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

河野健二[†] 品川高廣[††] ラハト カビル[†]

インターネット・サーバに対する不正攻撃では，通信メッセージに攻撃用プログラムを埋め込み，サーバの不具合を利用して攻撃用プログラムを巧みに実行する．こうした攻撃メッセージの多くは，アプリケーション層プロトコルの規約や運用上の慣例に反したメッセージであることが多い．本論文では，こうした攻撃メッセージを事前に破棄する TCP ストリーム・フィルタリングと呼ぶ手法を提案する．TCP ストリーム・フィルタリングでは，仮想通信路を流れるデータ・ストリームを監視し，プロトコル規約や運用上の慣例に反したメッセージを破棄する．監視対象のアプリケーション層プロトコルごとに “正しい” 振舞いを定義し，データ・ストリームの内容を解析・解釈しつつ定義違反のメッセージを検出する．そのため，規約や慣例に違反したメッセージを高い精度で検出することができる．TCP ストリーム・フィルタリングをすり抜けて攻撃を行うには，正しい振舞いに準じた攻撃メッセージを作成しなければならず，攻撃メッセージの作成が飛躍的に困難になる．また，フィルタリングのオーバヘッドも十分に小さく，Apache ウェブ・サーバを用いた実験では 10%未満であった．

Improving Internet Server Security by Filtering on TCP Streams

Kenji Kono,[†] Takahiro Shinagawa[††] and Md. Rahat Kabir[†]

To hijack Internet servers, the cracker sends malicious messages that attempt to execute attacking code by exploiting the servers' vulnerabilities. These malicious messages are often illegal; they do not conform to the application-layer protocol or to the administrative conventions. This paper proposes TCP Stream Filtering, a novel approach to defending against illegal messages that attempt to execute attacking code. In TCP Stream Filtering, the administrator defines the “correct” behavior of each application-layer protocol and the illegal messages, not conforming to the correct behavior, are detected by monitoring, parsing, and interpreting data streams running through the network connection. To deceive TCP Stream Filtering, the cracker must construct an attacking message that does conform to the correct behavior. Experimental results suggest that the overhead incurred by using TCP Stream Filtering is small enough.

1. はじめに

インターネットはその開放性・匿名性がきわめて高く，インターネット上のサーバはつねに不正攻撃の脅威にさらされている．実際，インターネット・サーバの脆弱性を突き，サーバの権限を乗っ取る攻撃がしばしば報告されている．こうした不正攻撃の多くは，通信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，攻撃用プログラムを巧みに実行する．

† 電気通信大学情報工学科
Department of Computer Science, The University of Electro-Communications
†† 東京農工大学共生科学技術研究部システム情報科学部門
Division of System Information Science, Institute of Symbiotic Science and Technology, Tokyo University of Agriculture and Technology

たとえば，バッファ溢れ攻撃[1)]と呼ぶ手法では，バッファ境界の検査が不十分なサーバに対しバッファ長を超えたメッセージを送りつけ，スタック上の戻りアドレスやヒープ上の関数ポインタを書き換え，通信メッセージに埋め込んだ攻撃用コードに制御を移す．また，フォーマット文字列攻撃[2)]と呼ぶ手法では，C 言語の標準関数である printf() などの使用法が不適切なサーバに，%n というフォーマット指定子を含むメッセージを送りつけ，通信メッセージに埋め込んだ攻撃用コードに制御を移す．

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用上の慣例に反していることが多い．たとえば，SecurityFocus[3)]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）から成ることが多く，バイナリ・コードを含むパス名は攻撃メッセージである危険性が高い．サーバの管理者であればパス名の慣例を熟知しており，“正しい” パス名の定義を与えることにより，攻撃メッセージを判別することができる．

本論文では，プロトコル規約や運用上の慣例に反したメッセージを破棄すれば，多くの不正攻撃が防止できることに着目し，**TCP ストリーム・フィルタリング**と呼ぶ方式を提案する．先に述べた ftp の例であれば，バイナリ・コードを含むパス名はサーバに受け渡さずに破棄する．TCP ストリーム・フィルタリングでは，仮想通信路を流れるデータ・ストリームを監視し，アプリケーション層プロトコルの規約に従ってデータ・ストリーム中のバイト列を解釈し，パス名などを正確に認識し，運用上の慣例に反していないかどうか検査する．これによって精度の高い検査が可能となっており，さまざまな攻撃を未然に防ぐことができる．

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，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正しく定義する必要がある．TCP ストリーム・フィルタリングでは，プロトコルの振舞いを定義する専用言語が用意されており，1) メッセージのフォーマット，2) メッセージの送受信順序，3) メッセージ長の制限などを宣言的に記述すればよい．これを**プロトコル定義**（protocol definition）と呼ぶ．TCP ストリーム・フィルタリングの特徴は次のとおりである．

- **攻撃メッセージの作成が困難：** TCP ストリーム・フィルタリングをすり抜けるためには，プロトコル規約や運用上の慣例に従った攻撃メッセージを作成しなければならず，攻撃用メッセージの作成が難しい．wu-ftpd [3] のようにパス名の解析部に脆弱性が発見されても，パス名に任意のバイナリ・コードを埋め込むことができないため，攻撃を達成することは難しい．
- **未知の攻撃に対する有効性：** TCP ストリーム・フィルタリングでは，プロトコル定義に準拠した攻撃メッセージでない限り，未知の攻撃であっても防ぐことができる．シグネチャ・マッチングという攻撃メッセージの検出方式では，既知の攻撃メッセージをデータベース化し，データベースと照合を行って攻撃メッセージを検出する．そのため，データベースに登録されていない攻撃は防ぐことができず，未知の攻撃には脆弱である．
- **フィルタリング・ルールの柔軟性：** TCP ストリーム・フィルタリングでは，専用言語を用いてプロトコル定義の作成や保守ができるため，新しいプロトコルへの対応や運用サイト固有のフィルタリング・ルールの追加が容易である．たとえば，CGI の入力に含まれる危険な HTML タグを排除するルールを追加することができる．

