Network Event Recognition for Packet-Mode Surveillance

Karthikeyan Bhargavan
University of Pennsylvania
bkarthik@seas.upenn.edu

Carl A. Gunter
University of Pennsylvania
gunter@cis.upenn.edu

March 2002

Abstract

Surveillance of packet-mode communications can draw on ideas from firewalls and network intrusion detection systems but has features that raise distinct software engineering challenges. We propose an architecture, CSF, for composable separation functions that can enhance privacy, clarity of specifications, and assurance. We introduce a language, NERL, for network event recognition and use it to build an open-source surveillance system, OpenWarrants, based on CSF. We demonstrate how NERL can be used as a basis for formally analyzing privacy protections and how CSF can be used to provide new capabilities within formally-specified privacy policies.

1 Introduction

Use of packet-mode communication technologies is pervasive and rising quickly. Increasing amounts of information, including much of what was once delivered by telephone voice communication, is now passing over packet-mode networks, particularly the Internet. Law Enforcement Agencies (LEAs) have used surveillance based on information from telephone circuit voice communications to gather evidence for a wide range of suspected criminal activities. There is a desire to extend these means of evidence-gathering to packet-mode communications. This has raised a variety of legal and privacy concerns, but it also entails a variety of serious software engineering challenges.

The aim of this paper is to present an architecture, which we call the *Composable Selection Function (CSF)* architecture, for addressing some of these software engineering challenges in the context of the TCP/IP protocol suite used on the Internet. To do so, we exploit ideas from Network Event Recognition (NER), a technique for passive analysis of IP packet streams. In particular, we propose a formal language, the *Network Event Recognition Language (NERL)*, that can be used to precisely define and analyze general multi-layer filters capable of efficiently extracting specific data from packet streams. NERL is derived from the authors' experience with analyzing routing protocol simulation traces [2] and transport layer monitoring infidelities [1]. The CSF architecture is based on four aspects of processing: aggregation (of packets associated with a specified protocol), identification (of the 'owner' of these packets), abstraction (of information to protect privacy), and delivery (of the abstracted information to an LEA or other monitoring party). An additional element of our architecture is a concept of escrowed data, which is held in storage without being abstracted or delivered, but can later be used for further filtering.

We demonstrate NERL and CSF by using them to address one of the more controversial aspects of warrants for packet-mode communications, namely the generalization to packet-mode surveillance of the *pen register and trap and trace devices* for circuit-switched networks. Such devices are used to collect the numbers called (pen register) or calling (trap and trace) a given suspect telephone number. Access to such information can be obtained with fewer approvals than access to the content of the phone calls for the suspect number. Generalizing this concept to packet-mode communications based on IP over the Internet is subtle since analogies between Internet packet streams and circuit-based telephone calls are difficult to make. (We use the term *pen mode* for pen register and trap and trace for packet data from here on.) To illustrate the difficulty, consider an email message from a suspect person to a correspondent. What is the analogy to the phone number of the suspect? Is an IP address, a Network Access Identifier (NAI), an email address, a Public Key Certificate (PKC), a Ethernet address, or something else? Assuming it is an email address, what information should be delivered as part of a pen-mode surveillance order? Should it be the headers of the IP packets used in the email, the header of the email, or something else? The answer is surely 'something else'. We will describe protocols later, but, to see the problem, the header of an email `TO` field may not be the recipient of the message since the destination of an email is determined by the Simple Mail Transfer Protocol (SMTP), not by the email header. Thus a client with a custom Mail Transfer Agent (MTA) can send messages in which the message header is highly misleading.

We do not attempt to address legal issues in any depth in this paper, but instead focus on software engineering challenges. A few brief comments are worth noting, however. Privacy can be enhanced by advances in the software engineering of monitoring systems. Current challenges include the desire to avoid sending large amounts of data to LEAs and trusting them to sort out the data to which they are entitled. Dually, LEAs are concerned about obtaining too much data because cases can be compromised by exclusion of evidence based on a 'fruit of the poison tree' argument. (There are indications [12] that a problem with mis-collection did lead to difficulties for an investigation using an early version of an FBI packet-mode monitoring system.) More advanced monitoring architectures can allow judges to grant finer-tuned warrants to better protect privacy while expediting justified investigations. Recent law also drives better analysis of monitoring architectures. In particular, the USA Patriot Act (passed in October 2001 after the destruction of the World Trade Centers) specifically calls[1] for procedures to deploy and audit pen-mode monitors. The question of how to define and carry out this kind of surveillance is still not fully determined.[2]

In this paper we address the following questions.

1. How does one precisely specify the contents of a surveillance order that covers 'abstracted' information like pen mode?
2. How can a monitoring architecture be made extensible enough to deal with changing requirements and modular enough to be used in conjunction with other monitoring systems and delivery requirements?
3. How can a suitable architecture be used to enhance privacy protections, accountability, and effectiveness by exploiting improved semantic clarity, extensibility, and composability?
4. Is it possible to build an efficient, open-source system to meet legal and other requirements for packet-mode surveillance?

Our discussion is given in six sections. In the second section we provide background on the history of packet-mode surveillance, enumerate some of the challenges involved, and describe the FBI Carnivore system. In the third section we describe the concept of a buffering firewall as a hybrid of stateful filtering firewalls and Network Intrusion Detection Systems (NIDSs). We then describe the CSF architecture. In the fourth section we describe NERL. In the fifth section we describe our OpenWarrants implementation of a email surveillance system with pen mode and escrow capabilities. We also describe how we used SPIN to derive privacy protection deficiencies in existing NIDSs (viewed as surveillance monitors) and prove that these errors do not arise in OpenWarrants. The sixth section provides conclusions. We have included three appendices, one for readers unfamiliar with the Internet email protocol, SMTP, and the other for readers wanting more details about NERL, including examples of most of the language constructs, and a third for readers who want to see details of the recognizers. The paper can be understood without the appendices.

2 Background

2.1 A Little History

In 1994 the US federal government passed a bill known as CALEA[3] calling for public telecommunications carriers to provide LEAs with the ability to carry out certain forms of surveillance. The Telecommunications Industry Associate (TIA) coordinated the creation of a technical standard to be used by the more than 1000 carriers affected by CALEA. This standard, known as J-STD-025 [19] was created in its first version by 1998, but did not fully satisfy the Federal Communication Commission (FCC), which indicated the following concern about privacy protections:

> We find that the approach taken with regard to packet-mode communications in J-STD-025 raises significant technical and privacy concerns. Under this standard, LEAs would

[1] Uniting and Strengthening America by Providing Appropriate Tools Required to Intercept and Obstruct Terrorism (USA Patriot Act) Act, Pub. L. 107-56, 115 Stat. 272 (2001), at 115.

[2] There are also some questions about the constitutionality of the new laws; see `www.research.att.com/~smb/talks/Wiretaps/index.htm` for example.

[3] Communications Assistance for Law Enforcement Act, Pub. L. No. 103-414, 108 Stat. 4279 (1994).

> be provided with both call-identifying information and call content even in cases where a LEA is authorized only to receive call-identifying information (i.e., under a pen register). ... We believe that further efforts can be made to find ways to better protect privacy by providing law enforcement only with the information to which it is lawfully entitled.[4]

The FCC nevertheless accepted the J-STD-025 approach on a tentative basis and invited TIA to study the issue of packet-mode communications and CALEA further and provide a report. The TIA report [18] was completed in September of 2000 after a pair of Joint Experts Meetings (JEMs) convened by TIA. The TIA JEM report raised a number of questions about the feasibility of packet-mode surveillance; we review some of these issues shortly. The second TIA JEM meeting was the forum in which the FBI gave the first public demonstration of its packet-mode surveillance system, Carnivore. This system received considerable public attention, including calls for the release of the system for public review. The FBI allowed the system (including its source code) to be reviewed by the Illinois Institute of Technology Research Institute (IITRI) and Chicago-Kent College of Law. A draft IITRI report was released in November 2000. A number of comments were written on this draft report from legal and technical perspectives.[5] The final version of the IITRI report [17] was released in December 2000. The current version of the TIA standard, J-STD-025-A [20], was completed prior to the JEM report and does not include technical standards for functionality addressing the concerns of the FCC as quoted above. The Department of Justice recently released a guide [6] for CALEA-conformance on packet-mode communication indicating that packet-mode surveillance capabilities must be provided by carriers even in the absence of a technical standard.

