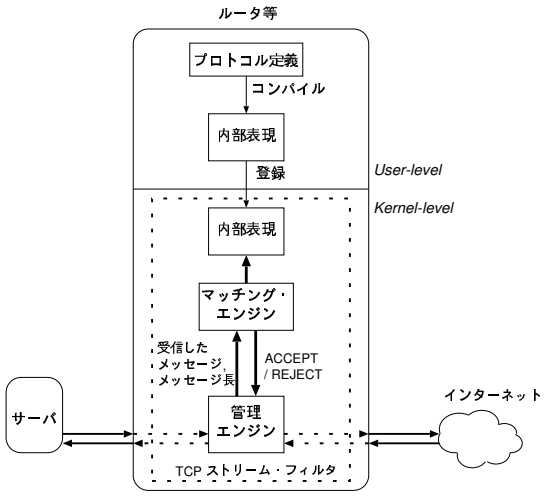

図 3: システム構成

態がなかったときは，プロトコル記述に反していると判断し，パケットを破棄する．

3 設計と実装

TCP ストリーム・フィルタはアプリケーション層プロトコルに従ってデータ・ストリームを解釈するため，データ・ストリームを参照できるような仕組みが必要となる．従来のパケット・フィルタがパケットごとにフィルタリングを行うのとは異なり，一連のパケットがフィルタリングの対象となる．これを，TCP/IP のプロトコル・スタックによる処理を行わずに実現するために，コネクション管理やパケットの順序の入れ替わり，フラグメンテーションを考慮したパケットの管理を行う．

3.1 システムの概要

TCP ストリーム・フィルタは，プロトコル定義を記述するプロトコル定義部と，プロトコル定義を解釈しながら通信を監視するフィルタリング・エンジンから構成される (図 3).

フィルタリング・エンジンは，管理エンジンとマッチング・エンジンからなる．管理エンジンでは，到着したパケットがどのコネクションに属すかというコネクション管理，パケットの到着順序の管理を行う．マッチング・エンジンはメッセージがプロトコル定義に合致しているかどうかを検査する．

3.2 動作の概略

本システムの動作の概略を示す．図 4 の右上に示したプロトコル記述を与え，それに反するメッセージを送るときを考える．

1. まず，到着したパケットがどのコネクションに属するかパケットなのかを判断する．

2. 順序通りにパケットが到着した場合には，到着順にパケットのペイロードを参照し，プロトコル記述とのマッチングを行う．プロトコル記述に合致していれば，そのままパケットを送信し，プロトコル記述に合致していなければ，そのパケットを破棄する．その様子を図 5 に示す．

3. 順序が入れ替わってパケットが到着した場合，そのパケットのコピーを保持し，そのパケットはサーバに送信する．順番通りのパケットが到着した時点で両パケットのペイロードを検査する．図 6 では，(3) のパケットが (2) より先に到着した．TCP ストリーム・フィルタは，(3) のコピーをとってそのままサーバに通過させる．その後 (2) が到着したら，(2) のチェックをしてから (3) のチェックをする．このとき，(3) のパケットのメッセージがプロトコル記述に反していることが分かる．このとき，遅れて到着した (2) のパケットを破棄する．

先に到着したパケットを通過させるこの方法では，先に通過させたパケットが攻撃メッセージを含んでいた場合が問題となる．このとき，攻撃メッセージを含んだパケットは通過させてしまっているため破棄できないが，かわりにあとから到着したパケットを破棄すれば良い．サーバ側では順番通りのパケットが到着するまでデータをアプリケーションに渡さないため，途中のパケットを通過させないことで，攻撃を防御することができる．

このとき，先に来たパケットをサーバ側に渡さないとすると，TCP ストリーム・フィルタ側で ACK 応答やフロー制御などの TCP/IP プロトコルの処理を全て行わなければならない．そこで，TCP ストリーム・フィルタでは，パケットのコピーをとったあとそのまま通過させる．こうすることで，プロトコルの処理はサーバ側の TCP/IP プロトコル・スタックが行うようにできる．

