Requirements for a Practical Network Event Recognition Language

Karthikeyan Bhargavan

University of Pennsylvania
bkarthik@seas.upenn.edu

Carl A. Gunter

University of Pennsylvania
gunter@cis.upenn.edu

Abstract

We propose a run-time monitoring and checking architecture for network protocols called Network Event Recognition. Our framework is based on passively monitoring the packet trace produced by a protocol implementation and checking it for properties written in a formal specification language, NERL. In this paper, we describe the design requirements for NERL. We show how the unique requirements of network protocol monitoring impact design and implementation options. Finally we outline our prototype implementation of NERL and discuss two case studies: checking the correctness of network protocol simulations and privacy issues in packet-mode surveillance.

1 Introduction

Runtime monitoring enables a monitor M to observe events created by a program P and check properties of these events as the program executes. If the program P can be instrumented to create a program P' that generates events internal to P and reports them in a convenient form to M, then the monitor M can be simplified to take advantage of this "white box" event recognition. In some circumstances, it is desirable or essential to avoid modification of the original program by the monitor and make do with "black box" analysis of the events produced by the monitored program. In this case the ability to choose what events are monitored falls much more completely on the monitor M, which must now *recognize* the events it aims to monitor from possibly incomplete and low-level events observable from P.

The aim of this paper is to describe some key requirements for event recognition for black box monitoring of packet-mode communication protocols, particularly protocols built on the Internet Protocol (IP). Such protocols produce

This is a preliminary version. The final version will be published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

packets as low-level events but analysis typically focuses on properties derived from collections of packets that are amalgamated to produce high-level events. We argue that a new language is appropriate for this task and describe a collection of requirements for the language. Many of the ideas from runtime monitoring languages can be applied, but our experience shows that new techniques can substantially improve convenience, clarity, and efficiency.

Black box network monitoring is already a popular technique to study protocol behavior after deployment. So far, it has been used in primarily three ways: performance testing, intrusion detection, and surveillance. Performance testing involves placing monitoring agents at various points in the network that collect statistics on the packets that pass through them. These statistics are then correlated to trigger alarms or produce reports when unexpected behavior is encountered. Intrusion detection systems reconstruct protocol sessions and match them against a library of "attack signatures", to see if a malicious user is exploiting known security holes in the network. Finally, surveillance systems capture protocol sessions and simply record the emails or files sent between parties, if they match one of several triggering key-phrases.

We propose to add a fourth runtime verification aspect to network monitoring, which we call Network Event Recognition (NER). For instance, consider the SMTP [Pos82] protocol that transfers emails across the internet. Every email has a "From" address that the sender fills in. In addition, SMTP requires a mail envelope that also contains a "MailFrom" field, which usually has the same address as the From field. Suppose we want to check that these fields do indeed have the same address for all emails in our local network. Then the event that we are interested in is FromMismatch, which is triggered whenever an email's From and the SMTP MailFrom do not match. In an event definition language, such as MEDL [LKK+99], this would be written as

```
event FromMismatch = EmailSeen when
  (EmailSeen.Header.From != EmailSeen.Envelope.MailFrom)
```

Then, we can compile such a definition to a runtime monitor that analyzes a packet trace and produces FromMismatch alarms whenever such an email is seen.

Clearly the above definition still leaves a number of open questions. For example, how is EmailSeen defined? How are the packets in the packet trace read into the monitor, and how are emails reconstructued from individual packets? Is it feasible to run such an analysis on all emails in a network? In this paper, we shall discuss what would be required to write such an event definition, and then what would be required for it to run efficiently.

The paper is presented in eight sections. In the second section we define the concept of Network Event Recognition, including a characterization of the kinds of events that need to be monitored. In the third section we describe four principal requirements for a practical implementation of NER. In the fourth section we describe related work in terms of these requirements. In

the fifth section we describe our prototype design and implementation based on the Network Event Recognition Language (NERL). The sixth and seventh sections provide case studies based on network simulation analysis and packet-mode surveillance respectively. The eighth section concludes.

2 NER Model

Our monitoring model is based on passive, black-box network monitoring. We place one monitoring machine on the same broadcast LAN as the hosts being monitored. The monitor can see only the messages sent between hosts, not the state at each host. We assume that all the interesting IP traffic on the LAN can be captured and fed into an analysis engine. This capture and analysis may be done either on-line, using a packet capture library (such as libpcap), or off-line, where the captured packets are stored in a trace file as is the case in network simulations.

The goal of a protocol monitor is to recognize *protocol events*. At the lowest layer, a protocol event is as simple as *a packet P has been sent from A to B*. Higher layer data like emails, however, are broken up into several packets, and some of them may be resent for reliability. So at higher layers, recognizing complex events like *an email E has been sent from U to V* may involve capturing a number of packets, correlating them and putting the data in these packets together. We call this hierarchical process of recognizing protocol events at different layers *Network Event Recognition (NER)*.