2.2 Challenges

It is useful to begin with some of the context of packet-mode surveillance. To monitor a phone conversation roughly involves creating a conference call that adds the LEA to a suspect call. This is illustrated in Figure 1. This is known legally as an *interception* and requires a warrant from a judge. A *pen register* is the lawful acquisition of certain outgoing dialing or signaling information. A *trap and trace* is the lawful acquisition of the originating number of any wire or electronic communication. Authority for pen registers and trap and trace are given in 18 USC 3123 and 50 USC 1842. Access to this information is based on a lower standard than interception. The application of these concepts to packets is controversial, especially when it is interpreted to mean access to the sender and the recipient(s) of an Internet email message. For this paper we will put aside the legal questions about whether the sender and recipient(s) of an email, for instance, are really covered by these laws. We focus on the technical feasibility of building a system that treats them as such. There are at least six problems in monitoring of packet data: fidelity and vantage point, identification and abstraction, standardization and maintenance, efficiency, and encryption.

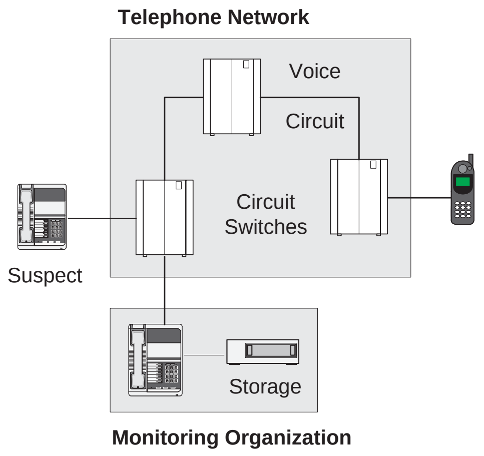

Figure 1: Telephone Wiretaps

Fidelity and Vantage Point. Packet-mode communication breaks a transmission into a collection of packets that are routed independently to their destination. In the Internet, these packets can proceed along different paths and may be dropped, re-ordered, duplicated, or even corrupted. Hence the sequence of packets seen by a monitor may be different form the sequence seen by either of the communicating parties. This is the *fidelity* problem. A closely related problem is that of *vantage point*. Since the packets may travel on different paths, fidelity problems may be exacerbated by missed packets.

Fidelity is a problem more for unreliable protocols than reliable ones. In reliable protocols like TCP a packet is resent if it is not acknowledged. Thus

[4]FCC 99-230, CC Docket No. 97-213, Third Report and Order, at 26-27

[5]The Center for Democracy and Technology (CDT) has a useful reference page at `www.cdt.org/security/carnivore` with links to many of these reports as well as news reports on Carnivore.

a monitor with a vantage point on a bottleneck in the physical or logical network topology between the sender and the receiver can see any acknowledged packet and therefore reconstruct the entire TCP session. However, infidelities can be introduced deliberately too [15]. For instance a trick to confuse monitors is to send a packet with a limit (IP Time To Live bound) that ensures it will be received by the monitor but not the apparent destination and then send another packet with the same sequence number but different contents and adequate resource bound. A monitor not prepared for this trick is likely to record the first packet and discard the second.

Identification and Abstraction. IP packets have source and destination addresses, but these can be misleading about the actual 'owner' of the packet as well as the actual (high level) source and destination. Thus it is a problem to identify the owner and abstract the source and destination. Protocols like DHCP assign IP numbers dynamically to sources and Network Address Translators (NATs) hide actual source addresses behind proxy addresses. Protocols like IP email do not typically send messages directly from the email sender machine to the email recipient machine but rather through a sequence of email relay machines. Thus the destination IP address in the original email dispatch is typically not that of the ultimate recipient, and the IP source address in the final email delivery is typically not that of the initial sender. Moreover, many aspects of IP can be forged or made misleading. We mentioned already the problem that there is some subtlety in defining to whom an email is addressed. A key issue here is that the identity associated with the packet is difficult to determine from any one level of analysis. IP protocols are layered, and this layering is represented by encapsulated packet headers. Thus information about the identity associated with a packet may not be represented at the IP (network) layer but instead at any of several application layers that are meaningful to the endpoints but not necessarily the network. To carry out whatever identification is possible requires a system that can analyze these layers. This challenge was considered at the TIA JEM and the following options were considered: send nothing or all of the packet; send headers or the whole packet; or 'Peel the onion' on a packet to examine multiple layers. Given the protocol layering, 'peeling the onion' is necessary to determine which packets contain the proper data. For example, to determine the parties to whom the suspect sent email, it is useless to look only at the IP headers.

Standardization and Maintenance. A significant problem with any complex specification of information to be collected is the need to manage these specifications in the implementations in hundreds of sites. If an error is detected or a new capability must be added, a tedious standards process followed by an expensive upgrade process may be necessary. Moreover, the TCP/IP protocols are organized to place significant amounts of processing on hosts rather than network elements ('intelligence at the edge'). For example, end-to-end reliability is handled by endpoints with routers performing 'best effort' transmission of packets. If the endpoints of a communication are using a protocol that is unfamiliar to a monitor than it will be unable to interpret high-level features of the protocol to carry out steps like identification and abstraction. It is therefore essential to have an extensible and modular monitor platform that allows changes to be added incrementally with as little fuss as possible.

Efficiency. Routers are designed to do a comparatively simple task very quickly. A surveillance monitor that must 'peel the onion' must perform a far more computationally complex task. It is essentially impossible for a monitor to keep up with the fastest routers. Fortunately monitors can be often be placed near the edge of the network where performance demands are less stringent. Also, as we will discuss later, the technology for surveillance monitors can track advances in firewalls and intrusion detection systems, which have many of the same performance challenges.

Encryption. 'Peeling the onion' is difficult or impossible if encryption is used. In most cases a monitoring system can only log packets that contain encrypted data. Email encryption like PGP and S/MIME is done at application layer so the sender and recipients of the email can be determined without decryption. However, if a client accesses email over a transport layer tunnel (viz. SSL), then nothing will be visible beyond the TCP header. Encryption at network layer using IPSEC ESP in transport mode will leave nothing but the IP header exposed to examination, and in tunnel mode even the (original) IP header is encrypted. Link layer encryption is presumably not a problem since the monitor is assumed to have access to the link layer or a network element used by it.

2.3 Carnivore

Carnivore is a system implemented on behalf of the FBI for the surveillance of packet-mode communications over Ethernet. The description here is based on the IITRI Report, which considered version 1.3.4. A Carnivore-like monitoring system is illustrated in