TCP ストリーム・フィルタリングは，現在，そのプロトタイプが Linux2.4.20 上で動作している．フィルタリング・エンジンは，ネットワーク通信用の標準関数呼び出しをフックする形で実現されており，既存のサーバを改変することなく利用できる．また，暗号化通信にも適用可能な実装となっており，暗号通信用ライブラリにフックを行い，暗号化前や復号化後にフィルタリングを行うようにすればよい．

本プロトタイプを用いて，Apache ウェブサーバ（ver. 2.0.50）に対してフィルタリングを行ったところ，そのオーバヘッドは 10%に満たなかった．また，ウェブサーバの負荷が高くなると，フィルタリング以外の処理が性能上のボトルネックとなり，相対的にフィルタリングのオーバヘッドが無視できるところまで小さくなるという結果が得られた．

なお，本方式は HTTP や SMTP などの文字列主体のアプリケーション層プロトコルを対象としている．また，TCP ストリーム・フィルタリングという名前ではあるものの，トランスポート層のプロトコルを TCP/IP に限定する必要はなく，コネクション指向の信頼性のあるプロトコルを用いることができる．

以下，2 章では TCP ストリーム・フィルタリングが対象とする攻撃例を示す．3 章では TCP ストリーム・フィルタリングの設計と実装を示し，4 章では実験結果を示す．5 章では関連研究についてまとめ，6 章で本論文をまとめる．

2. 対象となる攻撃例

本章では，TCP ストリーム・フィルタリングで防御可能な攻撃例を示す．これらの攻撃例は TCP ストリーム・フィルタリングでなければ防止できないというものではなく，それぞれの攻撃に対して個別に対策を講じることもできる．TCP ストリーム・フィルタリングの利点は，プロトコル定義を適切に記述しておけば，攻撃手法にかかわらずさまざまな攻撃が防止できる点にある．

2.1 極端に長いメッセージ

極端に長いメッセージを送信しサーバを停止させるという単純な攻撃法がある．このような脆弱性は最近でも報告されている [4]．TCP ストリーム・フィルタリングは，極端に長いメッセージを破棄することが可

能であり，このような攻撃を未然に防止することができる．

2.2 バイナリ・コードを含むパス名

HTTP の URI や ftp のパス名などは，スラッシュ，ドット，チルダなどの記号を正しく解釈しなければならず，その扱いは繁雑になりがちである．そのため，パス名の解析部に潜む脆弱性はしばしば報告されており，`realpath()` という標準ライブラリ関数ですらバッファ溢れの脆弱性が指摘されている[3]．`realpath()` の脆弱性を利用した攻撃では，1 バイトのバッファ溢れを利用し，パス名中に埋め込んだバイナリ・コードを巧妙に実行する．

TCP ストリーム・フィルタリングでは，パス名は英数字と特定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．

2.3 フォーマット文字列攻撃

フォーマット文字列攻撃は，`printf("%s", msg)` とすべき箇所で，`printf(msg)` とする誤りを利用する．文字列 `msg` が %n という指示子を含む巧妙なメッセージの場合，深刻なセキュリティ・ホールになりうることが知られている[2]．

プロトコル規約や運用上の慣例から %n という奇妙な文字列が出現しないと期待できることは多い．そのような場合，TCP ストリーム・フィルタリングを用いて，%n を含むメッセージを破棄することができる．

なお，フォーマット文字列攻撃を防止するには，`printf(s)` を `printf("%s", s)` に機械的に置換すればよいように見える．しかし，現実はそれほど単純ではなく，静的な解析[5] や動的な検査[6] を行う手法が提案されている．

2.4 想定外の制御文字

CGI の入力に HTML タグを埋め込むなど，メッセージ中に想定外の制御文字を埋め込む攻撃がある．CGI プログラミングなどでは，プログラミング上の慣例として，ユーザからの入力内容を検証してから利用することが推奨されている．しかし，このような慣例を厳密に順守することは難しく，熟練したプログラマでも誤りを犯す場合がある．TCP ストリーム・フィルタリングでは，正しい入力だけを受理するプロトコル定義を記述しておき，危険な入力を事前に破棄することができる．

3. TCP ストリーム・フィルタリング

3.1 システム構成

TCP ストリーム・フィルタリングは，プロトコル定義を記述するプロトコル定義言語と，プロトコル定義を解釈しながら通信の監視を行うフィルタリング・エンジンから構成される．

TCP ストリーム・フィルタリングを利用するには，監視対象となるアプリケーション層プロトコルについて，1) メッセージのフォーマット，2) メッセージの送受信順序，3) メッセージ長の制限などを定義したプロトコル定義を記述する．プロトコル定義は，tsfrule（Tcp Stream Filtering RULE）言語という専用言語を用いて記述でき，tsfrule 言語コンパイラによってフィルタリング・エンジンが解釈できる内部表現に変換される．

通信内容の検査を行うフィルタリング・エンジンは動的リンク・ライブラリとして実装されており，サーバ・アプリケーションにリンクされて動作する．フィルタリング・エンジンはサーバ・アプリケーション本体と通信ライブラリの間に介在し，1) 通信ライブラリ呼び出しをフックする部分と，2) プロトコル定義との照合を行うマッチング・エンジンとから構成される．

図 1 の例では，サーバ・アプリケーションが `recv()` を呼び出し，メッセージを受信している．フィルタリング・エンジンはその呼び出しをフックし，マッチング・エンジンを呼び出す．マッチング・エンジンは受信した内容がプロトコル定義に従っているかどうか検査し，従っていれば受信した内容をサーバに返す．従っていなければエラー・コードを返す．

3.2 tsfrule 言語

tsfrule 言語はクライアント・サーバ間でやりとりされるメッセージの送受信順序やフォーマットを定める専用言語であり，いわば “正しい” メッセージのやりとりを定める．

3.2.1 プロトコル・オートマトンとメッセージ定義

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．このオートマトンを**プロトコル・オートマトン**（protocol automaton）と呼ぶ．

tsfrule 言語ではプロトコル・オートマトンの定義を行う．プロトコル・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受信または送信し，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フォーマットは，正規表現を用いて定義する．