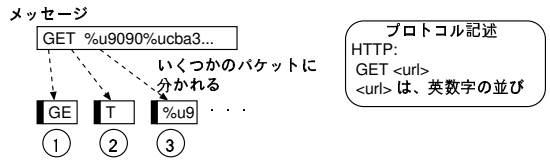

図 4: パケットの順序

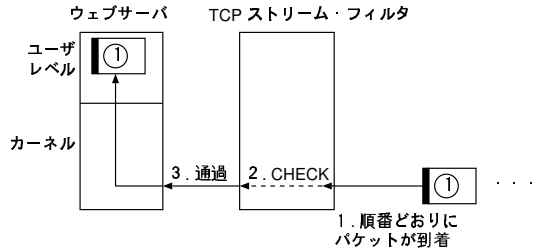

図 5: 順番通りに到着した場合

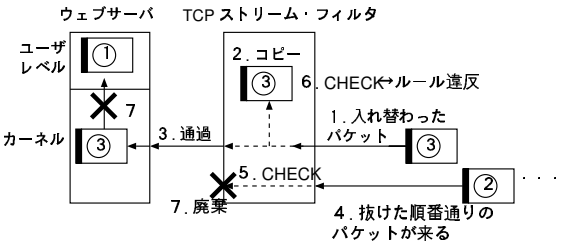

図 6: 順番が入れ替わった場合

3.3 コネクション管理

3.3.1 IP フラグメンテーションのない時

コネクション管理に用いる要素は，1) 発信元 IP アドレス，2) 宛先 IP アドレス，3) 発信元ポート番号，4) 宛先ポート番号，5) プロトコル番号 の 5 つである．

この 5 つをキーとしたコネクション情報を，コネクション確立時すなわちクライアントから SYN パケットが到着したときに作成して保持する．サーバからの SYN パケットが到着した時には，既にコネクション情報があるため何もしない．以後到着したパケットがどのコネクションのものなのかを上記 5 つのキーで判別する．そして，コネクションが切断されるとき，すなわちクライアント・サーバ双方から FIN パケットが来たときにコネクション情報を破棄する．

3.3.2 IP フラグメンテーションが起こったとき

IP 層では下位のデータリンク層が規定する最大パケット長 (MTU) に基づいて データ (TCP ヘッダも含む) を分割して送信する．このとき，分割されたデータにはそれぞれ IP ヘッダが付けられ，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，これによってもと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．MF フラグとフラグメント・オフセット両方の値が 0 の時はそのパケットは IP フラグメントされてなく，どちらかが 0 でないときは IP フラグメントされている．

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, 3.3.1 節で述べたコネクション管理は使えない. 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，3.3.1 節で述べたコネクション管理をする．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 つをキーとして用いて，どのコネクションに属するかを判別する．まず，SYN パケットをやりとりしてコネクションを確立するとき，ポート番号を含んだ 5 つをキーとして含むコネクション情報を作成すると同時に，ポート番号を含まず IP アドレスとプロトコル番号の 3 つをキーとするコネクション情報も作成する．その後， IP フラグメントされたパケットが到着したら，上記 3 つをキーとしてどのコネクションかを判別してパケットのコピーを保持しておく．

3.4 パケットの順序整理

3.4.1 IP フラグメンテーションのない時

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．最初に SYN パケットをやりとりしてコネクションを確立するときに，シーケンス番号の初期値を入れて送る．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．ここで，データ長とは，IP ヘッダの全データ長から，IP ヘッダのヘッダ長と TCP ヘッダのヘッダ長の和の 4 倍を引いたものである．

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サーバ・クライアント別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．

3.4.2 IP フラグメンテーションが起こったとき

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・オフセットの情報を利用して行う．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．このとき，次に来るべきパケットの識別子はそのパケットの識別子と同じ値であり，フラグメント・オフセットはフラグメント・オフセットとデータ長の和であると予測できる．ここで，データ長とは IP ヘッダの全データ長から IP ヘッダのヘッダ長の 4 倍を引いたものである．