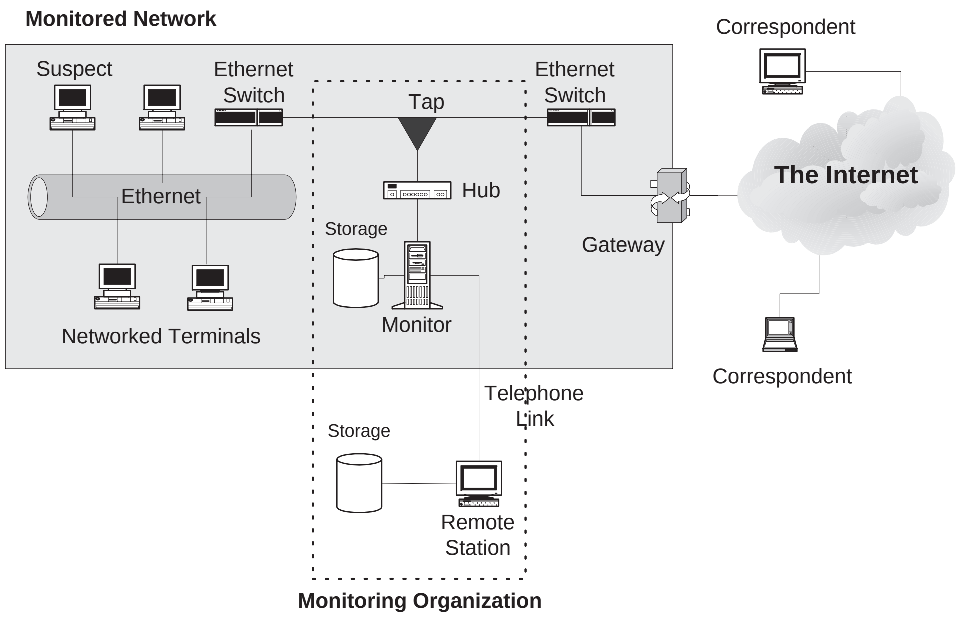

Figure 2: Carnivore-Like Monitoring System

Figure 2. Packets from one or more networked terminals including the one being used by the suspect are routed over an Ethernet link toward other parts of the Internet. A tap is placed in an Ethernet link that may bear some or all of the suspect's packets. All packets on this link are copied through a hub to a monitor machine. The hub is configured so that packets can flow only from the network into the monitor; the monitor cannot affect traffic on the monitored link. The monitor is connected to a remote station by a modem link; remote terminal software is used to control the monitor. The monitor copies packets into local storage. Data of interest is collected from the monitor and stored on the remote station where it can be viewed through a user interface and processed with software for parsing and statistical analysis.

In order to protect privacy and manage the amount of data collected, Carnivore carries out two 'minimizations' (abstractions). First, packets that are not intended for surveillance, such as those sent by parties other than the suspect, are dropped. Second, only a limited view of the data is allowed at the remote station. Certain personnel protocols are followed to help ensure that the data abstraction is not threatened. For instance, a Technically Trained Agent (TTA) who sets up the monitor is not allowed to view the data.

Carnivore provides a variety of modes for data collection, including data from FTP, SMTP (email), HTTP (web) protocols identified as belonging to the suspect by fixed IP address, an address dynamically assigned by Radius or DHCP, and Ethernet MAC address. For each of TCP, UDP, and ICMP there is an option for pen mode collection. For example, if email is collected in pen mode using SMTP (TCP port 25) and POP3 (TCP port 110), then Carnivore was tested to collect only addresses appearing in the `FROM` and `TO` fields. The IITRI report indicates that Carnivore collects the SMTP `RCPT-TO` fields. Finding ways to specify these details is a central part of our effort in this work. When pen mode is used for HTTP (TCP port 80) Carnivore collects the source and destination IP addresses not the URL or its contents.

3 Architecture

3.1 Buffering Firewalls

Despite the daunting list of problems with packet-mode surveillance, much of the technology needed to do parts of it already exists in products for sale to the public and even open source systems. The IITRI report reviews EtherPeek, a popular monitoring ('sniffer') system from Wild Packets, Inc. EtherPeek is able to collect Ethernet frames from a specific IP address, for example. After the IITRI report, NetworkICE, a vendor of intrusion detection systems, released an open source system called *Alti-*

vore with most of the features of Carnivore. Altivore was built by cutting and pasting functionality from existing code. A number of parties have suggested that the FBI provide an open source version of Carnivore. The FBI declined to do this for a variety of reasons we will not review here.

Firewalls are network elements that act something like a perimeter defense and are deployed at topological choke points on IP internetworks [3]. They are generally classified according to the level at which they process packets: filtering firewalls (network layer), circuit firewalls (transport layer), and application firewalls (application layer). A filtering firewall uses a set of rules for which packet patterns to forward or discard. Firewalls are also classified as *stateless* or *stateful* depending on whether or not they collect information from packets that creates state influencing the processing of future packets. A salient feature of firewalls is that they forward or block screened packets. A stateful filtering firewall does this by examining packet patterns in the context of state created by prior packets. *Network Intrusion Detection Systems (NIDSs)* monitor network traffic for unusual patterns or signatures [7]. They typically create logs and raise alarms if observed traffic triggers rules that aim to identify an attack. These alarms can sometimes be used to stop an attack, perhaps by enlisting the aid of a firewall. NIDSs typically can be configured with rule sets based on patterns associated with common means of attack. A salient feature of NIDSs is that they observe and create logs of packets as they pass and raise alarms if they see undesirable patterns.

A surveillance system like Carnivore is a hybrid of a firewall and a NIDS. It observes and logs traffic like a NIDS, but filters the observed traffic like a firewall to prevent too much information from reaching the LEA or other monitoring organization. The system we describe in this paper is therefore like a new kind of firewall, a *buffering* firewall, in which packets are passively collected as if for analysis and logging, and then forwarded (or not) depending on collection policy.

3.2 CSF Architecture

The TIA JEM report refers to the FBI Carnivore system as a ‘Separation Function’. This is distinguished from the Collection Function in J-STD-025 in the way it prunes information to obtain only authorized content. In order to satisfy the FCC request it might be possible to develop *composable* separation functions that would allow carriers to filter surveillance data before handing them over to the LEA. The challenges described earlier remain daunting, but we would like to push forward the study of this direction by introducing the Composable Separation Function (CSF) architecture. It is illustrated in Figure 3. Filtering

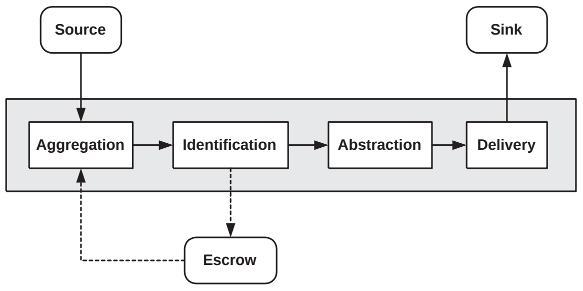

Figure 3: Composable Separation Function Architecture

is conceived as consisting of four conceptual steps of processing. In the first step, *aggregation*, data are collected according to the protocol of which they are a part. A typical aggregation will be packets in a TCP session, but other aggregations like audio transmissions based on a UDP stream or application layer entities like an email message are also likely. In the second step, *identification*, an aggregation is associated with an identity, which may be defined in a number of ways, many of them protocol-specific. In the third step, *abstraction*, an abstraction is performed to remove data that should not be forwarded. In the fourth and final step, *delivery*, the data are rendered in a form that allows it to be passed along to a similar CSF module.

This sequence is *conceptual* and implementations would not typically consist of four separate modules. For example, it may be highly desirable to carry out identification at the same time as aggregation since an aggregate that does not belong to the monitored data can be recognized before they are fully collected. The composition assumption could enable a carrier to run a separation function ‘in front of’ a system like Carnivore or could simply aid the modular construction of surveillance systems. In particular, composability enables a simple concept of *escrow* (see Figure 3) in which data are with-held from immediate delivery but may be delivered later under a new filter.

A particular illustration of CSF based on a stream of packets is given in Figure 4. In the first column are all packets from the packet stream Source. In the second column are the packets that have passed the aggregation and identification steps, here called the *warranted* packets. In this case it is all packets of Alice. In the third column are packets that are directed to escrow. In the fourth column are the

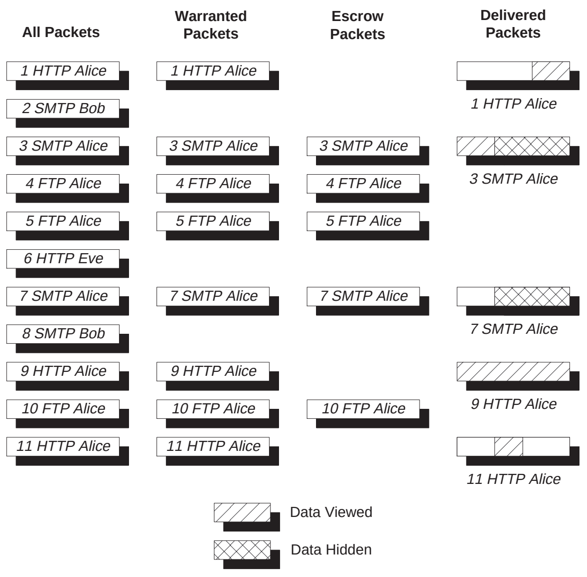

Figure 4: First Stage Packet Processing

packets that are delivered to the Sink. These packets have been modified so that some data are removed. This may consist of simply not delivering pieces or by otherwise 'blanking out' those portions. The recipient may choose to apply its own abstraction and only view portions of the delivered packets.