The NER framework can be visualized as in Figure 1. The figure is annotated with terms from the *WRSPM* software artifact model [GGJZ00]. The 'World' W is the operational environment, which includes endpoint activities that spawn network events. The 'Requirements' R are properties expected of the system, such as the requirement that a routing protocol finds paths to destinations. The 'Specification' S denotes the software specification, such as an IETF RFC. The 'Program' P denotes the implementation of the system. The 'Machine' M is the programming platform. These are related by the idea that P is built on M to satisfy S, and S is chosen to ensure R given the operational assumptions in W.

At the bottom in Figure 1 is the interface with the protocol implementation under test, which is running on a device (DUT). In the case that NER is being used to monitor a live network, this interface is the 'wire' on which the monitor is 'sniffing' (listening) for interesting packets. On the other hand, when NER is used to analyze network simulations then the interface between the monitor and the implementation is the simulation trace file.

For each primitive event interface, we need a primitive event capture module that can convert network or simulation trace events into a format that can be understood by the event recognizers. In Figure 1, the IP packet capture module carries out this task, while in simulation analysis a trace-to-event converter parses the simulation trace and generates events. Meta event recogniz-

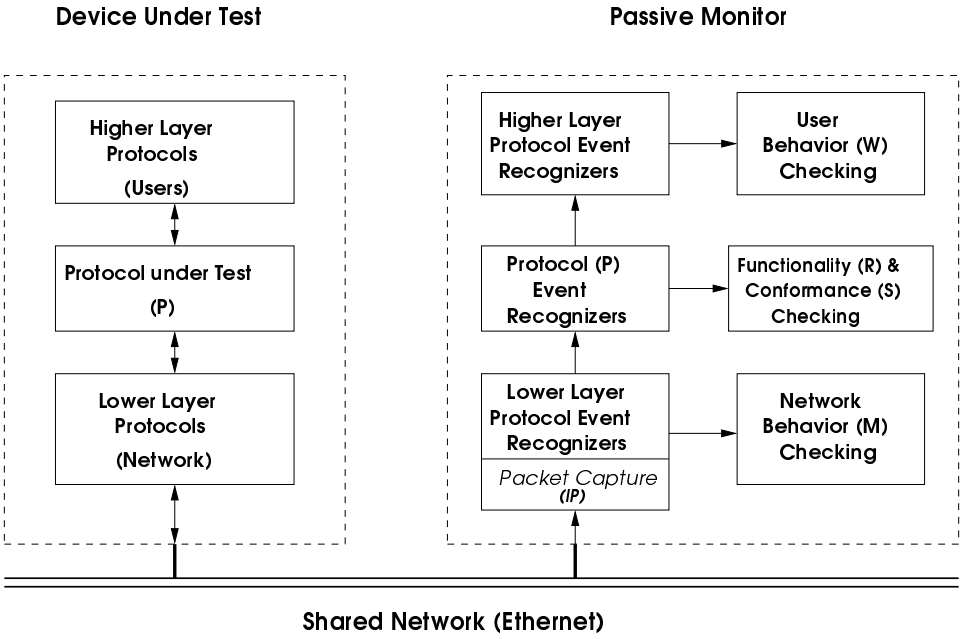

Fig. 1. Live Monitoring using Network Event Recognition

ers carry out the actual analysis of the stream, and check for the monitorable properties of interest for the protocol. Finally, some events are classified as alarms, and must be delivered to the appropriate authorities. For instance, when an intrusion event is recognized, the administrator may need to be notified. Event delivery can range from writing to a log, sending email to an administrator, or sending corrective messages to the device under test itself.

There are two kinds of primitive events of interest:

Primitive Network Events These are identified and delivered by the capture module. For instance, `libpcap` (tcpdump.org) is an open source tool that captures IP packets from Ethernet wires and presents them to requesting programs, and can be used for live network testing (as the IP packet capture tool in Figure 1).

Timer Events Another kind of primitive event is the tick generated by a system clock. Many protocols have specifications and requirements that are time-bound. This makes it necessary for NER to keep track of the system clock to recognize events such as *timeouts* that are often necessary to signify the absence of an event.

For higher-layer events (also called meta-events), there are four main recognition techniques. Some of these have been studied before [LF98] in the context of event management systems.

Filtering Events have attributes. Filtering recognizes events whose attributes satisfy a predicate. For instance, recognize all email-related packets, and throw away the rest.

Correlation Recognizing that events E1 and E2 have both occurred, or that

one of the two has occurred and so on. For instance, recognizing that a TCP packet contains data D as well as ack A.

Aggregation Recognizing that a sequence (or pattern) of events has occurred. For instance, a typical SMTP session involves naming the sender S, naming the recipient R, and sending the email E. These three events can be aggregated into an EmailSent event containing all the information sent.