フラグメントされたパケットが到着した場合，3.4.1 節と同様にパケットのコピーをリストとして順番通りになるように保持しておく．パケットのコピーのリストを前から見ていったとき，フラグメントの最後のパケット (MF フラグが 0) まで全てのパケットが揃っていたら，再構成をして 1 つのパケットにまとめる．

3.5 フィルタリング

コネクション管理とパケットの順序整理は管理エンジンで行い，メッセージの検査はマッチング・エンジンで行う．

管理エンジンでは，コネクションが確立するときにコネクション情報の作成を行う．このときそのコネクションのサーバ側のポート番号に対応するプロトコルルールを得，それと照合を行うマッチング・エンジンを生成してコネクション情報と共に保持する．以後パケットが到着したら，順序管理などをした後，マッチング・エンジンにメッセージ，その長さ，メッセージが受信か送信かを渡してメッセージを検査する．マッチング・エンジンはルールに合致していれば真，合致していなければ偽を返す．最後にコネクション切断時に，コネクション情報を破棄するのと同時にマッチング・エンジンも破棄する．

3.6 実装

本論文では，TCP ストリーム・フィルタをサーバとクライアントの間をつなぐゲートウェイ部分に Linux 2.4.22 カーネルを拡張して実装した．サーバ・クライアント双方からのパケットは全て，ゲートウェイに到着すると IP 層で `ip_rcv()` → `ip_forward()` → `ip_send()` という順に転送される．そこで，TCP ストリーム・フィルタの呼び出しは，IP 層の受信関数 `ip_rcv()` 内で行う．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` は受信したパケットに相当するものである．そのパケットを通過させてよいなら `PASS`，破棄させるなら `REJECT` を返す．

4 実験

4.1 実験方法

TCP ストリーム・フィルタによるオーバヘッドを計測するためにウェブ・サーバを対象とした実験を行った．

TCP ストリーム・フィルタを搭載したゲートウェイを介して通信を行う，クライアントとサーバを用意した．8Kbyte の HTML ファイルを 100 回取得する実験を 10 回行い，同時接続数を 1 から 16 に変化させてスループットと通信時間の平均を計測した．

また，サーバ・クライアント双方のカーネルに手を加えて，故意に 500byte ずつに IP フラグメントをおこし順番を入れ換えて送信するような環境を作り，同様にして実験を行った．順番の入れ換えは，フラグメントしたパケットを 2 つずつ入れ換えており，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．

ウェブサーバは広く利用されている Apache (ver. 2.0.52) を用い，ベンチマーク・プログラムは Apache に付属している ApacheBench (ver. 2.0.41) を用いた．Apache の設定は全てデフォルトとし，HTTP における KeepAlive を有効にして計測を行った．サーバ計

表 1: 通信時間 (msec) の比較

同時接続数	1	2	4	8	16
フィルタなし	1.193	1.575	3.124	6.273	12.660
フィルタあり	1.225	1.554	3.108	6.244	12.657
増加率 (%)	2.7	-1.3	-0.5	-0.5	0.0

表 2: スループット (Kbyte/sec) の比較

同時接続数	1	2	4	8	16
フィルタなし	6949	10535	10618	10574	10478
フィルタあり	6766	10670	10669	10622	10480
低下率 (%)	2.6	-1.3	-0.5	-0.5	0.0

表 3: 通信時間 (msec) の比較 (フラグメントあり)

同時接続数	1	2	4	8	16
フィルタなし	1.273	1.872	3.742	7.524	15.209
フィルタあり	1.275	1.868	3.724	7.485	15.120
増加率 (%)	0.2	-0.2	-0.5	-0.5	-0.6

表 4: スループット (Kbyte/sec) の比較 (フラグメントあり)