At a later point, if it is requested that all of the SMTP (aggregate) / Alice (identity) traffic be supplied, then all of the escrow data can be routed back through the filters as illustrated in Figure 5.

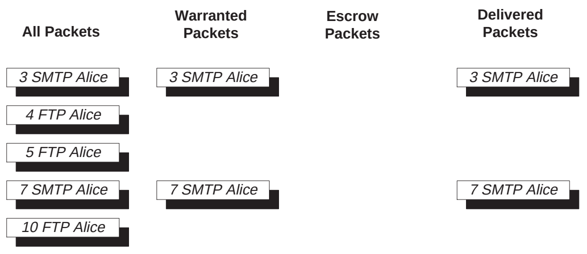

Figure 5: Packet Processing from Escrow

4 NERL

NERL is a language for programming network protocol monitors. Network protocols involve interactions between two (or more) entities that exchange data in a pre-arranged format. A protocol monitor has three tasks: it collects the data exchanged between entities, parses the data to identify the protocol events that they represent, and follows the protocol state machines at the two ends to reconstruct the stage and result of the interchange. For instance, SMTP [14, 10] is a protocol for transferring email messages from a client to a server. The data transferred in an SMTP session consists of client commands and server responses, encoded as specially formatted strings. A monitor must collect these strings, parse them into commands and responses, and follow the client and server state machines to find out which emails and which recipients have been accepted by the server.

Collecting data exchanged by a protocol is not trivial, because network protocols are layered. For instance, SMTP uses a TCP stream to transfer its formatted strings reliably across the network. TCP in turn splits each string into datagrams, which are delivered by IP. To collect SMTP data, we must recognize IP packets, and then TCP streams. Therefore a realistic monitoring setup must have a stack of monitors, one for each protocol layer, up to the protocol of interest.

Parsing protocol data is a simple but tedious and error-prone task. For instance, packet formats have to deal with endian-ness issues of storing 32-bit values, which causes a lot of errors. We advocate using languages such as PacketTypes [11] that can be used for automatically generating parsers from packet specifications. However, for string formatted data, such as the commands used in SMTP, it is most efficient to write the parser in C using string matching libraries. Such parsers are typically short functions, but they must still be inspected carefully for errors.

The bulk of the protocol monitoring effort is in reconstructing the state at the protocol participants, in order to extract meaningful information from the captured protocol data. NERL programs consist of *recognizer* modules that carry out this analysis. Each protocol event recognizer takes input events that trigger a state machine. When a high-level event is recognized, it is triggered as an output event. A NERL program composes several layers of recognizers, feeding outputs of one layer to the inputs of the one above it. In addition to recognizer modules, it may include parser modules that impart additional structure to captured events.

The SMTP recognizer takes as input several command events, such as `Hello` and `MailFrom`, and response events such as `ResponseOk` and `ResponseErr`. (See Figure 10 in Appendix A for the full SMTP state machine.) State variables store the `sender` and `recipients`' email addresses, and the email message `data` sent to the server. A typical SMTP session may have up to 10 intermediate states; the current state is stored in an integer variable (`state`). When a sender and recipient are accepted by the server, a high-level event `EnvelopeAccepted` is generated. Similarly, when an email is successfully sent, the high-level event `MailAccepted` is generated. This is represented in NERL by the following event definition:

```
event MailAccepted = ResponseOk OccurredWhen
                         (state == DATAEND)
      WithAttributes {
          MailAccepted.envelope.from = sender;
          MailAccepted.envelope.to   = receiver;
          MailAccepted.message = data
      }
```

Here, the `OccurredWhen` construct serves to restrict the response event based on the current state. The `WithAttributes` construct attaches attribute values to the `MailAccepted` event. Input events such as client commands trigger state transitions. For instance, the `MailFrom` event triggers the following state transition.

```
MailFrom -> { state = MAIL;
              sender = MailFrom.address }
```

An overview of NERL with other examples appears in Appendix B.

The NERL language suite comes with a compiler that translates NERL recognizers into a hierarchy of C monitoring functions. These functions are composed with protocol parsers, and executed on packet traces to recognize high-level protocol events. The NERL language suite contains a base library of recognizer code (derived from open source software) for the more commonly used protocols, such as IP and TCP. New protocol recognizers can be built on top of these and added to the base library.

It is important to check that recognizers are correct, especially when we compose a long stack of them. The NERL suite provides a number of tools and strategies to ensure that recognizers have the intended semantics. For one, the NERL syntax makes it simple to generate a recognizer from a protocol standard specification. In addition, NERL is strongly typed. The type-checker looks for simple errors such as undeclared variables, arithmetic operations on non-integers, and array indexing violations.

NERL recognizers can also be translated to Promela [8] and model-checked by SPIN [9]. SPIN is a powerful verification tool and can be used to check that the recognizer behaves correctly for a wide range of inputs. For instance, we can check that the SMTP recognizer, when composed with an SMTP client and server, correctly captures all emails sent between them, for all possible client-server interactions.

In addition, we are in the process of developing transformation tools for NERL that incorporate the algorithms in [1] to address fidelity and vantage point issues.

5 OpenWarrants

In this section, we describe OpenWarrants, our realization of a composable separation function for packet-mode surveillance. Given a specification of data that is to be collected, called a *warrant*, the OpenWarrants system uses NERL recognizers to filter a packet stream and deliver exactly the warranted data. OpenWarrants currently handles email warrants.

OpenWarrants sits on a box between the monitored network (the source) and the monitoring party (the sink) and acts like a buffering firewall: it stores packets, analyzes them, and lets them through only when they are determined to be covered by a warrant. The operational setup is shown in Figure 6. Packets are

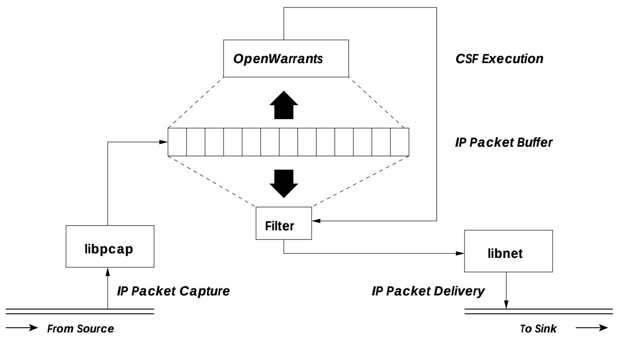

Figure 6: OpenWarrants Setup

captured using the `libpcap` (`tcpdump.org`) library and stored in a packet buffer. The OpenWarrants module analyzes this buffer and informs the filter to let through packets covered by the warrant. The `libnet` (`libnet.sourceforge.net`) library is used to deliver the packets to the network on the other side. Escrow is handled by another filter, also managed by the OpenWarrants analysis module, and is placed into a file. We do not implement any capability for hiding parts of packets.

To execute OpenWarrants, the user must specify an email warrant that consists of several pieces of information.

- Aggregation Level: High-level events of interest, such as SMTP messages, or Internet Message Headers, and NERL modules to recognize them.

- Identification: A NERL recognizer that identifies emails covered by the warrant.
- Abstraction: A NERL recognizer that describes the portions of emails that must be sent through to the sink.
- Escrow: Whether identified emails should be saved for later analysis.

Specifying two levels of aggregation results in four NERL recognizers as illustrated in Figure 7. In the rest of this section, we describe each of these components one by one.