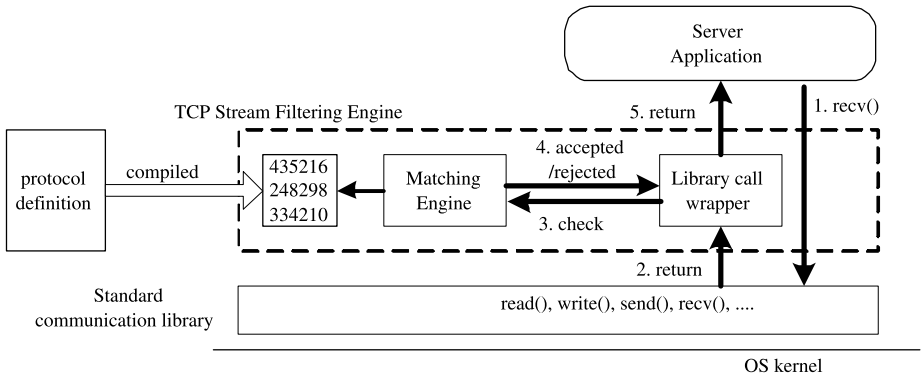

図 1 TCP ストリーム・フィルタリングのシステム構成
Fig. 1 System architecture of TCP Stream Filtering.

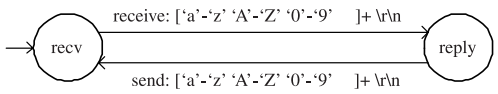

図 2 エコー・プロトコルのオートマトン
Fig. 2 Protocol automaton for the echo protocol.

メッセージ・フォーマットの定義に，より記述力のある文脈自由文法を用いるという設計も可能である．しかし，メッセージ・フォーマットの定義に用いる文法の記述力とマッチング効率とはトレードオフの関係にあり，ここでは次の理由から正規表現を採用した．1) POP3，SMTP，HTTP，FTP などの実用プロトコルでも正規表現で記述できる，2) マッチング効率が良い，3) 実装が容易である．tsfrule 言語の記述力については，3.3 節で議論する．

3.2.2 エコー・プロトコルの例

tsfrule 言語の概要を示すため，エコー・プロトコルの定義例を示す．エコー・プロトコルに対応するプロトコル・オートマトンを **図 2** に示す．エコー・プロトコルは *recv* と *reply* という 2 つの状態を持ち，初期状態は *recv* である．*recv* 状態でメッセージを受信すると *reply* 状態に遷移する．*reply* 状態でメッセージを送信すると，再び *recv* 状態に遷移する．なお，やり取りされるメッセージは英数字の並びとしている．

図 3 に図 2 に対応するプロトコル定義を示す．プロトコル定義の冒頭ではプロトコル名とポート番号を宣言し，2 行目と 3 行目では正規表現の定義を行っている．ここでは英数字の並びに *echo_message* という名前を付け，改行コードに *CRLF* という名前を付けている．

4 行目から 6 行目は，初期状態 *recv* の定義である．状態の定義中に記述された，

<: *echo_message CRLF* –> *reply*;

は，*echo_message CRLF* にマッチするメッセージを

```
1: protocol echo(9000);

2: regexpr echo_message=[‘a’-‘z’‘A’-‘Z’‘0’-‘9’]+;
3: regexpr CRLF = ‘\r\n’;

4: state recv {
5:         <: echo_message CRLF –> reply;
6: }

7: state reply {
8:         :> echo_message CRLF –> recv;
9: }
```

図 3 エコー・プロトコルの定義例
Fig. 3 An example definition of the `echo` protocol.

受信した後，状態 *reply* に遷移するということを表す．冒頭の <: はメッセージを受信することを意味し，末尾の –> *reply* は状態 *reply* に遷移することを表す．

同様に，7 行目から 9 行目は，状態 *reply* の定義であり，*echo_message CRLF* にマッチするメッセージを送信したのち，状態 *recv* に遷移することを表す．8 行目の冒頭の :> はメッセージを送信することを表す．

3.2.3 状態遷移先の分岐

アプリケーション層プロトコルでは，受け取ったメッセージに応じて異なる状態に遷移することがある．これを状態遷移先の分岐と呼ぶ．たとえば，POP3 における認証では，ユーザ名とパスワードを受信したのち，認証が成功すれば `+OK` を返答し *transaction* 状態に遷移する．認証に失敗すれば `-ERR` を返答し，*terminated* 状態に遷移する．

このような遷移先の分岐を記述するため，tsfrule 言語では記号 | を用いて，メッセージ・フォーマットと遷移先を並列に定義することができる．POP3 における認証は次のように定義すればよい．

```
1: :> ‘+OK’ –> transaction;
2: | ‘-ERR’ ‘–> terminated;
```

これは，+OK メッセージを送信した場合は *transaction* 状態に遷移し，-ERR メッセージを送信した場合は *terminal* 状態に遷移することを定義している．

3.2.4 メッセージ長制約

アプリケーション層プロトコルの規約を定めたRFCなどでは，しばしばメッセージ長についての制限が記載されている．たとえば，POP3 ではサーバからの返答は 512 バイト以下であることが定められている．こうしたメッセージ長制約に違反したメッセージは，バッファ溢れ攻撃などを起こす可能性が高いため，メッセージ長に関する制約は tsfrule 言語でも厳密に定義できる必要がある．

メッセージ長に関する制約を定義するため，正規表現にマッチする文字列長を指定できるようにした．正規表現に : [<= *length*] という修飾子をつけると，その正規表現にマッチする文字列は *length* バイト以下に制限される．たとえば，[‘a’-‘z’]+ : [<= 512] とすると，最長 512 バイトの英小文字の列にマッチする．指定された文字列長を超えるメッセージは破棄される．他に : [< *length*]，: [== *length*] という修飾子があり，それぞれ *length* バイト未満，*length* バイトに一致を表す．

メッセージ長には動的に決まる値を用いることも可能であり，送受信するメッセージの長さが動的に決まる場合にも対応できる．HTTP では返信する HTML ファイルの長さを Content-Length というフィールドで指定する．たとえば，8,192 バイトのファイルを返信する場合，Content-Length: 8192 というメッセージを返信し，それに続けて 8192 バイトのバイト列を送信する．このようなメッセージ長の指定は次のように定義する．

```
1: :> ‘Content-Length: ’ [‘0’−‘9’]+
2:                 : [$ length = atoi(this); $];
3: :> .* : [== length ];
```