Abstraction Abstraction is the only technique that *reduces* the information content of events, by enabling the recognition of more general events from particular ones. For instance, having recognized an email E that contains a virus, as event VirusEmail(from,to,E), we may want to abstract away the contents of the email to protect the privacy of the sender, and only propagate the alarm VirusSent(from,to).

When put together, these techniques allow us to mimic the protocol stack, and recognize events layer-by-layer up to the protocol of interest. Note that Aggregation requires that we maintain some state, in order to record that part of the sequence (or pattern) that has been recognized.

3 Requirements

In this section we consider the problem of programming network event recognizers. For instance, to carry out an NER analysis for SMTP, we would write a monitoring program that analyzes captured SMTP packet traces and raises alarms whenever a desired property is violated. This monitoring program may be written in a general purpose programming language, such as C, or in a domain specific language that we choose to design. In either case, what are the programming constructs that we would need to describe the monitoring program? Are there any additional constraints that guide the way we write and implement the monitoring specification?

First, let us consider some general requirements for writing an event recognizer. A network event recognizer analyzes a trace of primitive packet events and reconstructs higher-level protocol events, raising alarm events when anomalies in protocol behavior are detected. The implementation of the event mechanism may be as sophisticated as method invocation in an object-oriented language, or as simple as setting a boolean flag in a language like C. For instance, SMTP packets are defined as TCP packets at port 25. One way to define this is

```
event SMTP_Pkt = TCP_Pkt when
                 (TCP_Pkt.srcport == 25) ||
                 (TCP_Pkt.dstport == 25)
```

This definition may be compiled in terms of a boolean variable `SMTP_Pkt` which is set to true exactly when the condition on the right hand side is true: that is when a TCP packet is seen whose source or destination port is set to

25.

In addition, any monitoring language must keep state about the pattern of events seen so far. A protocol specification, such as SMTP, often comes with detailed state machine description that each participant must follow. So programming an SMTP monitoring program, for instance, involves coding a chunk of this state machine, where the transitions are triggered by events and the state is stored in variables. This means, for instance, that a pure logic such as LTL is probably not well suited to NER, while an EFSM based language such as Promela is better suited.

Apart from programmability, another essential requirement for practical network event recognition is fast execution. When NER is used for live network monitoring, it has to keep up with real network speeds, which can involve data transfer of 100Mbps or more. If we assume that most packets are a few kilobytes in size, NER still needs to process tens of thousands of packets (primitive events) every second. This makes a fast execution environment necessary. For instance, in [BGK+02], we found that our Java implementation of Verisim, while adequate for monitoring 5 nodes at the same time, was too slow for analyzing the large amount of traffic generated by a 50 node network simulation.

The above requirements demand a basic level of expressiveness and efficiency that would be required in many monitoring contexts. In the following sub-sections we describe requirements particular to the network protocol monitoring domain. These requirements will guide our choice of programming language and execution environment for NER.

3.1 Packet Format Representation

In the network monitoring context, the interface between the device under test and the monitor is in terms of packet events. Packets on the network have a number of fields, such as the source and destination, and the format of these fields is particular to the protocol under analysis. For instance, the format of an IP packet is given by the following PacketTypes [MC00] description.

```
nybble := bit[4];
short := bit[16];
long := bit[32];
ipaddress := byte[4];
ipoptions := bytestring;

IP_PDU := {
  nybble      version;
  nybble      ihl;
  byte        tos;
  short       totallength;
  short       identification;
  bit         morefrags;
```

```
    bit         dontfrag;
    bit         unused;
    bit         frag_off[13];
    byte        ttl;
    byte        protocol;
    short       cksum;
    ipaddress   src;
    ipaddress   dest;
    ipoptions   options;
    bytestring  payload;
  } where {
    version#value = 0x04;
    options#numbytes = ihl#value * 4 - 20;
    payload#numbytes = totallength#value - ihl#value * 4;
  }
```

A TCP packet is an IP Packet in which the payload field is *overlayed* (replaced) by another record with fields corresponding to the TCP header and the TCP data. In turn, and SMTP packet is a TCP packet with the TCP data overlayed by the SMTP format, and so on.

Therefore, in order to extract and reconstruct the messages sent and received by a protocol, a network event recognizer must first capture and understand packet events. For a natural representation of packet events, a NER language must allow each event to have attached with it a record of attribute fields that mimics the packet format, such as in the above `IP_PDU` description. This requirement is specific to protocol monitoring; typical runtime verification do not need to deal with such low-level structures and formats.

Suppose a NER language did not allow complex event attributes. In such a language, we would have to define the single IP packet event as multiple events corresponding to each field of interest:

```
event IP_Pkt;
event IP_Pkt_protocol;
event IP_Pkt_src;
event IP_Pkt_dest;
event IP_Pkt_payload;
```

In addition, every meta-event that is constructed from these events would have to represent the attributes in an artificial way as in the following, where the attributes are represented as parameters.

```
event TCP_Pkt[src][dst] = IP_Pkt when
            (src == IP_Pkt_src) &&
            (dst == IP_Pkt_dest) &&
            (IP_Pkt_protocol == TCP)
```

In the case that `src` and `dst` can range over 50 nodes each, the above definition actually defines 2500 different events. Any such unnatural representation of

event attributes can seriously affect the performance of the event recognizer, as was shown in the Verisim case study [BGK+02].