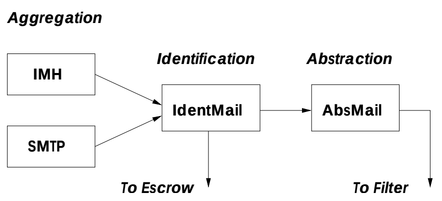

Figure 7: OpenWarrants Components

5.1 Aggregation

To aggregate SMTP data, or Internet Message Headers [5, 16], we write NERL recognizers for these protocols. Because of the layered nature of network protocols, we need to identify protocol events at each layer as shown in Figure 10. Large IP packets in the Internet are sometimes broken down into *fragments*, which are reassembled at the destination. An accurate IP monitor must therefore implement a state machine that collects and reassembles IP fragments, to generate an IP packet event. TCP monitors must similarly keep track of sequence numbers and acknowledgments to reconstruct the TCP data sent. The NERL suite already contains recognizer code for the IP and TCP protocols.

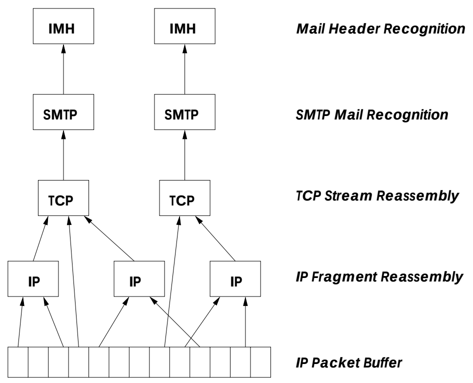

Figure 8: Aggregation: Recognizing Emails

For OpenWarrants, we need to add recognizer modules for SMTP and Internet Message Headers (IMH). The SMTP recognizer collects SMTP commands sent across a TCP session, reconstructs the SMTP dialogue, and recognizes successful email transmissions. It then produces high-level events such as `EnvelopeSent`, `EnvelopeAccepted`, `MailSent`, and `MailAccepted`, indicating different events in the SMTP dialogue. The IMH modules parse Internet message headers, producing `IMHMessage` events that contain header fields as attributes.

5.2 Identification

The NERL identification module IdentMail specifies which email events are covered by the warrant. IdentMail takes as input the events produced by the SMTP and IMH recognizers, and produces the `IdentEnvelope`, `IdentHeader`, and `IdentMail` events when the corresponding input event is covered by the warrant. For instance, the `IdentMail` event may identify `joe@foo.com`'s email by looking at the message header fields and the SMTP envelope. The following rule accepts messages where `joe@foo.com` sends or receives the message or appears in `TO`, `CC`, or `FROM` header fields.

```
event IdentMail = (MailAccepted &
                   IMHMessage) OccurredWhen
    (MailAccepted.envelope.from ==
                                 "joe@foo.com") ||
    (MailAccepted.envelope.to == "joe@foo.com") ||
    (IMHMessage.header.to == "joe@foo.com") ||
    (IMHMessage.header.cc == "joe@foo.com") ||
    (IMHMessage.header.from == "joe@foo.com")
```

The user is responsible for configuring the identification module based on the intent and range of the desired warrant. To simplify matters this code could be produced automatically from a high-level GUI or specification language where it would be described by a rule like 'email for `joe@foo.com`'. NERL helps a technical person determine what this really *means*.

5.3 Abstraction

We write the Abstraction module as another NERL recognizer that takes `IdentMail`, `IdentEnvelope` and `IdentHeader` events from the identification module and removes information not covered by the warrant. For instance, if the warrant only covers the `TO` and `FROM` fields in message headers, the `AbsMail` event is defined as follows.

```
event AbsMail = IdentHeader
       WithAttributes {
           Absmail.to = IdentHeader.to;
           Absmail.from = IdentHeader.from
       }
```

5.4 Implementation details

Finally, the `AbsMail` event needs to be connected to the Filter module to deliver the packets covered by the warrant. OpenWarrants links every high-level event at run-time with the packet events that caused it. As events propagate along the NERL recognizers generating higher-layer events, this packet information is carried along. Eventually the packets linked to the `AbsMail` event are communicated to the filter, which sends them through for delivery. All other packets are dropped. If the warrant requires Escrow, then the packets linked to the `IdentMail` event are saved into a file. Escrow data is sensitive and must be protected by encryption.

The NERL recognizers described in this section for aggregation, identification, and abstraction are composed and translated to a C monitoring program by the NERL compiler. When this program is executed on SMTP packet traces it collects warranted packets.

5.5 Design and Analysis of Recognizers

Designing recognizer modules is the subtle part of this kind of surveillance monitor. To study our ability to do this correctly in OpenWarrants we looked for open source information about SMTP recognizers in NIDSs. We were unable to find rules for reconstructing SMTP messages in Snort (`snort.org`), the most popular open source NIDS. As mentioned before, Altivore provides filter rules written in C and claims to imitate Carnivore. We also tried to conjecture the kind of rule used in a NIDS like BlackICE, based on survey reports [7] and online documentation. We coded these in NERL and they appear in Appendix C. Recognizer `A` is a transcription of the rule from Altivore for capturing complete emails with no recognition of message headers. Recognizer `N` is a more sophisticated stateful analysis like a NIDS might use. Recognizer `O` is the corresponding module from OpenWarrants. Each recognizer analyzes SMTP message events and attempts to identify emails associated with a suspect.

We translated these three NERL recognizers to Promela and analyzed them using the SPIN model-checker. The SPIN model has three processes: an SMTP client, an SMTP server, and the translated recognizer. There are two users in the system: the suspect S and another user U. The client attempts to deliver a number of emails to the server. Each email can be addressed from S or U to one or both of S and U. The recognizer attempts to capture emails that are sent to or from S. The Linear Temporal Logic (LTL) property we checked asserted that, for all client-server interactions in SPIN, the recognizer module never captures emails from U to U. Recognizer `A` fails and SPIN produces a counter-example: `A` does not correctly handle the case when two `MailFrom` commands are issued by the client. It captures the second email even if it is from U to U. A message sequence chart produced by SPIN for this counter-example appears in Figure 11 in Appendix C. To find this error, SPIN analyzed 871 states and 3135 transitions to produce a counter-example with 12 message exchanges. This represents the least number of messages necessary to demonstrate a violation of the LTL property. We then attempted the same proof for recognizer `N`. Again SPIN provides a counter-example: when a `Data` command results in an error response from the server, `N` fails to notice this event and captures the next message sent, even if it is from U to U. A message sequence chart for this counter-example appears in Figure 12 in Appendix C. The SPIN counter-example has 16 messages and was found after analyzing 1610 states and 9897 transitions. Again, this represents the least number of messages necessary to demonstrate a violation. This does not mean that the NIDS rule is incorrect since it was designed to protect the server not the privacy of users. Finally, we checked the property for the OpenWarrants recognizer `O`. SPIN model-checked this recognizer and found no errors; it analyzed 2330 states and 18,689 transitions to reach this conclusion.

6 Conclusions

We have introduced a new architecture for packet-mode surveillance and developed a prototype based on this architecture for monitoring SMTP. Our system uses a new language that is capable of precise descriptions of packets to be monitored. Our architecture and implementation provide improved modularity and a novel escrow feature that exploits this modularity. We have also demonstrated the value of more formal treatment of surveillance filters to improve assurance of privacy protections. Our work partially addresses concerns with identification, standardization, and maintenance as described in 2.2. Although we were not able to complete performance experiments for OpenWarrants in time for the deadline for this paper, we have tested the performance of NERL on the analysis of some complex routing protocol conditions. Monitors derived from the NERL specifications could process 10,000 packets per second while maintaining state for 50 routing tables. This can be compared to the rates sustained by common packet capture modules, which have been measured to operate at around 50,000 packets per second [1].

Our analysis techniques can complement system-level checks like those run by IITRI on Carnivore. We were able to exhaustively cover thousands of cases for the email content monitoring scenario, compared to couple of tests in the IITRI report that found no errors. While system-level tests are indispensable, formal analysis can add assurance guarantees to what they are likely to establish.

References

[1] Karthikeyan Bhargavan, Satish Chandra, Peter J. McCann, and Carl A. Gunter. What packets may come: Automata for network monitoring. In *Proceedings of the Symposium on Principles of Programming Languages (POPL'01)*, pages 206–219. ACM Press, January 2001.

[2] Karthikeyan Bhargavan, Carl A. Gunter, Moonjoo Kim, Insup Lee, Davor Obradovic, Oleg Sokolsky, and Mahesh Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Transactions on Software Engineering*, 28(2):129–145, February 2002.