最初の行では，Content-Length: に続けてバイト長を送信することを定義している．バイト長に相当する正規表現 [‘0’−‘9’]+ は : [$···$] で修飾されている．: [$, $] で囲まれた部分には任意の C プログラムが記述でき，変数 this を用いて，修飾されている正規表現（ここでは [‘0’−‘9’]+）にマッチした文字列を参照することができる．2 行目では変数 length にバイト長を代入している．その次の行では，変数 length を用いて，Content-Length フィールドで指定したバイト数のバイト列を送信することを定義している．

3.2.5 フィルタの動的起動

アプリケーション層プロトコルには動的なネゴシエーションによって，使用するポート番号を決めるものがある．ftp におけるデータ接続用ポートがその例である．このようなプロトコルに対してフィルタリングを行うには，動的に決まるポートに対してフィルタリング・ルールを適用しなければならない．TCP ストリーム・フィルタリングは全メッセージを監視しているので，ネゴシエーションの過程を追従し使用するポート番号を知ることができる．そのため，実行時に定まるポートに対してもフィルタリングを行うことができる．

ここでは ftp プロトコルを簡略化した例を用い，フィルタの動的起動の定義を示す．ftp では PORT コマンドを用いて，データ接続に用いるポート番号を指定できる．PORT コマンドからポート番号を抽出するには，次のようにすればよい．

```
:> ‘PORT’ [‘0’−‘9’]+ : [$ port=atoi(this);$]
```

PORT コマンドの送信後，データ接続に対してフィルタリングを行うには，PORT コマンドの定義に

activates *ftp-data* **on** *port*

を付加する．これは *ftp-data* で指定されたプロトコル定義を，ポート番号 *port* の接続に適用することを示す．

なお，プロトコル定義の記述を容易にするため，tsfrule 言語ではさまざまな糖衣構文（syntax sugar）を用意している．これらの糖衣構文を用いると，名前なしの状態（anonymous state）を定義することができ，状態名を定義する煩わしさを軽減できる．また，デフォルトの状態遷移先を指定することができ，メッセージ定義ごとに遷移先を定義する面倒を避けることができるようになっている．

3.3 記 述 力

TCP ストリーム・フィルタリングの有効性は，tsfrule 言語の記述力に依存する．tsfrule 言語はプロトコル・オートマトンと正規表現の組合せという簡潔な仕様であるにもかかわらず，POP3 や HTTP などの主要なアプリケーション層プロトコルの定義が可能である．実際，tsfrule 言語を用いて POP3，HTTP，SMTP，FTP の記述を行い，その記述力を確認している．なお，tsfrule 言語は著者らが開発を行った April[7] というコード生成ツールの派生である．

なお，これらのプロトコルの中には入れ子の括弧を含むメッセージが許されているものがある．RFC の規約上では任意の段数の入れ子が許されているものの，実際にはたかだか数段程度の入れ子しか利用されることはなく，入れ子の段数を限定しても問題はないと期待される．そのため，たとえば二重の入れ子括弧にま

で対応することとし，正規表現を用いて (‘(’ (‘(’ ‘)’)* ‘)’)* のように記述を行った．

3.4 攻撃の防止例

tsfrule 言語を用いるとプロトコルの規約や運用上の慣例を明確に定義することができ，さまざまな攻撃を防止できる．広く利用されているプロトコルを例にその定義を示し，防止可能な攻撃例を示す．

メール受信プロトコルの 1 つである POP3 では，挨拶メッセージの送信後，サーバはクライアントのユーザ名を受信する．POP3 を定めた RFC1939 では，ユーザ名は表示可能（printable）な ASCII 文字列から成り，その長さは 40 文字以下と定められている．これを tsfrule 言語で定義すると次のようになる．

```
/* ユーザ名は表示可能な ASCII 文字から成る*/
regexpr username=([ ‘0x20’-‘0x7e’]+)[<= 40];

state authorization {
    /* 挨拶メッセージの送信 */
    :> ‘+OK ’ greeting CRLF;

    /* ユーザ名の受信 */
    <: ‘USER ’ username CRLF;
    ...
}
```

文献 8) で報告されている脆弱性では，攻撃用バイナリ・コードを仕組んだ長いユーザ名を送付し，POP サーバの権限でその攻撃用コードを実行することができる．また，文献 9) で報告されている脆弱性では，%n を含むユーザ名を用いてフォーマット文字列攻撃を行い，ユーザ名に仕組まれた攻撃用コードを実行させることができる．これらの攻撃では，ユーザ名に仕組む攻撃用コードに表示可能な ASCII 文字以外の文字が含まれていたり，ユーザ名の長さが 40 文字を超えていたりすると，*username* の定義に反してしまう．攻撃用バイナリ・コードは機械語の命令列であり，シェルを起動するような攻撃用コードの多くは，表示可能な ASCII 文字以外の文字が含まれていることが多い．

文献 3) で報告されている ftp サーバに対する攻撃では，攻撃用バイナリ・コードを埋め込んだ長いパス名に送信する．FTP プロトコルを定めた RFC959 では，パス名は CR，LF 以外の ASCII 文字から成ると定義されている．したがって，パス名の定義は次のようになる．

```
regexpr pathname =
    [‘0x00’-‘0x09’ ‘0x0b’-‘0x0c’ ‘0x0e’-‘0x7f’]+
```

カレント・ディレクトリを変更する CWD メッセージのように，パス名を含むメッセージは次のように定義すればよい．

```
state commands {
    ...
    <: ‘CWD ’ pathname CRLF;
    ...
}
```

これによって，パス名に非 ASCII 文字（0x80〜0xff）を含ませることはできなくなり，パス名に攻撃コードを含ませることが困難になる．

これらの例では，RFC で定められた規約に沿ってプロトコル定義を記述している．そのため，RFC の規約に違反しないように巧妙に攻撃メッセージを作成することができれば，TCP ストリームフィルタをすり抜けることが可能である．そのような攻撃を防止するには，各サイトにおける運用上の慣例を利用し，より厳格な検査を行うようにプロトコル定義を変更すればよい．たとえば，POP クライアントのユーザ名は英数字のみから成るという運用上の慣例があれば，*username* の定義を次のように変更する．この場合，攻撃コードに含められる機械語は英数字に限定され，攻撃メッセージの作成はさらに困難なものになる．

```
regexpr username=([ ‘a’-‘z’ ‘A’-‘Z’ ‘0’-‘9’]+)
    [<= 40];
```