We advocate that events be allowed to have attached attributes with a C-struct-like format. Even structs, however, are often not adequate since packet formats can only be expressed in very powerful type systems such as PacketTypes [MC00]. It would be desirable for a NER language to allow events to have attributes whose types can be described by PacketTypes specifications.

3.2 Layering and Modularity

An important characteristic of network protocols is layering. For instance, mail users send emails, which are broken up into SMTP commands, which are in turn transmitted as TCP segments in IP packets which are sent over the wire. The monitor captures the IP packets; but then it needs to reconstruct TCP streams, and then to extract SMTP emails from these streams. An event recognizer thus consists of several layers of protocol analysis with one or more layers for each protocol itself. This layering makes itself felt in the order in which events are recognized. For instance, if we write

```
event SMTP_Pkt = TCP_Pkt when
          (TCP_Pkt.srcport == 25) ||
          (TCP_Pkt.dstport == 25)
event SMTP_Command = SMTP_Pkt when
          (TCP_Pkt.dstport == 25)
```

then the dependency of the second event upon the first defines a fine layering between them, even though both are protocol events for the same protocol. The more pronounced layers correspond to the TCP and SMTP protocols, as is implied by the `SMTP_Pkt` event, which cannot be executed until the `TCP_Pkt` event has been identified.

The layering between and inside protocols may be expressed simply in terms of event dependencies as shown above. However, this would imply that each monitoring program be a large, flat collection of event descriptions containing events from several protocols. A cleaner way to deal with this would be to write separate monitoring *modules* for each protocol, which could be stacked one above another in the appropriate order.

There is another reason to wish for a modular structure: we will need to write many versions of monitoring specifications for each protocol and we will want to swap one for another.

3.3 Library Compatibility

A network event recognizer must interface with low-level infrastructure that captures and presents packet traces to it. Therefore, a more pragmatic requirement for an NER language is that it must be compatible with these support libraries.

In the case of live network monitoring, packets need to be captured from the network interface, they must be filtered and sent to the appropriate recognizers, and sometimes IP packets and TCP streams must be reconstructed. All these functions are available in the form of C libraries, such as libpcap (tcpdump.org) and tcpflow (www.circlemud.org/~jelson/software/tcpflow). In addition, any support libraries that we write: for reading packet traces, parsing packet formats, maintaining network data structures, and constructing NER events, must be written on top of these libraries in C.

In the case of network simulations, such as those carried out in NS (www.isi.edu/nsnam/ns), the protocol analysis code is written in C++, and on execution produces a simulation trace file that has a format specific to NS. We have written support libraries that parse the NS trace format to construct NER events. This form of text trace analysis is fastest in a language like Perl or C. Similar libraries will have to be written for every possible front-end that we wish to apply NER on.

Therefore, any implementation of a NER language must interface with these support libraries. In general, there must be a mechanism for tying together library calls with the event recognizer program, whatever language the program is written in. Clearly, it would be best if the monitors to be executed were written in C and linked with these libraries during compilation. Indeed, we have found that the execution speed drops markedly if the monitoring program is written in Java and the library interface consists of text files, or text sockets [BGK+02].

3.4 Model Checking

The previous requirements all serve to increase the complexity of network event recognizers. Unlike formulae in simple monitoring logics whose meaning is immediately apparent, network event recognizers will consist of several layers of large state machines with complex events, written with an eye to execution speed and C-library compatibility. It will be a challenge to write these recognizer modules correctly, and to understand what they mean.

For instance, to recognize emails sent on a network, we might need modules for IP, TCP, SMTP, and Internet Mail, layered above each other and passing reconstructed events between them. If the TCP module does not reconstruct text streams as expected by SMTP, then the entire analysis at the Internet Mail layer may be incorrect. In addition to incompatibilities between modules, there may be bugs in the individual modules. While many simple errors can be avoided by strong type checking, when complex state machines need to be encoded subtle errors may creep in. We advocate model-checking the recognizer stack to guard against module bugs and layer incompatibilities. For instance, a model-checker would take the recognizer modules for IP, TCP, SMTP, and Internet Mail, models for mail senders and receivers, and a logical property that says that all incorrect emails between senders and

receivers are (correctly) flagged by the Internet Mail module. While model-checking for a general pupose programming language like C is very difficult, there have been attempts to model-check stylized programs in subsets of these languages [Hol00]. We believe it would be simpler to use a NER language that already has a model-checker for it, or translate our recognizers to such a language.