[3] William R. Cheswick and Steven M. Bellovin. *Firewalls and Internet Security.* Addison-Wesley, 1994.

[4] M. Crispin. Internet Message Access Protocol - Version 4rev1. Technical Report RFC 2060, IETF, 1996.

[5] D. Crocker. Standard for the Format of ARPA Internet Text Messages. Technical Report RFC 822, IETF, 1982.

[6] Department of Justice Federal Bureau of Investigation CALEA Implementation Section. *Flexible Deployment Assistance Guide Second Edition Packet-Mode Communications*, August 2001.

[7] NSS Group. Intrusion detection systems - group test, December 2001.

[8] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991.

[9] Gerard J. Holzmann. The Spin Model Checker. *IEEE Trans. on Software Engineering*, 23(5):279–295, May 1997.

[10] J. Klensin. Simple Mail Transfer Protocol. Technical Report RFC 2821, IETF, 2001.

[11] Peter McCann and Satish Chandra. PacketTypes: Abstract Specification of Network Protocol messages. In *ACM Conference of Special Interest Group on Data Communications (SIGCOMM)*, August 2000.

[12] MSNBC. FBI's Carnivore hunts in a pack, October 2000. `http://zdnet.com/com/2100-11-124798.html`.

[13] J. Myers and M. Rose. Post Office Protocol - Version 3. Technical Report RFC 1939, IETF, 1996.

[14] Jonathan B. Postel. Simple Mail Transfer Protocol. Technical Report RFC 821, IETF, 1982.

[15] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., 1998.

[16] P. Resnick. Internet Message Format. Technical Report RFC 2822, IETF, 2001.

[17] Stephen P. Smith, J. Allen Crider, Jr. Henry Perrit, Mengfen Shyong, Harold Krent, Larry L. Reynolds, and Stephen Mencik. Independent review of the carnivore system - final report. Technical report, IIT Research Institute, December 2000.

[18] TIA TR45. *Report to the Federal Communications Commission on Surveillance of Packet-Mode Technologies*, September 2000.

[19] TIA/EIA/IS-J-STD-025. *Lawfully Authorized Electronic Surveillance*, December 1997.

[20] TIA/EIA/IS-J-STD-025-A. *Lawfully Authorized Electronic Surveillance*, May 2000.

A Internet Mail

Email has for many years been one of the most prevalent services on the Internet. As a result, the Internet Mail Architecture has been closely studied and quite heavily engineered. To use email, a *sender* writes a *message*, addresses it to a *recipient*, and hands it over to a Mail Transport Agent (MTA) such as Sendmail. Once an email has thus entered the mail system, the system becomes responsible for delivering the message to the recipient or returning an error message (via email) to the sender. Senders and recipients are users (or administrators) of mail servers, and are often represented by *email addresses* of the form user@domain, where domain may be a mail server anywhere in the Internet.

The actual transfer of email across the Internet is carried out by the MTAs. An MTA is given a message, and an *envelope* that contains the sender and recipient email addresses,[6] say `S@domainA` and `R@domainB`. The MTA then attempts to deliver the message to the MTA at `domainB`. If there is no direct way to contact the MTA at `domainB`, the message may be delivered to an intermediate relay server that would later forward the email to its destination.

The protocol that runs between MTAs in order to carry out the transfer of email is called the Simple Mail Transport Protocol (SMTP) [14, 10]. SMTP uses a TCP session between two MTAs to deliver multiple emails between them. After the TCP session is established, the SMTP dialogue begins when the sender MTA (the *client*) sends the `HELO` *command* to the recipient MTA (the *server*), which then sends either an Ok or an Error *response*. The client can then send the next command, and wait for the next response and so on.

A typical SMTP session that delivers an email is given in Figure 9 where `C:` indicates client commands, and `S:` indicates server responses. Here, the MTA at domainA is talking to the MTA at domainB, and in the first 3 lines the two MTAs identify their domains. The client then initiates an email delivery by naming the sender (S) in a `MAIL FROM` command, and then naming the recipient (R) in an `RCPT TO` command. The client can name multiple recipients for an email, but only one sender. The server can reject the sender or a recipient by sending an error message (line 9). After sending the envelope information, data delivery begins after the `DATA` command in Line 10 is accepted by the server. Lines 12 to 21 contain the message sent by the client. Data delivery ends at Line 22 with a special line that just has a full stop in it. The client can then send another email or close the session with a `QUIT`.

An SMTP client can also issue a `RSET` command at any time during a transaction to reinitialize the session. Similarly, `HELO` and `MAIL` commands can also be issued at any time to reinitialize the session. The complete SMTP client state machine for these commands is shown in Figure 10. In the diagram, transitions are labeled by actions. A! indicates that A is sent from client to server. B? indicates that B is received at client from server. Each `OK` and `ERR` response may span several lines and must be suitably parsed. The server state machine is symmetric, with the inputs swapped with the outputs.

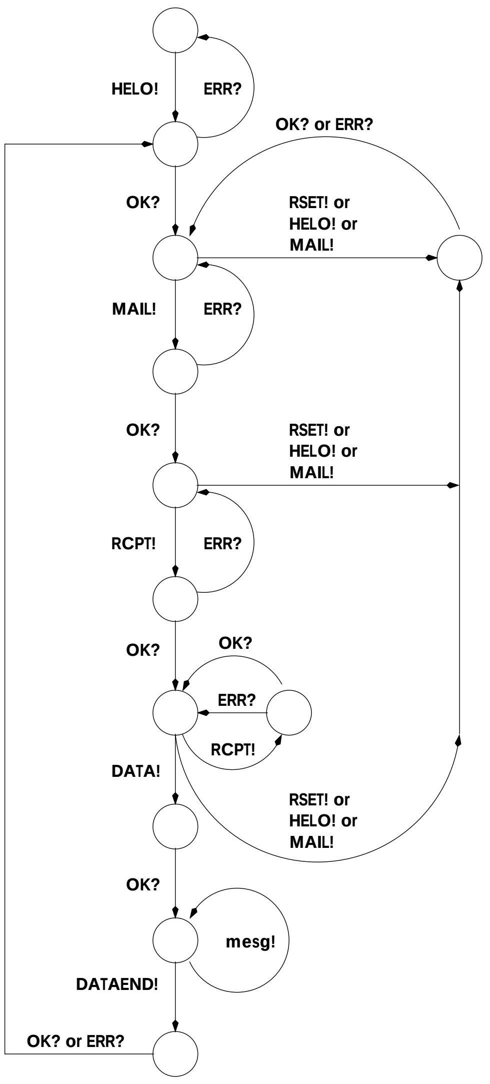

Figure 10: SMTP Client State Machine

[6]Envelopes typically contain the complete path that the message should take to the recipient, or an error message should take back to the sender. This path includes the email addresses of the sender and receiver.

```
 1. S: 220 domainB Simple Mail Transfer Service Ready
 2. C: HELO domainA
 3. S: 250 domainB greets domainA
 4. C: MAIL FROM:<S@domainA>
 5. S: 250 OK
 6. C: RCPT TO:<R@domainB>
 7. S: 250 OK
 8. C: RCPT TO:<Rsec@domainB>
 9. S: 550 No such user here
10. C: DATA
11. S: 354 Start mail input; end with <CRLF>.<CRLF>
12. C: To: "Rob R Roy" <R@domainB>
13. C: From: Sam S Smith <S@domainA>
14. C: Reply-To: "Smith:Personal" <S@personal.domainA>
15. C: Cc: "Roy's Secretary" <Rsec@domainB>,
16. C:     "Smith's Secretary" <Ssec@domainA>
17. C: Subject: Saying Hello
18. C: Date: Fri, 21 Nov 1997 11:00:00 -0600
19. C: Message-ID: <abcd.1234@local.machine.tld>
20. C:
21. C: This is a message just to say hello.
22. C: .
23. S: 250 OK
24. C: QUIT
25. S: 221 domainB Service closing transmission channel
```

Figure 9: Sample SMTP Session

In addition to these commands, SMTP allows the `VRFY`, `EXPN`, and `HELP` commands at any time, to extract information from a server. Newer versions of SMTP also allow service extensions to the standard protocol. We do not consider these commands and extensions here since they do not affect standard message delivery.