プロトコル規約では認められていても，慣例としては好ましくないメッセージを破棄すれば安全性を高められる例は多い．たとえば，CGI の入力に HTML タグを組み込む攻撃は，プロトコル規約上の違反はしていない．しかし，CGI への入力を定める正規表現を変更し，HTML タグの開始，終了を示す <，> を含む文字列にはマッチしないように変更すれば，このような攻撃も防ぐことができる．

3.5 フィルタリング・エンジンの実装

3.5.1 通信ライブラリのフック

通信ライブラリ呼び出しのフックは環境変数 LD_PRELOAD を用いて実現している．環境変数 LD_PRELOAD を用いると，標準ライブラリ内で定義されている関数をフックし，独自の機能を追加することができる．現在のプロトタイプはソケット通信に関する 12 種類の関数をフックしている．監視対象のポートへの接続の検出，フィルタリング対象の通信路に結び付けられた記述子の特定を行うため，socket()，bind()，accept() をフックしている．また，通信内容の検査を行うため実際に通信を行う send()，recv() などをフックし，必要に応じてマッチング・エンジンを呼び出してプロトコル定義との照合を行う．

環境変数 LD_PRELOAD を用いてライブラリ呼び出しのフックを行っているため，既存のサーバ・アプリケーション，通信ライブラリ，OS カーネルなどに改変を

表 1 マッチング・エンジンの提供する関数
Table 1 Functions provided by TCP matching engine.

`tsf_protocol_t *tsf_get_protocol_by_name(const char *name)` 引数 `name` で指定されたプロトコル名を持つプロトコルの定義を得る.
`tsf_protocol_t *tsf_get_protocol_by_port(uint16_t port)` 引数 `port` で指定されたポート番号を用いるプロトコルの定義を得る.
`tsf_engine_t *tsf_create_engine(tsf_protocol_t *protocol)` 引数 `protocol` で指定されたプロトコル定義と照合を行うマッチング・エンジンを生成する.
`bool tsf_check(tsf_engine_t *e, void *msg, size_t len, tsf_dir_t dir)` 送信または受信しようとしているメッセージがプロトコル定義に合致しているかどうか検査する. 合致していれば真, 合致していなければ偽を返す. 引数 `e` はマッチング・エンジン, `msg` は検査対象のメッセージ, `len` はその長さ, `dir` はメッセージが送信か受信かを指定する.
`void tsf_destroy_engine(tsf_engine_t *e)` マッチング・エンジン `e` の使用していた資源を解放する.

行う必要がない. ただし, LD_PRELOAD による標準ライブラリのフックは, 標準ライブラリが静的にリンクされている場合や setuid プログラムには無効であるため, このような場合にはサーバ・アプリケーションの再リンクを行う必要がある.

また, フィルタリング・エンジンはユーザ空間で動作するため, 送受信するメッセージの内容はユーザ空間に存在する必要がある. 通常の `send()` や `recv()` では, 送受信されるメッセージはユーザ空間に存在する. しかし, ファイルの内容を送信する `sendfile()` というシステムコールは, ファイルの内容をユーザ空間に読み出すことなく送信する. そのため, マッチング・エンジンから送信内容を参照することができない. 現在の実装では, 一度ファイルの内容をユーザ空間に読み出し, 通信内容の検査を行ってからファイルの送信を行っている.

これは `sendfile()` 関数の利点を損なう実装であり工夫の余地がある. 攻撃メッセージからサーバ・アプリケーションを防御するという視点からすれば, サーバが送信するファイルの内容を検査する必要はない. したがって, tsfrule 言語を拡張し, 特定のメッセージについては検査を行わないという指定ができるようにすればよい. あるいは, フィルタリング・エンジンをカーネル内で動作させればよい.

3.5.2 マッチング・エンジン

プロトコル定義との照合を行うマッチング・エンジンは, 受信または送信しようとしているメッセージの内容を検査する. tsfrule 言語によって記述されたプロトコル定義は, マッチング・エンジンの利用する内部表現に変換される. 正規表現は決定性の有限状態オートマトンに変換され, マッチング・エンジンは正規表現のマッチングを行いながら, プロトコル状態の遷移を行う. マッチング・エンジンは**表 1** に示した関数を提供しており, 通信ライブラリのフック関数はこれらの関数を利用して通信内容の検査を行う.

3.5.3 暗号化通信への対応

TCP ストリーム・フィルタリングの特徴の 1 つに, 暗号化された通信にも対応できるという点がある. フィルタリング・エンジンがサーバ・アプリケーションと通信ライブラリを仲介する形で実現されているため, 暗号通信を行うライブラリとサーバ・アプリケーションの仲介をするように変更し, 暗号化前/復号化後の通信内容に対してフィルタリングを行うようにすればよい. たとえば, SSL による暗号通信を提供する OpenSSL [10] ライブラリであれば, メッセージ送信を行う `ssl_write()`, メッセージ受信を行う `ssl_read()` をフックし, `ssl_write()` 内では暗号化前のメッセージに対してフィルタリングを行い, `ssl_read()` では復号化後のメッセージにフィルタリングを行うようにすればよい.

4. 実　　験

TCP ストリーム・フィルタリングによるオーバヘッドを計測するため, フィルタリングを行ったウェブサーバと行っていないウェブサーバの性能比較を行った. ウェブサーバに対する同時接続数を変更しながら, 8 Kbyte の HTML ファイルを 100 回取得する実験を 10 回行い, スループットと通信時間の平均を計測した. プロトコル定義は HTTP1.0 相当のものを用いた. このプロトコル定義は約 200 行程度で 12 の状態数を持つ.

ウェブサーバは広く利用されている Apache (ver. 2.0.50) を用い, ベンチマーク・プログラムは Apache に付属の ApacheBench (ver. 2.0.40) を用い

表 2 スループット（Kbyte/sec）の比較
Table 2 Throughput (Kbyte/sec).

