A High-Performance Network Intrusion Detection System*

R. Sekar
SUNY at Stony Brook, NY

Y. Guang S. Verma T. Shanbhag
Iowa State University, Ames, IA

Abstract

In this paper we present a new approach for network intrusion detection based on concise specifications that characterize normal and abnormal network packet sequences. Our specification language is geared for a robust network intrusion detection by enforcing a strict type discipline via a combination of static and dynamic type checking. Unlike most previous approaches in network intrusion detection, our approach can easily support new network protocols as information relating to the protocols are not hard-coded into the system. Instead, we simply add suitable type definitions in the specifications and define intrusion patterns on these types. We compile these specifications into a high-performance network intrusion detection system. Important components of our approach include efficient algorithms for pattern-matching and information aggregation on sequences of network packets. In particular, our techniques ensure that the matching time is insensitive to the number of patterns characterizing different network intrusions, and that the aggregation operations typically take constant time per packet. Our system participated in an intrusion detection evaluation organized by MIT Lincoln Labs, where our system demonstrated its effectiveness (96% detection rate on low-level network attacks) and performance (real-time detection at 500Mbps), while producing very few false positives (0.05 to 0.1 per attack).

1 Introduction

Network-based attacks have been increasing in frequency and severity over the past several years. Consequently, many research efforts have focussed on *network intrusion detection techniques* aimed at identifying such attacks. This paper describes a new approach to detect such attacks. The centerpiece of our approach is a domain-specific language that enables concise specification of network packet contents under normal as well as attack conditions. These specifications are compiled to produce a high-performance network intrusion detection system. The main benefits of our approach are:

- *concise, easy-to-develop intrusion specifications.* Using our domain-specific language, we can specify network-based attacks or other anomalous behavior easily and concisely. We have encoded the signatures for most low-level network probes and attacks using a specification that is about five lines each. Such conciseness contributes to increased confidence in the correctness of specifications, and leads to reduced development and debugging efforts.
- *high-speed, large-volume monitoring.* A central component of our approach is a fast pattern matching algorithm whose runtime is insensitive to the number of attack signatures. This algorithm ensures that the same packet field is never examined more than once, regardless of the number of patterns that refer to the field. This factor, combined with efficient data aggregation mechanisms, enable our system to support real-time performance at up to 500Mbps even when run on a standard PC.
- *robust and extensible.* Since an attacker is likely to attempt to disable the intrusion detection system by any means possible, it is particularly important for the system to be robust under all traffic conditions, e.g., malformed network packets should not crash the system. We have developed a novel type system that enables compact declarations of network packet structure and the constraints on their contents, so that these conditions can be automatically checked at compile-time and/or runtime without programmer involvement. Unlike previous approaches such as [MJ92] that hardcode network protocol specifics into the compiler for packet-filtering rules, our approach achieves robustness without compromising extensibility, as it is very easy to specify new packet structures (and thus be able to deal with new protocols and network services) without any modifications to the compiler.
- *comprehensive evaluation of performance.* This paper presents a comprehensive evaluation of our IDS based on a large set of intrusion training and test data provided by MIT Lincoln Labs [GLCFKWZ98]. The data covers a period of seven weeks, with each day's data in the range of 0.4 to 1.2GB. The evaluation results indicate that our approach is very effective (e.g., detects 96% of all network protocol related attacks in the test data), fast (approximate runtime of 15 seconds per GB of network traffic), and uses very little memory (less than 1MB).

1.1 Organization of the Paper

The rest of this paper is organized as follows. In Section 2 we describe our specification language. We illustrate this language with several examples in Section 3. An overview of our implementation is given in Section 4. Detailed study of the effectiveness and performance of our system are presented in Sections 5 and 6. Comparison with related work is presented in Section 7. We then conclude the paper with Section 8.

2 Specification Language

Intrusion specifications consist of variable and type declarations, followed by a list of rules. The rules are of the form $pat \rightarrow action$, where pat captures a pattern on sequences of network packets, and $action$ denotes the actions to be taken when we have a match for pat. Each of these components of the language are described in more detail below. We confine these descriptions to features that are unique to our language.

*This research is supported in part by Defense Advanced Research Agency's Information Technology Office (DARPA-ITO) under the Information System Survivability program, under contract number F30602-97-C-0244.

2.1 Declarations

The declarations consist of type and variable declarations. Types may be primitive or user-defined. Primitive data types in the language include `bool`, `bit`, `byte`, `short`, `int`, `long`, `float`, and `string`. For integral types, both signed and unsigned versions are supported. The compound types supported include event types (used to capture transmission or reception of packets or other events), packet types (used to capture the structure and content of network packets), lists, arrays and tuples. Below, we describe the event and packet types that are unique to our language.

2.1.1 Events

Events may be *primitive* or *user-defined.* Primitive events are generated by external systems and constitute the input to our detection system. Primitive event declarations are of the form

$$\texttt{event}\ \ eventName(parameterDecls)$$

where $parameterDecls$ is a list of declarations specifying the types of the parameters to the event $eventName$. In a system with a single network interface, we may have just two events (say, `tx` and `rx`), corresponding to the transmission and reception of packets on this interface. On systems with multiple interfaces, we may still have the two event types, but have then take an additional argument that specifies the interface, e.g.,

```
event  rx(int deviceId, ether_hdr p)
```

In addition to capturing reception or transmission of raw packets, events may provide higher level information as well. For instance, the declaration

$$\texttt{event telnetConn(}client, server, username\texttt{)}$$

may denote an event that is generated by a telnet server on completion of a telnet connection. Similarly, additional events may be used to describe information available from intermediate protocol layers, such as packet contents after IP fragment assembly.

User-defined events are *abstract* events that correspond to the occurrence of (potentially complex) sequences of primitive events. They take the form

$$\texttt{event}\ \ eventName(params) = pat$$

where pat is an event pattern described in Section 2.2. All of the variables in $params$ must appear in pat.

Similar to naming event patterns, our language also permits naming of arbitrary expressions. Named expressions are defined using the syntax

$$exprName(params) = expr$$

where $expr$ is an expression over the variables in $params$.