3.5 Transformations

Monitoring a protocol implementation requires that the complete and correct trace of inputs and outputs to the protocol be available. Often, however, the trace available (or observed) is incomplete and inaccurate, because packets get dropped and re-ordered between the device under test and the monitor [BCMG01]. Suppose a TCP module assumes that it will see all the IP packets between S and R. If in fact some packets are being dropped, and others are not in the right order, the TCP module will be unable to correctly reconstruct the data stream. However, since packets that reach R will cause an acknowledgment to be sent back to S, when this acknowledgment is seen, the monitor can guess that it missed a packet. In this manner, it may be able to reconstruct portions of the data stream, which might be adequate to find errors at the SMTP layer. So in the presence of packet trace inaccuracies, we would like to *transform* the TCP module to use additional information from the trace to do the best it can in reconstructing data streams.

In [BCMG01] we have identified several classes of protocols and monitorable properties for which such transformations are possible and proposed transformation algorithms. So we require that a NER language allow us to perform such transformations, taking recognizer modules and producing new recognizer modules that can be plugged in their place. We note that carrying out transformations on a general purpose programming language, such as C, would be quite difficult, because the transformation needs to take into account the powerful features and idiosyncrasies of the language.

3.6 Examples

In the previous sections we have described what we reqquire from a network event recognition language. Now let us look at some examples of events that we will want to express in such a language. Each of these events is formulated as a failure of a desired property.

Request-Response An SMTP command C was followed by an incorrect response R. This means that the mail server is not following the protocol correctly.

Safety Three nodes running a routing protocol have formed a forwarding loop $N_1 \rightarrow N_2 \rightarrow N_3 \rightarrow N_1$. Such loops in routing protocols are highly undesirable as they cause unnecessary and heavy usage of bandwidth and

computing resources, while being difficult to detect.

Intrusion A host H has sent a sequence of packets $< p_1, \ldots, p_n >$ that matches the signature of a well known attack on a Linux firewall. Such attack signatures are widely and effectively used in network intrusion detection systems.

On the other hand, there are several kinds of events that we will not consider. While some of these may technically be within the scope of Network Event Recognition we do not require our system to be optimized for them. Examples of such events are

Quality of service The number of TCP packets seen between the host-port pairs (S, SP) and (D, DP) in the last T seconds is more than N. Such congestion events are important for network management.

Anomaly The packet traffic sent by host H is different from yesterday. These are used in anomaly-based intrusion detection.

Performance In the network simulation of a mobile wireless network less than half the packets sent were successfully delivered to their destinations. This might indicate poor performance of the routing protocol.

While these kinds of network events are useful in expressing and detecting unexpected errors in protocol implementations, our primary focus is on detecting protocol events that can be explicitly and formally specfied.

4 Related Work

Event recognition and analysis has sbeen studied extensively in several contexts, including network monitoring. The earliest reference to runtime verification for network protocols was in the Overseer project [PF76] that described a monitoring system for a pre-Internet network. The SCAN proposal [SFG+] extended the Overseer idea for Active networks, but was never implemented.

More recently, Network Intrusion Detection Systems [Gro01], that check for malicious user activity in the network, are gaining popularity. Intrusion detection languages, such as Bro [Pax99] are good candidates for network event recognition since they already have efficient implementations and can monitor packet traffic at high speeds. On the other hand, protocol specification languages, such as Promela [Hol91] are very expressive: they allow us to define protocol events and state machines with ease, and often have sophisticated analysis tools. Protocol implementations, however, are typically written in programming languages, such as C, or in specialized languages such as Prolac [KKM99].

An alternative is to look at network event recognition as a testing problem. Our network simulation analysis can be seen as a form of test trace analysis, which is traditionally carried out by specialized languages such as GIL [ORD96], or by fast text analysis languages like Perl (www.perl.com). In

addition, there are several runtime verification logics and languages, such as MEDL [LKK+99], which are geared towards testing programs at run-time.

To pick one of the above languages for NER, we must first consider each of the requirements that we have put forward in the previous section:

- In which languages can we easily express protocol state machines?
  Promela, C, Prolac, Perl, MEDL
- Which languages can be compiled to executable monitors?
  Bro, C, Prolac, GIL, Perl, MEDL
- Which languages have support to handle packet formats?
  Bro, C, Prolac, (primitive support in Promela, Perl)
- In which languages is it easy to express modularity and layering?
  Promela, C, Prolac (to some extent in Perl, MEDL)
- Which languages are compatible with the C libraries for network monitoring and simulation analysis?
  Bro (for libpcap), C, Perl (through C interface)
- Which languages lend themselves to easy analysis and transformation?
  Promela, GIL, MEDL

We note that while C seems to satisfy almost all the requirements, it is only to a certain extent. For instance, one can see that C can be used to express state machines, but it may not be very easy to write them. Similarly, while C functions can be used as modules, the interfaces between them are not very clean. Moreover, C programs are not strongly typed and so are susceptible to simple errors. However, a number of these limitations may be addressed by providing libraries and programming interfaces. Indeed, C is said to have good library compatibility and packet format representation only because the support libraries for monitoring are already written in C. The same would be true for Bro and Prolac if libraries were made available for them. In fact, any of these three languages could be chosen and enhanced with pre-processors, type-checkers and libraries to be effective monitoring languages. However, carrying out analyses and transformations on these languages will not be easy.