同時接続数	1	2	4	8	16
フィルタなし	7010	11332	11441	11456	11457
フィルタあり	6471	11230	11450	11453	11455
低下率（%）	7.7	9.0	0.0	0.0	0.0

表 3 通信時間（msec）の比較
Table 3 Latency (msec).

	1	2	4	8	16
フィルタなし	1.19	1.46	2.90	5.80	11.59
フィルタあり	1.28	1.48	2.90	5.80	11.59
増加率（%）	7.6	1.4	0.0	0.0	0.0

た．Apache の設定はすべてデフォルトとし，HTTP における KeepAlive を有効にして計測を行った．サーバ計算機，クライアント計算機はともに Pentium4 2.8 GHz のプロセッサ，512 Mbyte の主記憶，33 MHz，32 bit の PCI バスを備え，オペレーティングシステムとして Linux2.4.20 が稼働している．また，サーバ計算機とクライアント計算機は 1000Base-TX のスイッチを経由して接続されている．

表 2 にスループットの計測結果，**表 3** に通信時間の計測結果を示す．同時接続数が 1，2 と小さい場合には，フィルタリングを行ったことによる若干の性能低下があり，スループットが 7.7%から 9.0%程度低下し，通信時間が 1.4%から 7.6%程度増加している．これは，同時接続数が少ないためサーバの CPU 負荷が軽く，フィルタリングのオーバヘッドが相対的に大きくなっているためだと思われる．同時接続数が 4 以上になると，サーバの CPU 負荷が高まりフィルタリングによるオーバヘッドが相対的に小さくなっている．これはフィルタリングのオーバヘッドがボトルネックとはならないことを示唆している．同時接続数を 4，8，16 と増加させると，スループットは一定に保たれているものの，通信時間が著しく増大しており，サーバの CPU が過負荷状態にあることが推察される．

本実験では，TCP ストリーム・フィルタリングを用いてもそのオーバヘッドは 10%に満たず，そのオーバヘッドが観測されるのもサーバの負荷が軽い状況に限られている．サーバの負荷が大きい状況でも，TCP ストリーム・フィルタリングが性能上のボトルネックとなることはなく，TCP ストリーム・フィルタリングを用いるオーバヘッドは小さい．

5. 関 連 研 究

5.1 パケットフィルタ

TCP ストリーム・フィルタリングは，アプリケーション層でのフィルタリングを行う仕組みであり，ネットワーク層でのフィルタリングとは異なった仕組みである．ネットワーク層でフィルタリングを行う機構には，パケットフィルタと呼ぶ手法がある．CMU/Stanford packet filter（CSPF）[11] では，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，ヘッダ情報などを用いてパケットのフィルタリングを行う．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）[12] や，IP パケットを対象とした IP フィルタ[13)~15)] は広く利用されている．

パケットフィルタはネットワーク層を対象としているため，TCP ストリームに対するフィルタリングには適用しにくい．まず第 1 に，フィルタリング・ルールを低レベルな言語で記述する必要があり，TCP ストリーム・フィルタリングに必要な文字列処理を記述することが難しい．第 2 に，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．第 3 に，フィルタリング・エンジンがインタプリタとして実現されているため，TCP ストリームのような複雑なフィルタリングではそのオーバヘッドが無視できない．

パケットフィルタの効率を改善する研究には，複数フィルタの重複部分を 1 つにまとめる Mach Packet Filter（MPF）[16]，プロトコル固有のパターンマッチを用いる PATHFINDER [17]，動的コード生成による実行時情報を利用した最適化を行う Dynamic Packet Filter（DPF）[18] などがある．

フィルタリング・ルールの記述を高レベル言語で可能にした研究も多い．文献 19) では文脈自由言語を用いたルール記述の方式を提案している．BPF+ [20] では，宣言的な言語によるルールの記述を可能としている．また，ネットワーク・モニタリングの実装が容易になるように BPF を拡張した xPF [21] などがある．しかし，これらはいずれもネットワーク層を対象としており，アプリケーション層でのフィルタリングには適さない．

Stateful Packet Filter [22] や Dynamic packet filter [23] と呼ばれる手法では，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．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・フィルタリングは実現できない．

Stateful Inspection[24] や Deep Packet Inspection[25] 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る．そのため，バイナリが含まれる HTTP ヘッダや極端にサイズの大きい HTTP ヘッダを破棄するなど，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，本論文で提案する TCP ストリーム・フィルタリングとは異なり，汎用的なルール記述ができず，あらかじめ想定された特定の攻撃に対してのみ有効である．

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）とは，プロキシ（proxy）と呼ぶユーザ・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う仕組みである[26)~28)]．

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・ストリームにさまざまな処理を行うことができ，原理上，TCP ストリーム・フィルタリングと同等の機能を実現できる．しかし，tsfrule 言語などの支援がないため，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．フィルタリング・ルール記述用の専用言語を用いる Firmato[29] や，プロキシの実装を支援する TIS toolkit[30] などがあるが，TCP ストリーム・フィルタリングと同等の機能を実現したものではない．

5.3 ネットワーク侵入検知

TCP ストリーム・フィルタリングは，正しいメッセージのやりとりを定義し，それに反するメッセージを検出する．それに対し，ネットワーク侵入検知システム（NIDS）[31] では攻撃メッセージの特徴を何らかの形で把握し，その特徴に合致するメッセージを検出しようとする．そのため，両者では検出しやすい攻撃のパターンが相補的な関係にあり，併用の効果が大きいと期待される．

ネットワーク侵入検知システムの多くは，シグネチャ・マッチング[32)~34)] という方式を用いて攻撃メッセージの検出を行っている．ここでいうシグネチャとは，攻撃メッセージに含まれる特徴的なバイト列を指す．シグネチャ・マッチングを行う侵入検知システムは，既知の攻撃を特徴づけるシグネチャのデータベースを持ち，受信メッセージ中にそれらのシグネチャが含まれているかどうか検査する．そのため，攻撃メッセージが変化するポリモルフィック型の攻撃や未知の攻撃の検出は難しい．シグネチャ・マッチングを拡張した手法に，プロトコル状態などのコンテキストを採り入れた手法[35] や，N-Code という言語によりシグネチャを記述できる NFR[36] などがある．