2.1.2 Packets

An obvious way to access the contents of a network packet is to treat it as a sequence of bytes. Then, a reference to the protocol field of an Ethernet header in an Ethernet packet in buffer `p` may be expressed using C-like syntax as `(short)p[12]`. Drawbacks of the byte sequence approach are that the type information for each field is lost and type casting is needed for most data references. Type-unsafety leads to several problems. For instance, a simple programming bug may cause access using an offset that is outside the packet boundaries which may cause a memory protection fault. Or, another simple programming bug using explicit type casting, such as `(int)p[15]`, may lead to a memory-related error on architectures that require integers to be aligned on a four or eight byte boundary. Semantic errors may arise even more frequently than access errors, since we may access an offset believing that it contains certain information, but in fact, the packet may be of a totally different type and contain completely different information. Language features that minimize the likelihood of these common errors are needed, since errors such as those mentioned above can crash the intrusion detection system, which may in turn bring down the entire system or leave it open to attacks.

Type systems used in most imperative, object-oriented or declarative programming languages are not sufficiently expressive to model network packets. In particular, a type system for network packets needs to deal with the following problems:

- the compiler or runtime system for the language does not have the freedom to choose a runtime representation; rather, the representations are prespecified as part of protocol standards
- the complete type of a network packet can be determined only at runtime, so type checking cannot be completed at compile-time

One way to deal with these problems is to hand-craft a type checker that is developed explicitly for a prespecified set of network protocols. This approach, used in BPF [MJ92] hard-codes the structure of packets for the prespecified protocols into the compiler, thereby requiring a redesign of the compiler to accommodate protocols that are not already built into the compiler, e.g., ATM, SNMP, and IPv6. We have developed an alternative approach that is more extensible. It is based on a flexible and expressive type system that can capture complex packet structures, while providing the capabilities to dynamically identify packet types at runtime and perform all relevant type checks before the packet fields are accessed.

We begin the description of packet types with a simple example of the type declaration for an Ethernet header. We use syntax that is similar to that of the C-language.

```
ETH_LEN =  6
ether_hdr {
  byte  e_dst[ETH_LEN]; /*Ethernet destination*/
  byte  e_src[ETH_LEN]; /*and source addresses*/
  short e_type;    /*protocol of carried packet*/
}
```

To capture the nested structure of protocol headers, we employ a notion of inheritance. For instance, an IP header can be considered as a subtype of `ether_hdr` with extra fields to store IP protocol information.

```
ip_hdr: ether_hdr {
  bit   version[4]; /* ip version */
  bit   ihl[4];     /* header length */
  byte  tos;        /* type of service */
  unsigned short tot_len;  /* total length */
  unsigned short id; /* Id for IP packet   */
  bit   flag[3];  /*    Various flags      */
  bit   frag_offset[13];
  byte  time_to_live;
  byte  protocol;   /* high-level protocol */
  unsigned short checksum;
  unsigned int saddr, daddr;
   /* Source and destintion IP addresses   */
}
```

Similarly, a TCP header inherits all of the data members from IP header and Ethernet header. However, simple inheritance by itself is not powerful or flexible enough to satisfy our needs. In particular, the structure describing a lower layer protocol data unit (PDU) typically has a field identifying the higher layer data that is carried over the lower layer protocol. For instance, the field `e_type` specifies whether the upper layer protocol is IP, ARP, or some other protocol. To capture such conditions, we augment inheritance with constraints. The structure for IP and TCP headers with the constraint information is as follows.

```
ETHER_IP = 0x0800
ip_hdr: ether_hdr with e_type=ETHER_IP {
  ... /* all fields same as beore */
}
IP_TCP = 0x0006
tcp_hdr: ip_hdr with protocol=IP_TCP {
  short   tcp_sport; /*source port       */
  short   tcp_dport; /*destination port  */
  int     tcp_seq;   /*sequence number   */
  int     tcp_ackseq /*acknowledge number*/
  ....       /*other fields, omitted here */
  byte tcp_data[tot_len-ihl];
}
```

Finally, we need to deal with the fact that the same higher layer data may be carried in different lower layer protocols. For this purpose, we develop a notion of disjunctive inheritance as follows. To capture the fact that IP may be carried within either an Ethernet or a token ring packet, we modify the constraint associated with `ip_hdr` into:

```
    (ether_hdr with e_type=ETHER_IP) or
         (tr_hdr with tr_type=TOKRING_IP)
```

It is instructive to compare disjunctive inheritance with traditional notions of single and multiple inheritance. In single inheritance, a derived class inherits properties from exactly one base class. In multiple inheritance, a derived class inherits the properties of every one of the (several) base classes. In contrast, disjunctive inheritance asserts that the derived class inherits properties from exactly one of many base classes. Viewed alternatively, multiple inheritance would correspond to a conjunction of constraints, whereas disjunctive inheritance corresponds to an exclusive-or operation.[1]

The semantics of the constraints is that they must hold before fields corresponding to a derived type are accessed. In particular, note that at compile time, we will not know the actual type of a packet received on a network interface, except for the lowest layer protocol. For instance, all packets received on an Ethernet interface must have the header given by `ether_hdr`, but we do not know whether they carry an ARP or IP packet. To ensure type safety, the constraint associated with the `ip_hdr` must be checked (at runtime) before accessing the IP-relevant fields. More generally, before a field in a structure of a particular type T is accessed, all constraints associated with all of the base types of T need to be checked.

2.2 Patterns

Patterns on packet sequences are used to specify normal network traffic as well as intrusions. The simplest pattern captures the occurrence of a single event, and is of the form $e(x_1, ..., x_n)|cond$, where e denotes an event (typically the reception or transmission of a packet on a network interface) and $x_1, ..., x_n$ denote the event arguments (typically the packet content). $cond$ denotes a boolean-valued expression involving $x_1, ..., x_n$ and possibly other variables. It may contain standard arithmetic, comparison and logical operations, and make use of external functions provided by the runtime environment. Patterns denoting the occurrence of several events can be combined using the || operator to denote the occurrence of one of these events. We can use the negation operator ! to denote nonoccurrence of events. The term *primitive pattern* denotes a pattern obtained using the operator || and possibly containing a single negation operator at the outermost level. Two examples of primitive patterns are:

```
rx(p)|(p.daddr=129.186.44.33)&&(p.tcp_dport=80)
!(rx(p)|p.protocol in KNOWN_PROTOCOLS ||
  tx(p)|p.protocol in KNOWN_PROTOCOLS)
```

The first pattern captures a TCP packet addressed to port 80 on the host with IP address 129.186.44.33. The second pattern captures any transmitted or received packet that is a non-IP packet, or has the protocol field in the IP-header set to a value different from those contained in the list `KNOWN_PROTOCOLS`.