同時接続数	1	2	4	8	16
フィルタなし	6540	8855	8861	8815	8729
フィルタあり	6502	8878	8905	8860	8773
低下率 (%)	0.6	-0.3	-0.5	-0.5	-0.5

算機，ゲートウェイ，クライアント計算機は，それぞれ Pentium4 2.66GHz, Pentium4 2.4GHz, PentiumIII 600MHz のプロセッサ，512Mbyte, 512Mbyte, 128Mbyte の主記憶を備えている．プロトコル定義は HTTP1.0 相当のものを用いた．

4.2 結果

フラグメントのない場合について，表 1 に通信時間の計測結果，表 2 にスループットの計測結果を示す．また，同様にフラグメントがある場合について，表 3 に通信時間の計測結果，表 4 にスループットの計測結果を示す．

フラグメントがない場合，同時接続数が 1 のときは，フィルタリングによる性能低下が若干あり，通信時間が 2.7% 増加しスループットが 2.6% 低下している．しかし，同時接続数が 2 以上となると，ほとんど性能低下は見られなかった．このことから，TCP ストリーム・フィルタによるオーバヘッドは十分に小さいと言える．

一方，フラグメントがある場合は，同時接続数が 1 のときのフィルタリングによる性能低下は，通信時間が 0.2% 増加，スループットが 0.6% 低下と小さく，同時接続数が増えてもそれ以上の性能低下は見られなかった．このことから，フラグメントして順番が入れ替わったパケットの処理はほとんど影響しないものと考えられる．

5 関連研究

5.1 パケット・フィルタ

パケット・フィルタは，ネットワークのアクセス制御，監視をネットワーク層で行う手法であり，アプリケーション層でフィルタリングを行う TCP ストリーム・フィルタとは異なった仕組みである．パケット・フィルタには，インタプリタ型のスタックマシンをカーネル内で動作させ，ヘッダ情報等を用いてパケットのフィルタリングを行う CMU/Stanford packet filter (CSPF)[3] ，スタックマシンの代わりにアキュムレータマシンを用いて実行効率を改善した Berkley Packet Filter (BPF)[4] や，IP パケットを対象とした IP フィルタ [5] などがある．

パケット・フィルタはもともとネットワーク層を対象としているため，アプリケーション層を対象としたフィルタリングには適用しにくい．パケット単位でフィルタリングを行うため，データ・ストリームに対してフィルタリングを行うには複雑な状態管理が必要となるためである．

Stateful Packet Filter[6] や Dynamic Packet Filter[7] では，TCP の接続状態などを管理することができる．そのため，TCP の接続が確立しているポートに対してのみ通信を許可するなど，接続状態に応じたフィルタリングを可能としている．しかし，これらのフィルタリングでは，パケットのペイロードは参照しておらず，TCP ストリーム・フィルタは実現できない．

Stateful Inspection[8] や Deep Packet Inspection[9] と呼ばれる手法では，ペイロードを参照したフィルタリングを行うことができ，アプリケーション層でのフィルタリングが可能である．しかし，Stateful Inspection では，ペイロードの一部しか参照することができないため，順序整理をしてメッセージ全体を見てフィルタリングすることができない．また，Deep Packet Inspection はメッセージ全体を見ることができるが，TCP ストリーム・フィルタのように専用言語による支援がないため，フィルタリングのた

めにプロトコルごとに実装が必要になっている．

5.2 アプリケーション・ゲートウェイ

アプリケーション・ゲートウェイ (application gateway) [10][11] とは，プロキシ (proxy) と呼ばれるユーザ・プロセスを介して，通信内容のフィルタリングやアクセス制御などを行なう仕組みである．

アプリケーション・ゲートウェイはアプリケーション層に位置するため，仮想通信路を流れるデータ・ストリームに様々な処理を行なうことができる．しかし，ルール記述のための言語の支援がないため，アプリケーション層プロトコルごとに 1 からプロキシを作成しなければならない．また，アプリケーション・ゲートウェイでは，パケットの構成／分割，ユーザ／カーネル空間のデータのコピーが必要であり，クライアント・サーバ間の通信遅延が大きい．