SMTP transfers emails between mail servers. When an email is delivered to the MTA at the recipient domain, it is stored in a mailbox owned by the recipient on the mail server. The recipient can then log in to the mail server to check his mailbox. Many users, however, like to read their email on desktop computers that are not powerful enough to act as mail servers. Protocols such as the Post Office Protocol [13], and the Internet Message Access Protocol [4] enable users to access their mailboxes remotely, with facilities to download message headers and bodies, and delete them from the mail server. These protocols work well for incoming email. In order to send email, however, the desktop computer must still use SMTP as a client to hand over messages to the MTA at a mail server.

We have described the Internet mail architecture and the SMTP protocol that delivers email between the MTAs at two mail servers. Internet users, however, never need to be aware of SMTP or even the MTA at their own server. This is because most mail users use Mail User Agents (MUAs), such as Lotus or Outlook, that help them to compose, send, and receive email messages.

MUAs give the user a lot more flexibility in describing the attributes of a message. For instance, the sender S can define who the email is `From` as well as who the recipient should `Reply-to`. S can choose who to address the email `To`, and who should get a copy (`Cc`,`Bcc`). S can even specify the `Subject` of the email. All these attributes are included at the beginning of an email according to a standardized Internet Message Format [5, 16]. Although the complete format is quite involved, and has a number of options, a typical Internet message is as shown in Lines 12 to 21 in Figure 9.

The attributes at the beginning of the message comprise the message *header*, separated from the *body* by an empty line. The MUA is responsible for taking such a message and automatically generating the email envelope by looking at the addresses in the `To`, `Cc`, and `From` fields.[7] It then hands over the complete message and envelope to an MTA to carry out the actual delivery. When the MTA at the recipient's mail

[7][16] describes a number of other fields that may contain addressing information as well.

server receives the message, the recipient R looks at the email through his own MUA, which parses the mail headers and cleanly presents them.

It is important to note, though, that a mail user need not go through an MUA in order to generate an Internet message. S can type the message in a text editor, and directly interact with the MTA to deliver the typed message, according to a specified envelope. So there is no guarantee that the mail headers have any relation to the actual senders or recipients of a message. Moreover, since a user can specify an envelope to an MTA, the sender and even the recipient in the envelope may not exist. Some MTAs will refuse to accept messages if they can determine that the senders or recipients are unknown, but many MTAs do not have enough information to make this decision and will accept the messages anyway.

In addition to the message formats described in this section, further structure can be imposed on the message body, for instance to describe and include attachments (using MIME), and to authenticate or encrypt the message (using S/MIME).

B Overview of NERL

NERL monitors are written as a number of *recognizer* modules, one for each layer in the protocol stack up to the protocol of interest. For instance, to recognize SMTP events, recognizers are written for IP, TCP and SMTP. Each recognizer takes events recognized at the lower layer and follows the protocol state machine to recognize high-level protocol events. For instance, the SMTP state machine is described in the Appendix A. A TCP recognizer must first reconstruct each command and response, and then an SMTP recognizer follows the state machine to recognize events, such as `MailAccepted`. Additional NERL recognizers may be used for application specific processing, such as filtering `MailAccepted` events to recognize `IdentMail` events that indicate emails sent by a suspect.

Each NERL recognizer describes a state machine. It first contains typed declarations of its input and output events, and its local state variables. Variables can contain integers, doubles, bits, bytes, arrays and records. Events have typed attributes: a `timestamp` attribute indicates the time of occurrence, and other attributes, such as `IP_SRC` for IP packets, provide additional information. After the declarations, the recognizer consists of several event definitions, and state transitions. For instance, a small NERL recognizer for the Ping protocol is shown below.

```
Recognizer Ping =
 /* event declarations */
 typedef { time timestamp;int seq } pkt;
 input event pkt Ping_Request;
 input event pkt Ping_Echo;
 output event pkt Ping_Responded;

 /* state variables */
 int last_req;

 /* event definitions */
 event Ping_Responded = Ping_Echo OccurredWhen
                        (Ping_Echo.seq==last_req);

 /* state transitions */
 Ping_Request -> { last_req = Ping_Request.seq };
EndRecognizer
```

Event definitions describe high-level protocol events in terms of input events. For instance, when an SMTP client successfully delivers an email to the server, we say that a high-level event `MailAccepted` has occurred. NERL offers several constructs to help define such events. Events can be filtered by the `OccurredWhen` construct: we say that `Ev OccurredWhen B` occurs when `Ev` occurs and the boolean condition in `B` is true. `B` expresses a condition on the current state and the attributes of event `Ev`. For instance, when a server sends an SMTP Ok in response to a complete email message, the response code must be 250.

```
event MailResponseOk = ResponseOk Occurredwhen
                         (ResponseOk.code == 250)
```

`OccurredWhen` also enables state specific events, by allowing `B` to express conditions on state variables. For instance, the `MailAccepted` event occurs when a response is received in the `DATAEND` state.

```
event MailAccepted = MailResponseOk OccurredWhen
                        (state == DATAEND)
```

Events are correlated by the disjunction (`|`), conjunction (`&`), and implication (`=>`) operators. Note that NERL does not have event negation, because the *absence* of an event is not an event; it is a condition that can only be indicated by another event such as a timer expiry event. These have the usual meaning: `Ev1 | Ev2` occurs when one of the two events occurs, and so on. For instance, an SMTP session must be reinitialized when any of the three input events `RSET`, `HELO`, `MAIL` occur in a transaction:

```
event SessionReInit = RSET | HELO | MAIL
```

Finally, we can use the `WithAttributes` construct to assign attributes to an event. For instance, when a successful email transfer is detected (`MailAccepted`), we may want to tag the envelope and message with the event.

```
event MailAccepted = MailResponseOk OccurredWhen
                       (state == DATAEND)
      WithAttributes {
          MailAccepted.envelope.from = sender;
          MailAccepted.envelope.to  = receiver;
          MailAccepted.message = data
      }
```

State Transitions are triggered by events, and consist of updates to state variables. The usual arithmetic and boolean operations can be performed on state variables. We allow *while* loops for updating arrays, and for complex computation. Conditionals are expressed as *if...then...else* expressions. However, most transitions simply consist of assignment statements. For instance, the transition triggered by the SMTP `MailFrom` command updates the control state of the machine, and stores the sender's email address in a state variable.

```
MailFrom -> { state = MAIL;
              sender = MailFrom.address }
```

NERL is strongly-typed. The NERL type-checker enforces that arithmetic and boolean operators can only be applied to expressions of the appropriate types. It points out the usage of undeclared variables, event attributes, and record fields. For safety, every array indexing operation is guarded by an array-bounds check at run-time. Finally, the type-checker enforces several scoping rules. For instance, in `event x =  Ev WithAttributes S`, the statements in `S` can only refer to event variables that are mentioned in `Ev`. The type-checker is quite useful in finding simple errors that occur while transcribing long state machines.