To capture sequencing or timing relationships among events, we use several operators to compose primitive patterns into complex patterns. The basic composition operators are:

- *Sequential composition:* $p_1; p_2$ denotes pattern p_1 immediately followed by pattern p_2.
- *Alternation:* $p_1||p_2$ denotes the occurrence of either p_1 or p_2.
- *Repetition:* $p*$ denotes zero or more repetitions of the pattern p.
- *Real-time constraints:* p within $[t_1, t_2]$ denotes the occurrence of events corresponding to pattern p occurring over a time period $t_1 \leq t \leq t_2$. p over t is a shorthand for p within $[t, \infty]$, while p within t is a shorthand for p within $[0, t]$.

Note that most of these operators are similar to those used in regular expressions — the only difference is that we are trying to capture patterns on sequences of events with arguments, whereas regular expressions capture patterns on sequences of symbols. For this reason, we call our pattern language as regular expressions over events (REE).

To avoid excessive use of parenthesis, we define the following associativity and precedence for the sequencing operators. The operators || and ";" associate to the left. The operator ! has the highest precedence, "*" has the next lower precedence, ";" has the next lower precedence and || has the lowest precedence. As a convenient shorthand, we use the notation $p_1..p_2$ to stand for $p_1; (!(p_1||p_2))*; p_2$, i.e., occurrence of p_1 followed by p_2 without intervening occurrences of either pattern. Since we only permit negation of primitive patterns, the ".." operator is also applicable only for primitive patterns. To illustrate general patterns, consider:

```
rx(p1);tx(p2)*;rx(p3)|(p3.daddr=p1.daddr)
```

which denotes a sequence of two inbound IP-packets addressed to the same host with zero or more outbound packets in between. We will look at additional examples in the subsequent sections.

2.3 Data Aggregation Operations

In order to identify network attacks, it is often necessary to collect and aggregate information across many network packets, and act on the basis of this information. This operation needs to be more sensitive to recently received packets. Our language supports two principal abstractions for such aggregation, namely, counters and tables. We describe these abstractions below.

2.3.1 Counters

Counters keep track of the number of times a particular event pattern occurs. They are characterized by

- an aging function that assigns lower weights to events based on how far in the past they occurred
- higher and lower thresholds for the counter value
- high and low limit functions that are invoked when the counter value exceeds or falls below the high and low thresholds respectively

[1] From the point of view of describing packet structures, there seems to be little need for supporting multiple inheritance, as protocol layering typically ensures that a single PDU of a lower layer protocol carries a packet corresponding to exactly one higher layer protocol.

Desirable properties of a counter abstraction are that (a) it use constant space, and (b) the increment and decrement operations be performed in constant time. With arbitrary choice of aging functions, we cannot hope to satisfy these properties. For instance, suppose that we want the counter to weight event occurrences in a manner inversely proportional to how far in the past they occurred. Let $t_0, ..., t_n$ denote all of the times when the counter was incremented, with $t_0 \leq \cdots \leq t_n$. Then we end up with an equation such as the following one for the counter value at time $t \geq t_n$:

$$C(t) = \sum_{i=0}^{n} 1/(1 + k * (t - t_i))$$

Given the value of $C(t)$ and that another increment operation is invoked at time t', there is no apparent way to compute $C(t')$ short of reevaluating the above equation. Clearly, a reevaluation requires us to use $O(N)$ storage and $O(N)$ time where N is the number of occurrences of the increment operation in the past. Consequently, we focus on aging functions that can be implemented efficiently.

Our choice for aging function uses exponential weighting. In addition, since we may not be interested in occurrences of an event more than T units of time in the past, we may want to ignore such events. This leads us to the following equation for the value of the counter at time t, where m is smallest number such that $(t - t_m) \leq T$.

$$C(t) = \sum_{i=m}^{n} a^{(t_i - t)}$$

While there is no apparent way to compute $C(t)$ incrementally in this case either, we can use approximations. In particular, we divide the interval T into k windows of size T/k each. For each window, we maintain two quantities C_j and T_j that correspond respectively to the contribution of the window to the total count and the ending period of the window, i.e.,

$$C_j = \sum_{i=n_1}^{n_2} a^{(t_i - T_j)}$$

where $n_1, ..., n_2$ correspond to all occurrences of the increment operation within the time window j. Now, $C(t)$ can be computed using

$$C(t) = \sum_{j=1}^{k} a^{(T_j - t)} * C_j$$

which takes only $O(k)$ time. Choosing a small k makes this operation fast, at the expense of losing some accuracy. In practice, a choice of $k = 4$ seems to work well enough.

The syntax for specifying counters is as follows:

$$counter \;\; counterName(a, T, k, h, l, f_h, f_l)$$

where a, T and k have the meanings described above. The high and low thresholds are given by h and l, and the corresponding functions to be invoked are given by f_h and f_l. These counters support operations to increment, decrement or reset the counter, invoked using the syntax $counterName.inc()$, $counterName.dec()$ and $counterName.clear()$. The first two functions take an optional parameter that allows incrementing or decrementing by a number other than 1.

Typically, the functions f_h and f_l print something to a log file or generate an event that is to be processed by a higher level system. To avoid repetitive generation of the same message, we incorporate "hysteresis" into the process of invoking the threshold functions. In particular, f_h is invoked the first time the counter value crosses h after being below l. Subsequent crossings of h do not result in invocation of f_h unless the counter value goes below l. A similar condition applies to the invocation of f_l.

2.3.2 Tables

Tables are used to keep track of counts of many events simultaneously. They are similar to the histograms proposed in [Ranum97], but generalize it in the following ways: information stored in the table can be indexed using arbitrary types, "stale" entries in the table are automatically purged, and prespecified functions can be automatically invoked when the number of entries in the table go above or fall below specified thresholds. Purging stale entries is particularly important, as it enables us to remember information relevant for detecting attacks while using only a small amount of memory.