イベントによる高水準のポリシ記述を可能とした Bro[33] では，TCP ストリームの解析を行い，たとえば HTTP のリクエストに対して http_request などのイベントを生成し，スクリプト言語でポリシを記述できる．しかし，データ・ストリームからイベントを生成する部分は，アプリケーション層プロトコルごとに C 言語を用いて作成しなければならない．

Sheild[37] では，既知の脆弱性に対する攻撃を防ぐことを目的に，攻撃対象となるサーバの脆弱性を認識し，この脆弱性を突く攻撃を排除するフィルタを記述する．フィルタの記述には専用言語を使用し，プロトコルの状態遷移をトレースしながら攻撃パターンを検出する．しかし，Sheild では，プロトコル規約のうち既知の脆弱性に関連した部分しか記述しておらず，未知の脆弱性に対する攻撃を防ぐことは難しい．

6. ま と め

インターネット・サーバに対する不正攻撃を防止する手法として，TCP ストリーム・フィルタリングという手法を提案しその実現方法を示した．不正攻撃を仕掛ける攻撃メッセージは，アプリケーション層プロトコルの規約や運用上の慣例に反していることが多い．TCP ストリーム・フィルタリングはこの点に着目し，管理者などが定めた “正しい” プロトコルに反するメッセージを破棄する．“正しい” プロトコルの定義を容易にするため tsfrule 言語と呼ぶ宣言的な専用言語を用意し，新しいプロトコルへの対応や運用サイトに固有のルールの適用を容易にしている．フィルタリングを行うオーバヘッドは十分に小さく，Apache ウェブサーバを用いた実験ではたかだか 10%程度のオーバヘッドであった．

TCP ストリーム・フィルタリングをネットワーク侵入検知システムと統合すれば，より安全性の高いサーバを構築することができる．これは，ネットワーク侵入検知システムでは検出できない攻撃でも，TCP ストリーム・フィルタリングで検出できる場合があったり，その逆もありうるからである．また，TCP ストリーム・フィルタリングで検出可能な攻撃は，ネットワーク侵入検知システムで検出する必要はなく，ネットワーク侵入検知システムの負荷を低減させることもできる．今後，ネットワーク侵入検知システムと TCP ストリーム・フィルタリングの統合による安全性の向

上について検証していきたい．

参 考 文 献

1) Cowan, C., Pu, C., Maier, D., Hinton, H., Bakke, P., Beattie, S., Grier, A., Wagle, P. and Zhang, Q.: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks, *USENIX Security Conference*, pp.63–77 (1998).
2) Newsham, T.: Format String Attacks. http://www.guardent.com/docs/FormatString.PDF
3) SecurityFocus: Multiple Vendor C Library realpath() Off-By-One Buffer Overflow Vulnerability. http://www.securityfocus.com/bid/8315/info/
4) SecurityFocus: Orenosv HTTP/FTP Server HTTP GET Denial of Service Vulnerabillity. http://www.securityfocus.com/bid/10420/info/
5) Shankar, U., Talwar, K., Foster, J.S. and Wagner, D.: Detecting Format String Vulnerabilities with Type Qualifiers, *USENIX Security Symposium* (2001).
6) Cowan, C., Barringer, M., Beattie, S. and Kroah-Hartman, G.: FormatGuard: Automatic Protection From printf Format String Vulnerabilities, *USENIX Security Symposium* (2001).
7) 河野健二：アプリケーション層プロトコルの実現を容易にするフレームワーク，情報処理学会論文誌：プログラミング，Vol.44, No.SIG 2 (PRO16), pp.25–35 (2003).
8) Internet Security System: POP3 USER overflow. http://www.iss.net/security_center/advice/Intrusions/2000701/
9) SecurityFocus: Magic Winmail Server USER POP3 Command Format String Vulnerability. http://www.securityfocus.com/bid/7667/discussion/
10) OpenSSL: The Open Source toolkit for SSL/TLS. http://www.openssl.org/
11) Mogul, J.C., Rashid, R.F. and Accetta, M.J.: The Packer Filter: An Efficient Mechanism for User-level Network Code, *Proc. 11th ACM Symposium on Operating Systems Principles* (*SOSP '87*), pp.39–51 (1987).
12) McCanne, S. and Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture, *Proc. USENIX Winter Technical Conference* (1993).
13) Reed, D.: IP Filter. http://coombs.anu.edu.au/ avalon/
14) H. W., et al.: The netfilter/iptables project. http://www.netfilter.org/
15) Venema, W.: TCP WRAPPER: Network Monitoring, Access Control and Booby Traps, *Proc. UNIX Security III Symposium*, pp.85–92 (1992).
16) Yuhara, M., Bershad, B.N., Maeda, C. and Moss, J.E.B.: Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages, *Proc. USENIX Winter Technical Conference* (1994).
17) Bailey, M.L., Gopal, B., Pagels, M.A., Peterson, L.L. and Sarkar, P.: PATHFINDER: A Pattern-Based Packet Classifier, *Proc. 1st Symposium on Operating Systems Design and Implementation* (*OSDI '94*), pp.115–123 (1994).
18) Engler, D.R. and Kaashoek, M.F.: DPF: fast, flexible message demultiplexing using dynamic code generation, *ACM SIGCOMM '96*, pp.53–59 (1996).
19) Jayaram, M. and Cytron, R.K.: Efficient Demultiplexing of Network Packets by Automatic Parsing, *Proc. Workshop on Compiler Support for System Software* (*WCSSS*) (1996).
20) Begel, A., McCanne, S. and Graham, S.L.: BPF+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture, *ACM SIGCOMM '96*, pp.123–134 (1999).
21) Ioannidis, S., Anagnostakis, K.G., Ioannidis, J. and Keromytis, A.D.: xPF: Packet Filtering for Low-Cost Network Monitoring, *Proc. IEEE Workshop on High-Performance Switching and Routing* (*HPSR*), pp.121–126 (2002).
22) Hartmeier, D.: Design and Performance of the OpenBSD Stateful Packet Filter (pf), *Proc. FREENIX Track: 2002 USENIX Annual Technical Conference* (*FREENIX '02*) (2002).
23) Julkunen, H. and Chow, C.E.: Enhance Network Security with Dynamic Packet Filter, *7th International Conference on Computer Communications and Networks* (*IC3N'98*) (1998).
24) Check Point Software Technologies Ltd.: Stateful Inspection Technology. http://www.checkpoint.com/products/downloads/Stateful_Inspection.pdf
25) Dharmapurikar, S., Krishnamurthy, P., Sproull, T. and Lockwood, J.: Deep Packet Inspection Using Parallel Bloom Filters, *11th Symposium on High Performance Interconnects* (*HOTI'03*), pp.44–53 (2003).
26) Avolio, F.M. and Ranum, M.J.: A Network Perimeter with Secure External Access, *Proc. ISOC Symposium on Network and Distributed System Security* (1994).
27) Cheswick, B. and Bellovin, S.M.: A DNS Filter and Switch for Packet-filtering Gateways,