C SMTP Recognizers

C.1 Altivore

```
Recognizer A
  bit state;
  #define parsing_envelope 0
  #define parsing_message 1
  bool do_filter;
  emailAddr suspect;
  string message;

  input event {emailAddr address} MailFrom;
  input event {emailAddr address} RcptTo;
  input event basic Data;
  input event {string line} DataLine;
  input event basic DataEnd;
  output event {string message} IdentMail;

  Init -> {
     state = parsing_envelope;
     do_filter = false;
     message = "";}

  MailFrom OccurredWhen
     (state == parsing_envelope) &&
     (MailFrom.address == suspect) -> {
       do_filter = true;}

  RcptTo OccurredWhen
     (state == parsing_envelope) &&
     (RcptTo.address == suspect) -> {
       do_filter = true;}

  Data OccurredWhen
     (state == parsing_envelope) &&
     (do_filter == true)-> {
      state = parsing_message;}

  DataLine OccurredWhen
     (state == parsing_message) -> {
      concat(message,DataLine.line);}

  event IdentMail = DataEnd OccurredWhen
     (state == parsing_message)
    WithAttributes {
      IdentMail.message = message}

  DataEnd OccurredWhen
     (state == parsing_message) -> {
      state = parsing_envelope;
      do_filter = false;
      message = ""}
EndRecognizer
```

C.2 NIDS

```
Recognizer N
  int state;
  #define clear 0
  #define mail_done 1
  #define rcpt_done 2
  #define data_done 3
  bool do_filter;
  emailAddr suspect;
  string message;

  input event {emailAddr address} MailFrom;
  input event {emailAddr address} RcptTo;
  input event basic Data;
  input event {string line} DataLine;
  input event basic DataEnd;
  output event {string message} IdentMail;
  event {emailAddr address} MF;
  event {emailAddr address} RT;

  Init -> {
       state = clear;
       do_filter = false;
       message = "";}

  event MF = MailFrom OccurredWhen
     (state != data_done);

  MF -> {
       state = mail_done;
       do_filter = false;}

  MF OccurredWhen
     (MailFrom.address == suspect) -> {
       do_filter = true;}

  event RT = RcptTo OccurredWhen
     (state == mail_done) ||
     (state == rcpt_done);

  RT -> {state = rcpt_done}

  RT OccurredWhen
     (RcptTo.address == suspect) -> {
       do_filter = true;}

  Data OccurredWhen
     (state == rcpt_done) &&
     (do_filter == true) -> {
      state = data_done;}

  Data OccurredWhen
     (state == rcpt_done) &&
     (do_filter == false) -> {
      state = clear;}

  DataLine OccurredWhen
     (state == data_done) -> {
      concat(message,DataLine.line);}

  event IdentMail = DataEnd OccurredWhen
     (state == data_done)
    WithAttributes {
      IdentMail.message = message  }

  DataEnd OccurredWhen
     (state == data_done) -> {
      state = clear;
      message = "";
      do_filter = false  }
EndRecognizer
```

C.3 OpenWarrant

```
Recognizer O
int state;
#define clear 0
#define mail_wait 1
#define mail_done 2
#define rcpt_wait 3
#define rcpt_done 4
#define data_wait 5
#define data_done 6
#define dataend_wait 7

bool some_rcpt;
bool sender_matched;
bool recipient_matched;
bool last_recipient_matched;
emailAddr suspect;
string message;

input event {emailAddr address} MailFrom;
input event {emailAddr address} RcptTo;
input event basic Data;
input event {string line} DataLine;
input event basic DataEnd;
output event {string message} IdentMail;

event {emailAddr address} MF;
event {emailAddr address} RT;


Init -> {
     state = clear;
     sender_matched = false;
     recipient_matched = false;
     last_recipient_matched = false;
     some_rcpt = false;
     message = "";}

event MF = MailFrom OccurredWhen
   (state != data_done);

MF -> {
     state = mail_wait;
     sender_matched = false;
     recipient_matched = false;
     last_recipient_matched = false;
     message = null;}

MF OccurredWhen
   (MF.address == suspect) -> {
     sender_matched = true;}

ResponseOk OccurredWhen
   (state == mail_wait) -> {
    state = mail_done;}

ResponseErr OccurredWhen
   (state == mail_wait) -> {
    state = clear;
    sender_matched = false;}
```

```
event RT = RcptTo OccurredWhen
   (state == mail_done) ||
   (state == rcpt_done);

RT -> {state = rcpt_wait}

RT OccurredWhen
   (RT.address == suspect) -> {
    last_recipient_matched = true;}

ResponseOk OccurredWhen
   (state == rcpt_wait) -> {
    state = rcpt_done;
    some_rcpt = true;
    recipient_matched = (recipient_matched
                     || last_recipient_matched);
    last_recipient_matched = false;}

ResponseErr OccurredWhen
   (state == rcpt_wait) -> {
    if (some_rcpt = false)
     then state = mail_done
     else state = rcpt_done;
    last_recipient_matched = false;}

Data OccurredWhen
   (state == rcpt_done) &&
   ((sender_matched == true) ||
    (recipient_matched == true)) -> {
    state = data_wait;}

Data OccurredWhen
   (state == rcpt_done) &&
   ((sender_matched == false) &&
    (recipient_matched == false)) -> {
     state = clear;
     sender_matched = false;
     recipient_matched = false;
     some_rcpt = false;}

ResponseOk OccurredWhen
   (state == data_wait) -> {
    state = data_done;}

ResponseErr OccurredWhen
   (state == data_wait) -> {
    state = clear;
    sender_matched = false;
    recipient_matched = false;
    some_rcpt = false;}

DataLine OccurredWhen
   (state == data_done) -> {
    concat(message,DataLine.line);}

DataEnd OccurredWhen
   (state == data_done) -> {
    state = dataend_wait;}

  event IdentMail = ResponseOk OccurredWhen
     (state == dataend_wait)
    WithAttributes {
      IdentMail.message = message  }

  (ResponseOk | ResponseErr) OccurredWhen
     (state == dataend_wait) -> {
      state = clear;
      sender_matched = false;
      recipient_matched = false;
      some_rcpt = false;
      message = "";}
EndRecognizer
```

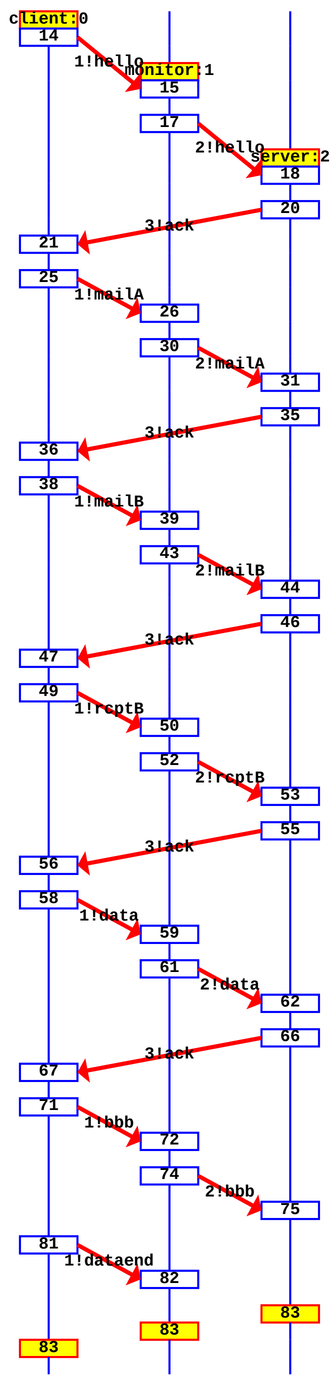

Figure 11: Violation in SMTP Recognizer A

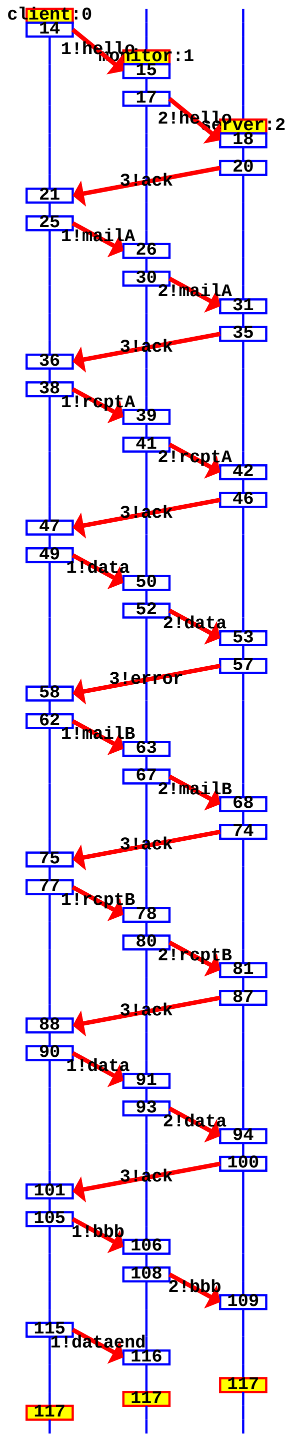

Figure 12: Violation in SMTP Recognizer B