Each entry in a table is characterized by:

- $keyType$: type of the key using which the entry will be accessed from the table
- $dataType$: type of additional data stored in the entry
- a counter associated with the entry, characterized by a, T, k, h, l, f_h and f_l as before

In addition, the following parameters are shared by all entries in a table:

- N: the maximum number of entries that can be stored in the table
- f_d, f_f: functions to be invoked when an entry is deleted from the table (f_d) and when the table gets full (f_f)

When the number of entries in the table reaches the maximum number permitted, the entries associated with lowest counts are deleted from the table. This operation has the effect of retaining entries that have been accessed more often. The aging aspect of the counters ensures that among the entries that have been accessed the same number of times, those that have been accessed more recently have higher counts than those accessed earlier.

Rather than identifying and deleting only the entries with lowest counts, which may result in repeated invocation of this deletion operation, we prefer to select a fraction f and delete $k = \lceil f * N \rceil$ entries with the lowest counts. This approach ensures that the deletion operation is invoked at most once every $k = O(N)$ increment operations. Moreover, note that we can identify the lowest k entries in $O(N)$ expected time[2]. This means that the (expected) amortized cost of the deletion operation is just $O(1)$.

The syntax for declaration of tables is as follows. Suitable defaults are used for unspecified arguments. The default for a is 1, while the default for k is 4.

$$Table \;\; tn(kt, dt, N, a, T, k, h, l, f_h, f_l, f_d, f_f)$$

where tn, kt and dt denote the table name, key type and data type for the entries in the table.

Tables may optionally be initialized with certain entries. We refer to these entries as *static* entries to distinguish them from entries inserted dynamically in the table in response to receiving certain packets. Counts associated with static entries are maintained as with dynamically inserted entries, but static entries are never deleted from the table.

3 Examples

3.1 Very Small IP Fragments

We begin with a simple example to identify unusual network packets that can often be used to launch attacks. For instance, very short IP fragments that are smaller than TCP headers can be used to bypass packet-filtering firewalls. We can detect such packets using:

[2] A standard algorithm for accomplishing this is based on the quicksort algorithm — the difference being that instead of operating recursively on both halves obtained after partitioning, the modified algorithm confines itself to the half that holds the kth smallest element.

```
MY_NET = 129.186.44.0
MY_NET_MASK = 255.255.255.0
my_net_addr(a) = ((a&MY_NET_MASK)=MY_NET)
is_frag(p) = (p.more_frags)||(p.frag_offset!=0)

Table tcpFrag(
  unsigned int, /*key is IP address, no data*/
  100, 30, /*size 100, time window 30 sec*/
  1, 0,          /* hi, lo thresholds */
  tcpFragBegin, tcpFragEnd) /*threshold fns */
rx(p)|my_net_addr(p.daddr) &&
      is_frag(p) && p.protocol=IP_TCP &&
      p.tot_len < 48 -> tcpFrag.inc(p.saddr)
```

The functions `tcpFragBegin` and `tcpFragEnd` write records to a log file. They both take an argument that is the value of the key field corresponding to the table entry for which the action is being executed.

The threshold values in the example make attack detection to be very deterministic: an attack is recognized even if a single packet matching the criteria is received. The reasons for using a table in such a case (as opposed to directly invoking a function that generates an attack report) are as follows. First, we are able to distinguish among packets received with different source addresses and treat them as separate attacks. Second, the attacking host may generate a large number of fragmented packets that match this criteria. Rather than generating many attack messages, we may generate just two messages that indicate the beginning and end of the attack.

3.2 TCP SYN-Flood Attack

SYN-flood attack, otherwise known as neptune attack, involves sending a TCP connection initiation packet to a victim host with a nonexistent source address. The victim host sends back a SYN-ACK packet, but since the source address of the first packet is non-existent, the victim does not receive the ACK packet to complete the connection. As a result, the connection remains in a half-established state until a timeout occurs after a period of more than a minute. Since implementations of TCP limit the number of such half-open connections to a small number, the ability of the victim host to accept further TCP connections on a socket can be effectively eliminated by an attacker that sends in such attack packets, even at a relatively slow speed. We detect this attack using the following set of rules:

```
same_session(p, q) =
 p.daddr=q.saddr && p.tcp_dport=q.tcp_sport &&
  p.saddr=q.daddr && p.tcp_sport=q.tcp_dport
event tcp_syn(p) =
 rx(p)| my_net_addr(p.daddr) &&
         p.tcp_syn && !p.tcp_ack
event tcp_synack(p, q) =
 tx(q)| same_session(p,q) && q.tcp_syn &&
  q.tcp_ack && p.tcp_seqnum+1 = q.tcp_acknum
event tcp_ack(q, r) =
 rx(r)| same_session(q,r) && !r.tcp_syn &&
  r.tcp_ack && q.tcp_seqnum+1 = r.tcp_acknum
Table neptune(
  (unsigned int, unsigned short)
  /*key: (IP address,port) pair, no data field*/
  1000, 120, /*size 1000, window 120 seconds  */
  4, 1, neptuneBegin, neptuneEnd
  /* thresholds and associated functions*/)

(tcp_syn(p1)..tcp_synack(p1,p2));
   ((!tcp_ack(p2,p3))* over 60) ->
 neptune.inc((p1.daddr, p1.tcp_dport))
```

Each time a TCP SYN packet is received by a victim host (which may be any host within the network being monitored for intrusion), and a subsequent SYN-ACK packet generated by the victim host, this pattern monitors for the receipt of an ACK packet signaling the completion of the 3-way handshake. If this does not happen within sixty seconds, then the `neptune` table is incremented. If the increment operation is invoked sufficiently many times over a short period (e.g., four times within 120 seconds) then the `neptuneBegin` function would be invoked, which may in turn generate an alarm or record a message in a log file.

To avoid false alarms due to connection attempts with a host that may be down or temporarily disabled for other reasons, the neptune pattern counts only those TCP connection attempts for which the victim responded with a SYN-ACK packet. If alternative means to verify the health of the victim host were available, then we may count all TCP connection attempts that do not progress within sixty seconds or so.

For simplicity, the above pattern does not account for the fact that TCP connection attempts may be aborted in the middle, e.g., on receiving the FIN or RST packets. They can be dealt with by incorporating such packets into the above pattern.