Proc. 6th USENIX Security Symposium, pp.15–20 (1996).
28) Leech, M., Ganis, M., Lee, Y., Kuris, R., Koblas, D. and Jones, L.: SOCKS Protocol Version 5, RFC 1928 (1996).
29) Bartal, Y., Mayer, A., Nissim, K. and Wool, A.: Firmato: A Novel Firewall Management Toolkit, *Proc. IEEE Symposium on Security and Privacy*, pp.17–31 (1999).
30) Ranum, M.J. and Avolio, F.M.: A Toolkit and Methods for Internet Firewalls, *Proc. USENIX Summer Technical Conference*, pp.37–44 (1994).
31) Mukherjee, B., Heberlein, L. and Levitt, K.: Network Intrusion Detection, *IEEE Network*, Vol.8, No.3, pp.26–41 (1994).
32) Roesch, M.: Snort — Lightweight Intrusion Detection for Networks, *Systems Administration Conference '99*, pp.7–12 (1999).
33) Paxson, V.: Bro: A System for Detecting Network Intruders in Real-Time, *Computer Networks*, Vol.31, No.23–24, pp.2435–2463 (1999).
34) Jackson, K.: Intrusion Detection System Product Survey, Technical Report LA-UR-99-3883, Los Alamos National Laboratory (1999).
35) Sommer, R. and Paxson, V.: Enhancing Byte-Level Network Intrusion Detection Signatures with Context, *Proc. 10th ACM Conference on Computer and Communications Security* (*CCS '03*), pp.262–271 (2003).
36) Ranum, M.J., Landfield, K., Stolarchuk, M., Sienkiewicz, M., Lambeth, A. and Wal, E.: Implementing a Generalized Tool for Network Monitoring, *Systems Administration Conference '97*, pp.262–271 (1997).
37) Wang, H.J., Guo, C., Simon, D.R. and Zugenmaier, A.: Shield: Vulnerability-Driven Network Filters for Preventing Known Vulnerability Exploits, *ACM SIGCOMM '04* (2004).

付　　録

tsfrule 言語によるプロトコル定義例として，RFC1939 に従った POP3 プロトコルの定義を示す．定義例を簡潔にするため，ここでは最低限必要とされるメッセージのみを定義している．

```
/*
 * TCP/IP Stream Filtering Rule for POP3
 */

protocol pop3;

regexpr ++ = ' ';
regexpr CRLF = '\r\n';

regexpr argument = (['0x21'-'0x7e']+) [<= 40];
regexpr message = (' ' argument)*;

regexpr username = argument;
regexpr passwd = argument;
regexpr num = (['0'-'9']+) [<= 40];

regexpr ok = ('+OK' message CRLF) [<= 512];
regexpr err = ('-ERR' message CRLF) [<= 512];

state authorization {
    :> ok;

    <: "USER" ++ username CRLF -> user_auth;
    | "QUIT" CRLF -> terminate;
}

state user_auth {
    <: "PASS" ++ passwd CRLF -> auth_reply;
     | "QUIT" CRLF -> terminate;
}

state auth_reply {
    :> ok -> transaction;
     | err -> termiate;
}

state transaction {
    <: "STAT" CRLF ->
           { :> '+OK' ++ num ++ num CRLF; }

     | "LIST" CRLF ->
           { :> ok -> scan_listing; }

     | "LIST" ++ num CRLF ->
           { :> '+OK'  ++ num ++ num CRLF;
              | err; }

     | "RETR" ++ num CRLF ->
           { :> ok -> mail;
              | err; }

     | "DELE" ++ num CRLF ->
           { :> ok | err; }

     | ("NOOP" | "RSET") CRLF ->
           { :> ok; }

     | "QUIT" CRLF -> update;
} -> transaction;

stat scan_listing {
    :> num ++ num  CRLF -> scan_listing;
     | '.' CRLF -> transaction;
}

state mail {
    :> ('\n'+ ([^ '\n'] |
            [^ '\r']'\n')* CRLF) [<= 512]
                -> mail;
     | '.' CRLF -> transaction;
}

state update {
    :> ok | err -> terminate;
}
```

(平成 16 年 7 月 23 日受付)
(平成 16 年 11 月 2 日採録)

河野　健二（正会員）

1970 年生．1993 年東京大学理学部情報科学科卒業．1997 年東京大学大学院理学系研究科情報科学専攻博士課程中退，同専攻助手に就任．理学博士．2000 年より電気通信大学情報工学科に勤務．現在，同大学同学科講師．1999 年より 2002 年まで科学技術振興事業団さきがけ研究 21「情報と知」領域・研究員を兼務．オペレーティングシステム，分散・並列システム，プログラミング言語システム等のシステムソフトウェア全般に興味を持つ．IEEE/CS，ACM 各会員．

品川　高廣（正会員）

1974 年生．1998 年東京大学工学部電子工学科卒業．2000 年東京大学大学院理学系研究科情報科学専攻修士課程修了．2003 年同専攻博士課程修了．博士（理学）．2003 年 4 月より東京農工大学大学院共生科学技術研究部助手．オペレーティングシステム等のシステムソフトウェア，セキュリティに興味を持つ．平成 11 年度情報処理学会論文賞受賞．ACM，IEEE/CS，日本ソフトウェア科学会各会員．

ラハト　カビル（学生会員）

1978 年生．2004 年電気通信大学情報工学科卒業．現在，同大学電気通信学研究科情報工学専攻博士前期課程在学中．オペレーティングシステム，ネットワークセキュリティ，分散システムに興味を持つ．