アプリケーション・ゲートウェイの実現を容易にする研究として，フィルタリング・ルール記述用の専用言語を用いる Firmato[12] や，プロキシの実装を支援する TIS toolkit[13] などがある．しかし，プロトコル・スタックの処理が必要であるという点で，高い性能は期待できない．

6 おわりに

本論文では，TCP/IP のプロトコル・スタックによる処理を行わずに，パケットの入れ替わり，IP フラグメント等に対処できるような TCP ストリーム・フィルタの設計と実装を行った．TCP ストリーム・フィルタは，アプリケーション層プロトコルについて正しいプロトコルを定義し，それに反したメッセージを破棄する手法である．そのため，単体のパケットではなく一連のパケットを参照できる仕組みが必要となる．そこで，コネクション管理や IP フラグメンテーションも含めたパケットの順序管理について検討を行い，Linux カーネルを拡張して実装した．本論文で実装した TCP ストリーム・フィルタのオーバヘッドは十分小さく，Apache ウェブサーバを用いた実験では高々 3% 程度であった．

今後は，さらに現実的な状況での実験をして，さらなる検証をしていきたい．また，状態を遷移しながらメッセージを検査していく方法をネットワーク侵入検知システム (NIDS) に応用することで，精度の高い NIDS ができると考えられる．

謝辞

本研究の一部は，科学技術振興機構 CREST「ディペンダブル情報処理基盤」による支援を受けている．

参考文献

[1] 河野健二, 品川高廣, ラハト・カビル: TCP ストリームに対するフィルタリングによるインターネット・サーバの安全性向上, 情報処理学会論文誌: コンピューティングシステム, Vol. 46, No. SIG4(ACS9), pp. 33–44 (2005).

[2] SecurityFocus: Multiple Vendor C Library realpath() Off-By-One Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/8315/info/.

[3] Mogul, J. C., Rashid, R. F. and Accetta, M. J.: The Packet Filter: An Efficient Mechanism for User-level Network Code, *Proc. of 11th ACM Symposium on Operating System Principles*, pp. 39–51 (1987).

[4] McCanne, S. and Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture, *Proc. of USENIX Winter Technical Conference* (1993).

[5] Reed, D.: IP Filter. http://coombs.anu.edu.au/ avalon/.

[6] Hartmeier, D.: Design and Performance of the OpenBSD Stateful Packet Filter (pf), *Proc. of FREENIX Track: 2002 USENIX Annual Technical Conference (FREENIX '02)* (2002).

[7] Julkunen, H. and Chow, C. E.: Enhance Network Security with Dynamic Packet Filter, *Proc. of 7th International Conference on Computer Communications and Networks (IC3N '98)* (1998).

[8] Check Point Software Technologies Ltd.: *Stateful Inspection Technology*. `http://www.checkpoint.com/products/downloads/Stateful_Inspection.pdf`.

[9] Dharmapurikar, S., Krishnamurthy, P., Sproull, T. and Lockwood, J.: Deep Packet Inspection using Parallel Bloom Filters, *Proc. of 11th Symposium on High Performance Interconnects (HOTI'03)*, pp. 44–53 (2003).

[10] Avolio, F. M. and Ranum, M. J.: A Network Perimeter with Secure External Access, *Proc. of ISOC Symposium on Network and Distributed System Security* (1994).

[11] Cheswick, B. and Bellovin, S. M.: A DNS Filter and Switch for Packet-filtering Gateways, *Proc. of 6th USENIX Security Symposium*, pp. 15–20 (1996).

[12] Y. Bartal *et al.*: Firmato: A Novel Firewall Management Toolkit, *Proc. of IEEE Symposium on Security and Privacy*, pp. 17–31 (1999).

[13] Ranum, M. J. and Avolio, F. M.: A Toolkit and Methods for Internet Firewalls, *Proc. of USENIX Summer 1994 Technical Conference*, pp. 37–44 (1994).