3.3 Teardrop Attack

The teardrop attack involves fragmented IP packets that overlap. The following pattern captures any such overlap, without flagging those cases where a fragment is simply duplicated.[3]

```
frag_begin(p) = p.frag_offset*8
frag_end(p) = frag_begin(p)+p.tot_len-20
same_pkt(p,q) =
 p.daddr=q.daddr && p.saddr=q.saddr && p.id=q.id
event overlapping_frag(p1,p2) =
 rx(p2)| same_pkt(p1,p2) &&
         frag_begin(p2) < frag_end(p1) &&
         frag_begin(p1) < frag_end(p2) &&
         !(frag_begin(p1)=frag_begin(p2) &&
           frag_end(p1)=frag_end(p2))

(rx(p1)|is_frag(p1);(rx|tx)*;
 overlapping_frag(p1,p2)) within 60 -> ...
```

The pattern matches any sequence of packets that spans a period less than sixty seconds (one may choose a larger or smaller time frame), begins and ends with fragments of the same IP packet, and these fragments overlap partially.

4 Implementation

Our implementation consists of a compiler and a runtime system. The compiler is responsible for translating the intrusion specifications into C++ code. The aspects of compilation unique to our system include type-checking for packet data types and the compilation of pattern-matching. The C++ code produced by our compiler is compiled by a C++ compiler and linked with the runtime system to produce our IDS.

4.1 Type-checking for Packet Types

Type checking tasks that are specific to packet types involve name resolution and constraint enforcement. Name resolution refers to the problem of identifying the entity referred by an expression such as "`a.b`". Name resolution is complicated by the fact that in an expression of the form `a.b`, we may not know the exact type of `a`, but only the base class to which `a` belongs. To illustrate the problem, suppose that we have declarations of the form:

```
event ethRx(ether_hdr p)
event tokRx(tr_hdr p)
```

[3] Not all overlaps correspond to teardrop attacks, but we used this pattern since it is simpler than the one that would permit legitimate fragment overlaps, and since overlapping IP fragments never appeared in the environments where our IDS was tested.

The intuition here is that we associate an event type to denote packet reception on each type of network interface. Reception of a packet will result in the generation of this event, with the packet contents passed as a parameter to the event. Now consider the rule:

```
ethRx(p)|p.tot_len<20 && p.tcp_sport=80 -> ...
```

At compile time, based on the declaration of the event `ethRx`, we only know that the type of `p` is `ether_hdr`. With this information, if we attempt to resolve `p.tot_len`, there will be an error, since the `ether_hdr` contains no such field. Reporting such an error is clearly not the desired result, so we extend the name resolution process so that it uses the field name information to infer the runtime type of `p`. Specifically, when we see an expression of the form `a.b`, we search for the field `b` in the (declared) type of `a` and all its subtypes. Using this process on the expression `p.tot_len`, we can infer that the type of `p` is `ip_hdr`. On encountering the expression `p.tcp_sport`, we will further refine the type of `p` to be a `tcp_hdr`. When the type of `p` cannot be determined uniquely using this process, the type checker will return an error. This would happen only when the declared type of `T` is such that two (or more) descendent classes of `T` use a field with the same name. To disambiguate such cases, the event arguments can be further qualified to indicate the runtime type of an argument:

```
ethRx(tcp_hdr p)|p.tot_len<20 ...
```

Finally, although a pattern may assume that packets being processed by that pattern are of a certain type, we need to verify this fact at runtime. More generally, before accessing a field `f` in a packet, we need to check all constraints associated with the type `T` containing `f` and all its ancestor types. For instance, in the above pattern, before we access the field `p.tot_len`, which is a field in the class `ip_hdr`, we need to verify the constraint `p.e_type = ETHER_IP` associated with this type. As part of type checking, we explicitly add these constraints to the event patterns. We also introduce checks to ensure that the packet length is large enough that all offset accesses fall within the packet.

To ensure that the preconditions are indeed checked at runtime before accessing a particular field, the ordering among the newly introduced conditions and the original conditions in the pattern have to be maintained. We do this by introducing a new ordered conjunction operation `&&&` as follows. The semantics of ordered conjunction `a &&& b` is that the condition `a` needs to be checked first, and only if it is true, `b` will be checked[4]. The result of applying these type-checking operations on the above pattern is:

```
ethRx(p)|length(p)>=offset(p.tcp_data) &&&
 (p.e_type=ETHER_IP &&& p.tot_len<20) &&
 (p.e_type=ETHER_IP &&& p.protocol=IP_TCP
   &&& p.tcp_sport = 80) -> ...
```

Note that the same constraint may appear multiple times in the pattern at this point, but later stages of the compiler will ensure that no constraint is checked more than once.

In the presence of disjunctions in the constraint, such as `tcp_hdr: ether_hdr with e_type=ETHER_IP or tr_hdr with tr_type=TOKRING_IP`, recall that the alternatives in the condition are mutually exclusive. For instance, in the above example of `ethRx`, we will be able to determine statically that the packet type is `ether_hdr` and not `tr_hdr`. Based on this, the constraint regarding `tr_hdr` is not applicable and can be discarded.

[4]This contrasts with the semantics of `&&`, which does not require us to order the tests. The reordering permitted by `&&` plays an important role in improving the performance of pattern-matching algorithms.

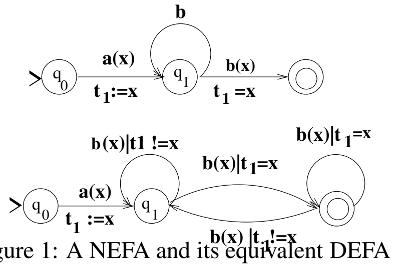

Figure 1: A NEFA and its equivalent DEFA

4.2 Compilation of Pattern-Matching

Efficient pattern-matching is key to the performance of our IDS. Our approach to pattern-matching is based on compiling the patterns into a kind of automaton in a manner analogous to compiling regular expressions into finite-state automata. We call these automata as *extended finite-state automata* (EFSA). EFSA are simply standard finite state automata (FSA) that are augmented with a fixed number of *state variables,* each capable of storing values of a bounded size. Every transition in the EFSA is associated with an event, an enabling condition involving the event arguments and state variables, and a set of assignments to state variables. The final states of the EFSA may be annotated with actions, which, in our system, will correspond to the response actions given in our rules. For a transition to be taken, the associated event must occur and the enabling condition must hold. When the transition is taken, the assignments associated with the transition are performed.