On the other hand, there are the logic-based, analyzable languages such as MEDL and Promela, which are easy to write and read. We could extend these languages with efficient compilers, and packet format support. The key issue to consider is the effort involved in modifying these languages for our purpose, compared to the effort required to design a new domain specific language with its own compiler and analysis tools.

5 Prototype Implementation

Our prototype implementation consists of a domain specific *Network Event Recognition Language,* NERL. NERL programs are translated to C, and then are compiled along with the support libraries to generate fast executable monitors.

In addition, NERL programs can be translated to Promela models and model-checked for desired properties using SPIN [Hol91]. The architecture of the implementation is as shown in Figure 2.

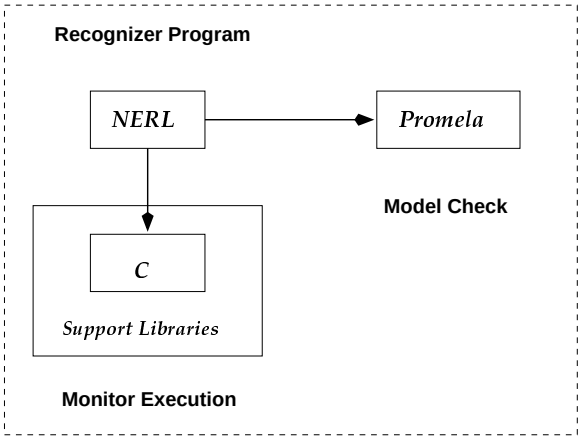

Fig. 2. NERL Implementation

The language NERL is inspired by the runtime verification logic MEDL. MEDL is an event definition language that allows simple state variables to store the pattern of events seen so far. To satisfy our requirements, we need to make several extensions to the language. In NERL, events are allowed to have packet-like attributes, which can be defined using C-struct definitions. A NERL recognizer consists of multiple, layered monitoring modules that communicate through events. Each module runs a state machine that is triggered by input events and produces higher-layer events or alarms. To express state, we include additional data structures such as tables, records, and bytestrings. In this way, NERL is designed to be modular and expressive enough to handle protocol state machines and packet events.

Moreover, several transformations and analyses can be carried out on programs in this specialized language. NERL is strongly typed, and has a type-checker that can check for simple mistakes. In addition, NERL modules can be translated to Promela and model-checked for simple errors. Finally, we argue that it will be much easier to define transformation functions on this restricted language, rather than a full-fledged programming language.

The NERL compiler consists of a parser, type-checker and translator written in OCaml. It parses an event recognizer written using the NERL syntax, type-checks it, and then translates it into a reactive C program that is triggered by input events and prints out output events as they occur. Since the target of the compiler is C, this program can access the support libraries, such as libpcap, satisfying the library interface requirement. The choice of C as a monitoring environment also gives us a performance boost in comparison with languages such as Java.

6 Verifying Routing Protocol Simulations

Verisim is a tool that was designed for the formal analysis of network simulations [BGK+02]. The first version of the tool used MEDL as the event processing engine. The current version, Verisim v2, uses NERL for the event recognition instead.

The case study that we describe in this section is the simulation of the wireless routing protocol AODV. Suppose there are a number of nodes (laptops) with wireless connections in a large convention hall. The wireless range of each node is limited, so a node A may not be able to exchange data with node B. However, if there is a node C between them that is willing to *route* (forward) data between A and B, then A and B can establish a connection. In general, there could be more than one node in the path between A and B. The aim of the AODV routing protocol is to enable such paths to be formed and maintained between nodes that are not within each other's range.

The simulation scenario we shall consider for AODV is from the CMU Monarch group (monarch.cs.cmu.edu). Consider a site of size 1500×300 meters with 50 nodes constantly moving at 20 meters per second. There are 150 data connections transmitting four 64 byte packets every second. Each of the nodes runs AODV, and the protocol attempts to provide paths for all these connections. The simulation is run on the network simulator NS. The simulation and our subsequent analyses were carried out on a dual Pentium-III 550Mhz Xeon processors machine with one gigabyte of memory. The OS was Red Hat Linux 7.2 with the 2.4.9-13 SMP Kernel. We used NS version 2.1b6, and the simulation itself required about 5220 seconds to complete and generated 6,446,316 packet events.