An EFSA is normally nondeterministic. The notion of acceptance by a nondeterministic EFSA (abbreviated as NEFA) is similar to that of an NFA. A deterministic EFSA (DEFA for short) is an EFSA in which at most one of the transitions is enabled in any state of the EFSA. A NEFA for the pattern $a(x); b*; b(x)$ is shown in Figure 1. The equivalent DEFA is also shown in the same figure.

We have shown that translating a NEFA to a DEFA can result in an unacceptable blowup in the size of the automaton. Therefore we have developed a new approach that is based on translating NEFA into what we call as *quasi-deterministic extended finite state automata* (QEFA). QEFA eliminate most of the sources of nondeterminism that are present in the NEFA, while still ensuring that their sizes are acceptable. A complete treatment of QEFA and the compilation algorithm can be found in [SU99].

4.3 Runtime System

The runtime system provides support for capturing network packets either from a network interface or from a file. The code for doing this is currently based on the Berkeley packet filter code. This code is used to read all network packets (either from a file or a network interface), leaving the actual filtering and other processing to the code generated by our compiler. The runtime system also provides the implementation of the data structures mentioned earlier for performing data aggregation.

5 Effectiveness

Our IDS participated in a comprehensive evaluation of intrusion detection systems conducted by MIT Lincoln labs [GLCFKWZ98]. To the best of our knowledge, this was the first comprehensive and comparative evaluation of intrusion detection systems to date. Participants in the evaluation included research groups from UC at Santa Barbara, Columbia University, RST Corporation, and two groups from SRI. A baseline system comparable to commercial intrusion detection systems was also included in the evaluation. It was determined that all of the systems participating in the evaluation provided significantly better detection rates over the baseline system, while reducing false pos-

```
09:51:21    Portsweep                 attack began   :   207.136.086.223  --> 172.016.114.050
09:51:21    Portsweep                 attack ended   :   207.136.086.223  --> 172.016.114.050
13:49:52    Ping of death             attack began   :   172.016.114.050
15:08:16    UDPIP Fragment Too Small  attack began   :   172.016.113.050
15:08:17    Teardrop                  attack began   :   172.016.113.050
15:08:18    Teardrop                  attack ended   :   172.016.113.050
13:49:52    Ping of death             attack ended   :   172.016.114.050
15:08:18    UDPIP Fragment Too Small  attack ended   :   172.016.113.050
07:32:04    Portsweep                 attack began   :   153.107.252.061  --> 172.016.114.050
07:40:15    Portsweep                 attack began   :   195.073.151.050  --> 172.016.114.050
07:34:15    Portsweep                 attack ended   :   153.107.252.061  --> 172.016.114.050
07:40:15    Portsweep                 attack ended   :   195.073.151.050  --> 172.016.114.050
09:11:37    Neptune                   attack began   :   172.016.114.050
09:14:52    Neptune                   attack ended   :   172.016.114.050
17:15:57    Neptune                   attack began   :   172.016.113.050
17:16:05    Portsweep                 attack began   :   166.102.114.043  --> 172.016.113.050
18:07:54    Neptune                   attack ended   :   172.016.113.050
18:05:55    Portsweep                 attack ended   :   166.102.114.043  --> 172.016.113.050
10:14:27    Portsweep                 attack began   :   207.253.084.013  --> 172.016.118.020
15:30:05    UDPIP Fragment Too Small  attack began   :   172.016.113.050
15:30:05    Teardrop                  attack began   :   172.016.113.050
15:30:07    Teardrop                  attack ended   :   172.016.113.050
15:30:07    UDPIP Fragment Too Small  attack ended   :   172.016.113.050
02:03:29    Portsweep                 attack ended   :   207.253.084.013  --> 172.016.118.020
```

Figure 2: Sample output produced by our IDS.

itive rates by an order of magnitude or more.

The evaluation organizers set up a dedicated network to conduct a variety of attacks. Care was taken to ensure the accuracy of normal traffic as well. All of the network traffic was recorded in tcpdump format and provided to the participants of the evaluation. The data provided consisted of seven weeks of training data, plus two weeks of test data. The tcpdump files were 0.4 to 1.2GB in length per day.

The participants were required to tag each TCP session and each non-TCP packet in the tcpdump file as representing an attack or not. Optionally, a probability could be assigned to indicate the likelihood of an attack. For TCP, entire sessions were tagged as opposed to individual packets. These "raw results" were then processed by MIT Lincoln labs to produce standardized scores for all the participants. Thus the results presented below have been independently verified [GLCFKWZ98]. Although the format of the raw results was verbose, the upside was that being so large (about 100K sessions per day), it was impossible to use manual approaches to cross-check the results produced by the IDS systems before they were scored by Lincoln Labs. In addition to the raw results, our system is capable of producing human-friendly attack reports, a sample of which is shown in Figure 2.

The attacks were classified into four categories. Of these, only two categories related to low-level network attacks for which our system is designed[5]. These two categories were probing and denial-of-service.

Figure 3 shows the list of attacks that are currently identified by our system. The attacks are identified using rules that are generally similar to the examples discussed earlier. However, in the process of training and debugging the system, we have found that the rules tend to get a bit more complicated than the examples. At times, we have also had to change the rules due to certain idiosyncrasies or artifacts in the test data.

Figure 4 shows the overall scores assigned to our system by Lincoln Labs [GLCFKWZ98]. The scoring scheme assigned fractional credit to each attack based on the percentage of the attack-containing packets (or sessions) identified by the IDS being evaluated. For instance, a port sweep may occur over hundreds or thousands of packets. Any IDS is able to identify only a subset of these packets as being part of a port sweep. This scoring procedure is not favorable for systems such as ours that emphasize low false positives. Such systems tend to err on the side of not identifying individual packets as attack-bearing, as long as a substantial number of packets within the attack can be tagged. Nevertheless, our system finished among the top two in both categories at low false-positive rates of 0.05 to 0.1 false alarms per attack. At much higher false-positive rates, e.g., 2 to 3 false alarms per attack, some of the other systems start performing better than us.

Figure 5 shows the scores obtained by our IDS for each kind of attack. Since it omits some of the higher-level probing and denial of service attacks that are not addressed by our system, the aggregate score shown in this table is an improvement over that given in Figure 4. This table also shows the result under a different scoring scheme that attempts to identify whether each an attack is completely missed by a system. If a substantial fraction of the attack-bearing packets (say, 50%) are detected by the system, then we treat the attack as having been detected. Otherwise, we treat the attack as having been missed. Our system demonstrated excellent detection capability (96%) when using this criteria. The only attacks missed were due to the fact that the tcpdump contained only packets arriving into the network from outside, while we had assumed that it contained all of the internal traffic as well. As such, the explosion in the number of packets expected by our system as part of a UDP loop attack was not present in the tcpdump data, and hence the attack was missed by our system.

6 Performance

Our emphasis on efficiency of implementation paid off in terms of performance, as shown by the CPU and memory usage of our IDS for the ten days of test data as shown in Figure 6. While running on a 450MHz Pentium II PC running RedHat Linux 5.2, our system can sustain intrusion detection at the rate of 15s/GB, or

[5] Attacks on higher-level software such as buffer overflows, race conditions and vulnerabilities in setuid programs are not detected by the system described in this paper. This is because it is much harder to piece together low-level information contained in network packets to identify these higher level attacks. Instead, we rely on a different subsystem that intercepts and monitors system calls made by processes to identify such attacks.

Attack name	Description
ipsweep	Surveillance sweep performing port sweep or ping on multiple hosts
land	Denial of service using TCP packet with the same source and destination address
neptune	Syn flood denial of service
pod	Denial of service using oversized ping packets
teardrop, nestea	Overlapping IP fragments
portsweep	Sweep through many ports to determine available services on a single host
smurf	ICMP echo reply flood, caused by an ICMP echo packet with spoofed address (of victim) sent to a network broadcast address
UDP loop	Denial of service, created by sending UDP packets with source address of a simple UDP service and destination address of another
pingflood	Flood of icmp packets (but no smurf present)
ipspoofing	Attempt to establish a TCP connection with a spoofed source address
smurf_int_site	Intermediate site for a smurf attack
fraggle	Like smurf, but uses UDP rather than ICMP
ipversion, protocol	Unknown protocol or version
tcpfragtoosmall	TCP packet that is very small, yet has been fragmented at the IP level
udpfragtoosmall	Similar to above, but for UDP
broadcast	Packets sent to broadcast addresses for simple UDP services
udpdataflood	unusually large volume of data for a UDP port
nmap, satan, saint, mscan	surveillance tools, produce many different variations of port and ip sweeps, as well as attacks on higher level services

Figure 3: Attack Repertoire

Attack Category	Number of Attacks	False positives	Our Score	Best score in evaluation
Probe	17	1	86%	86%
Denial of service	43	4	60%	65%

Figure 4: Overall scores by category of attacks.

equivalently, over 500Mb/second. (In measuring the CPU time, we considered only the time spent within the intrusion detection system, and ignored the time for reading packets from the tcpdump file.) Its memory consumption is also low, largely the result of our choice of data aggregation operations. The high performance is the result of our emphasis on the following aspects:

- *insensitivity of the pattern-matcher to the number of rules.* Our IDS currently contains about 75 rules, so any pattern-matching approach that involves checking each of this patterns individually will be slow. By compiling the patterns into an automaton, we are able to identify all pattern-matches, while spending essentially constant time per packet that is independent of the number of patterns. Thus the pattern-matching time remains independent of the number of rules.
- *fast implementation of data aggregation operations.* As described earlier, we have implemented the weighted counter and table data structures so that operations on them have an amortized $O(1)$ cost per operation. As a result, detection time increases only linearly (and slowly) with the number of attacks.

We note that the time for detection does not monotonically increase with the number of rules. This is because of the fact that the addition of a new rule can reduce the frequency with which

Attack	Number	Misses	Score
Smurf	8	0	100%
Teardrop	4	0	100%
Land	2	0	100%
Ping of Death	5	0	99%
IP Sweep	3	0	96%
satan	2	0	94%
Port Sweep	5	0	90%
saint	2	0	89%
nmap	4	0	78%
Neptune	7	0	70%
mscan	1	0	55%
UDP loop	2	2	0%
Total	45	2	85%

Figure 5: Scores on low-level network attacks.

Week	Day	Data file size (GB)	Time/GB of data (sec)	Memory (MB)
1	Mon	0.41	7.6	< 1
1	Tue	0.84	21.4	< 1
1	Wed	0.46	12.2	< 1
1	Thu	0.76	21.8	< 1
1	Fri	0.43	17.4	< 1
2	Mon	1.20	21.4	< 1
2	Tue	0.45	15.3	< 1
2	Wed	0.54	8.7	< 1
2	Thu	0.60	13.0	< 1
2	Fri	0.50	10.6	< 1

Figure 6: Runtime and memory usage for detection.

an earlier rule was matching. This factor can lead to the situation where the addition of rules *decreases* the execution time.

7 Related work

Historically, intrusion detection systems have been classified into two broad categories: host-based systems, which are aimed at protecting individual hosts and operate on the basis of information contained in audit logs or other similar sources of data, and network-based systems, which operate by monitoring network traffic. The system described in this paper falls in the second category.

Although network intrusion detection systems [Heberlein90, PN97, Hochberg93, LPS99, MHL94, Paxson98, VK98, Ranum97] operate by inspecting IP (or lower level)

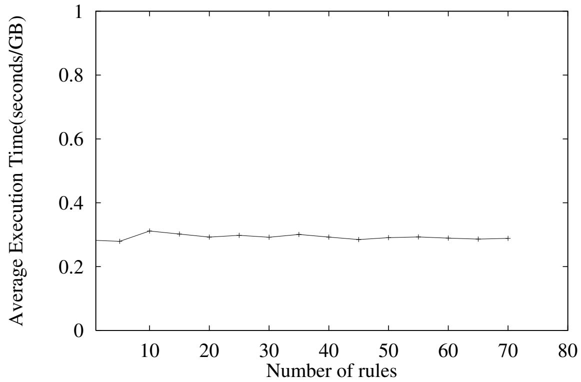

Figure 7: Pattern-matching time Vs number of rules.