We have earlier tried to analyze this trace using MEDL [BGK+02], which implements event recognizers in Java. A naive effort to use MEDL to check a relatively simple property on this trace was prohibitively time-consuming. We estimate that the time required to check this property, at each node and for each destination (2500 relations), after each of the 6,446,316 input events, is more than 100 days based on extrapolating a 4-day run of the MEDL analysis. After using a number of optimizations, such as using `grep` to specialize the trace to only 5 nodes (25 relations), we could bring the analysis time for Monotone Sequence Numbers down to 51 seconds. We failed to carry out the analysis for all 50 nodes together, but we estimate that we could have carried out several 5-node analyses to check the complete trace in around 50,000 seconds. This kind of piecemeal analysis would not, however, work for more complex properties where more than 5 nodes may interact to cause an output event.

A central reason for our inability to carry out the MEDL analysis for all 50 nodes was the lack of a natural representation for packet events in MEDL. For instance, the Routeinfo event which signifies that a packet has some AODV routing information is expressed in MEDL as

```
event Routeinfo[at][dst] = Pkt when
        ((value(atnode,0)==at) &&
         (value(src,0)==dst) &&
         (value(pktty,0) > 0))
```

This event definition is expanded for every value of `at` and `dst` (50 each) to 2500 event definitions. This is because complex event attributes were not easy to assign. In NERL, the same event is expressed as the following single event definition.

```
event Routeinfo = Pkt OccurredWhen
            (Pkt.pktty > 0)
     WithAttributes {
       Routeinfo.at := Pkt.atnode;
       Routeinfo.dst := Pkt.src;
     };
```

A second reason for the performance limitation of the MEDL analysis is that MEDL is written in Java and is meant to interact with a running Java program through a socket interface. Instead, we were using it to interact with a network simulation written in C++, through a large trace file (around 1GB). The support libraries for reading and parsing the simulation trace are best written in C. The NERL recognizer for AODV is compiled to a C function which interfaces with this C support library. This library compatibility, along with the C vs. Java speedup, increases the monitor performance significantly.

We re-ran NERL on the trace, and we could process the complete trace for all 50 nodes, for all the AODV properties in 675 seconds. We found 708,727 errors. These errors were due to 3 significant (known) bugs in the simulator code, which were causing nodes to update their routing tables incorrectly.

7 Privacy Issues in Packet-mode Surveillance

When monitoring is used to reconstruct high-level network events, such as emails and web sessions, a number of privacy issues crop up. This is even more so in the case of surveillance systems, such as the FBI's Carnivore [SCHP+00], whose primary purpose is to capture data, such as email and web sessions, associated with a suspect for whom a warrant has been issued. The expectation of privacy for users of the network is typically expressed in terms of a monitoring policy or warrant that defines what kinds of data are allowed to be captured.

However, it is debatable (and often controversial) whether a closed source monitoring system actually respects the monitoring policy. Moreover, even if it does not intentionally violate user privacy, an incorrectly encoded protocol monitor may inadvertently collect private data. To address this problem, we have proposed OpenWarrants [BG02]: an open source surveillance system based on NERL as an event recognition engine. 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. In this section, we describe the OpenWarrants system and show how the layering, modularity, and formal analysis features in NERL are brought into play.

To execute OpenWarrants, the user must specify an email warrant that consists of at least the following 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.

To aggregate SMTP data, or Internet Message Headers [Cro82], 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 3. Large IP packets in the Internet are sometimes broken down into

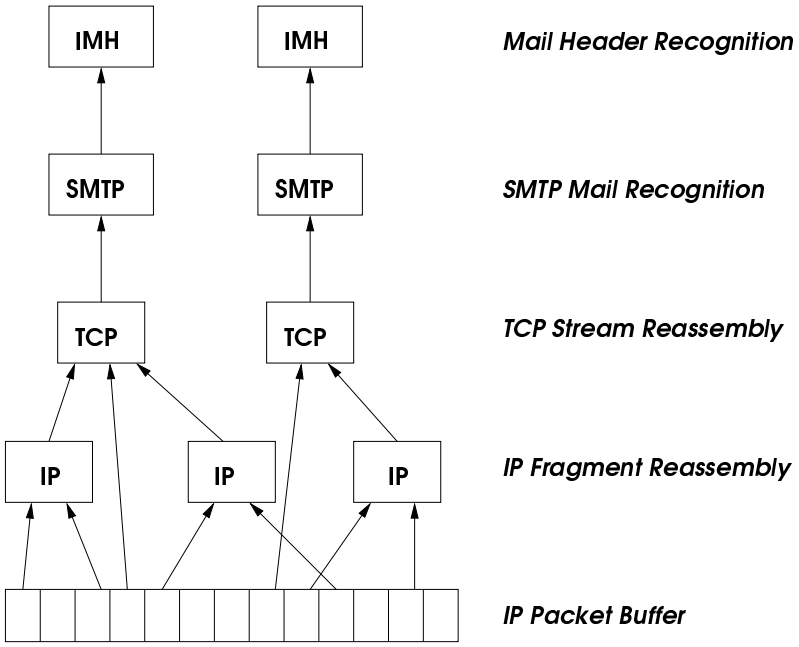

Fig. 3. Aggregation: Recognizing Emails

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.

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 recog-

nizes successful email transmissions. It then produces high-level events such as `MailAccepted`, indicating that an email was successfully delivered. The IMH modules parse Internet message headers, producing `IMHMessage` events that contain header fields as attributes.

Then, 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 `MailIdentified` event when the corresponding email is covered by the warrant. For instance, the `MailIdentified` 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 MailIdentified = (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")
```

Note how such a module definition is an easy-to-read, precise and executable representation of the monitoring policy. Compare it with an equivalent C program that would express this warrant. Moreover, one can see how aggregation and identification really validate the layering and modularity requirements that we set for NERL. They make the architecture of the OpenWarrants system clean and simple to implement.

Designing aggregation 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 Network Intrusion Detection Systems (NIDSs). We were unable to find rules for reconstructing SMTP messages in Snort (snort.org), the most popular open source NIDS. Another system, 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 [Gro01] and online documentation. We coded these in NERL as Recognizers `A` and `N`, and our own OpenWarrants version as Recognizer `O`. 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` captures some emails even if they are from U to U. To find this error, SPIN analyzed 871 states and 3135 transitions to produce a counter-example with 12 message exchanges. We then attempted the same proof for recognizer `N`. Again SPIN provides a counter-example. The SPIN counter-example has 16 messages and was found after analyzing 1610 states and 9897 transitions. 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.

This analysis shows that there are subtleties in writing protocol monitors, and not understanding what a module does can result in undesirable interactions, such as a loss of privacy in a surveillance system. We show how model-checking is an effective technique to find such interactions. As the protocol monitoring task gets more complex, having automated analysis tools becomes an essential requirement.

8 Conclusion

We have described the concept of Network Event Recognition and four requirements for convenient and efficient runtime analysis using NER. We have sketched a prototype that addresses the requirements by providing efficient packet format representations, support for layering and modularity, compatibility with existing network simulation and monitoring libraries, and support for formal analysis and automated transformations. The need for these features and the ways in which they have been addressed are illustrated by two case studies, one involving simulation analysis of routing protocols using the Verisim system, and one involving privacy in packet-mode surveillance using the OpenWarrants system. These studies show the importance of the requirements and the existence of feasible approaches for addressing them in significant and non-trivial applications.

References

[BCMG01] 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.

[BG02] Karthikeyan Bhargavan and Carl A. Gunter. Network event recognition for packet-mode surveillance. Submitted for publication, 2002.

[BGK+02] K. Bhargavan, C.A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Transactions on Software Engineering*, 28(2):129–145, February 2002.

[Cro82] D. Crocker. Standard for the Format of ARPA Internet Text Messages. Technical Report RFC 822, IETF, 1982.

[GGJZ00] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, May 2000.

[Gro01] NSS Group. Intrusion detection systems - group test, December 2001.

[Hol91] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, 1991. http://cm.bell-labs.com/cm/cs/what/spin/Doc/Book91.html.

[Hol00] G.J. Holzmann. Logic verification of ANSI-C code with SPIN. pages 131–147. Springer Verlag / LNCS 1885, Sep. 2000.

[KKM99] Eddie Kohler, M. Frans Kaashoek, and David R. Montgomery. A readable TCP in the Prolac protocol language. In *Proceedings of the ACM SIGCOMM '99 Conference: Applications, Technologies, Architectures, and Protocols for Computer Communication*, pages 3–13, Cambridge, Massachusetts, August 1999.

[LF98] David C. Luckham and Brian Frasca. Complex event processing in distributed systems. Technical Report CSL-TR-98-754, Stanford University, 1998.

[LKK+99] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M.Viswanathan. Runtime assurance based on formal specifications. In *Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[MC00] 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.

[ORD96] T.O. O'Malley, D.J. Richardson, and L.K. Dillon. Efficient specification-based test oracles. In *Second California Software Symposium (CSS'96)*, April 1996.

[Pax99] V. Paxson. Bro: A system for detecting network intruders in real-time. *Computer Networks*, 31:2435–2463, 14 December 1999. This paper is a revision of paper that previously appeared in Proc. 7th USENIX Security Symposium , January 1998.

[PF76] J. R. Pickens and D. J. Farber. The Overseer: A powerful communications attribute for debugging and security in thin-wire connected control structures. In *Proceedings of International Computer Communications Conference*, August 1976.

[Pos82] Jonathan B. Postel. Simple Mail Transfer Protocol. Technical Report RFC 821, IETF, 1982.

[SCHP+00] 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.

[SFG+] Jonathan M. Smith, David J. Farber, Carl A. Gunter, Sampath Kannan, Insup Lee, and Scott M. Nettles. Self-checking active networks (SCAN). Online at http://www.cis.upenn.edu/ jms/SCAN.html.