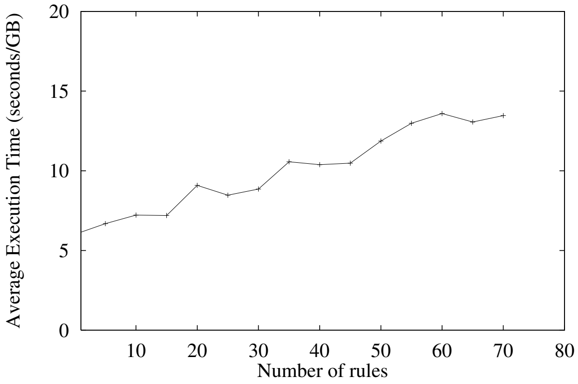

Figure 8: Intrusion detection time Vs number of rules.

packets, most of them attempt to reconstruct the higher level interactions between end hosts and remote users, and identify anomalous or attack behaviors. Based on this, they attempt to identify a broad class of attacks, focussing particularly on malicious attacks on network servers and other processes running on the target system. We place a different emphasis in our system — we are particularly interested in detecting low-level attacks that do not target specific processes, but exploit vulnerabilities in the design and implementation of host operating systems and network protocols. Most surveillance, probing and a large number of denial-of-service attacks in existence fall into this category of low-level network attacks. This approach complements host-based approaches that can identify higher-level attacks by examining audit logs or system calls.

A completely different approach is taken for intrusion detection in [LPS99], where techniques based on data mining are employed. Several previous works such as [Heberlein90] also employed statistical and expert-system based techniques for detecting anomalous behaviors that could be indicative of attacks. These techniques largely complement pattern-matching based schemes such as ours. In particular, the benefits of our approach are speed, specificity and reduction of false positives. The downside is that unknown attacks, hitherto not captured, may go undetected. The anomaly detection systems are typically better at detecting unknown attacks, but their downsides include high false-positive rates, nonspecific attack indicators, and need for extensive training. Combination approaches, such as those envisioned in EMERALD, can give us the benefits of both approaches while largely avoiding their drawbacks.

We earlier developed a specification language for capturing behaviors in terms of UNIX system calls [SBS99], and also an algorithm for fast matching of behavioral patterns [SP99]. The pattern component of the language presented in this paper is the same as that work, but the type system and data aggregation abstractions presented in this paper are new. Moreover, the implementation, experimentation and analysis results presented in this paper focus on network-based attacks, as opposed to attacks on processes.

7.1 Related Work in Languages for Network Intrusion Detection and Packet Filtering

The use of special-purpose languages for network intrusion detection has been studied earlier. The choices range from scripting languages that make it easier to write intrusion detection code [Ranum97], C-like-but-strongly-typed languages such as that used in Bro [Paxson98], to a pattern-matching language in NetSTAT [VK98]. A common feature of these languages is that they are based on an imperative programming paradigm, whereas our language is declarative. Moreover, our language permits us to more easily capture patterns on sequences of packets, as opposed to other languages where patterns can characterize only individual events. This capability, together with the data aggregation features provided by our language, contributes to the conciseness of intrusion specifications. Another important distinction of our approach is that our language is designed to support efficient implementations of the pattern-matching and data aggregation operations.

Most previous approaches for network intrusion detection (or packet filtering) hard-coded the details of TCP/IP packet formats in their implementations, whereas our language supports a type system to specify the structure and content of the packets. Our approach makes it easy to support new protocols — the effort involved is that of declaring the type for the packets formats corresponding to the new protocol. The rest of the work, including dynamic type-checking to identify packet types at runtime and offset calculations to access specific fields, are automated by the compiler for our language.

Our type system for network packets, originally described in [Guang98], is similar to *packet types* that have been developed independently in [CM99]. Their notion of type refinement is similar to our notion of inheritance for packets in that both approaches make use of constraints to augment the traditional notion of inheritance. This gives both approaches the ability to model layering of protocols. However, there are several significant differences as well. In particular, the approach of [CM99] affords increased expressive power in the following ways:

- layering is captured in our approach purely in terms of inheritance with constraints, with the contents of a higher layer PDU viewed as an extension of the header for the lower layer PDU. In contrast, [CM99] uses two distinct concepts, namely, refinement of a base type by which the values of certain fields are specified, and an overlay construct that overlays the data portion of the lower-layer PDU with that of the higher layer. The approach of [CM99] offers more power in that it can capture protocols that use a header as well as trailer for packet content, while our approach trades off this power for simplicity.
- processing of packets by protocol software can be captured using a *becomes* relation in [CM99], which maps the contents of packets before processing by a layer of protocol software to the contents after processing. For instance, a mapping could be provided between IP packets before and after reassembly of fragments. In our approach, this mapping relation is not captured by the type system.

Our approach provides the following features that are not supported in [CM99]:

- disjunctive inheritance that can capture the layering of a higher layer protocol on multiple lower layer protocols
- capabilities for type resolution even when complete type information is not provided (but only a base type) for a packet
- a general purpose algorithm that avoids repetition of constraint checking operations even if they are repeated along an inheritance chain or within rules

We remark that since both approaches are founded on the notion of inheritance and constraints, it should be easy to combine the features provided by the two approaches.

8 Conclusions

In this paper we presented a new approach for network intrusion detection. A key feature of our approach is a domain-specific language for capturing patterns on normal and/or abnormal network packet sequences. We illustrated our language with several examples. As shown by these examples, our language supports concise and easy-to-write attack patterns. This in turn increases our confidence in attack specifications and reduces the development and debugging times needed for defending against new attacks.

We have developed convenient and expressive abstractions for aggregating data across multiple network packets. We have also developed efficient implementation of these abstractions. In addition, we have developed efficient implementation of the pattern-matching operations needed in the language. A key feature of this implementation is that the pattern-matching time is insensitive to the number of rules, thus making the approach scalable to large number of rules, and consequently, to a large number of attacks. The high performance also enables us to perform network intrusion detection without packet drops on high speed networks – sustaining detection at gigabit rates appears quite feasible.

A contribution of this paper is a new type system for network packets that makes it convenient to operate on network packets, while also enhancing the robustness of the systems operating on those packets by protecting against a variety of memory access and other type-related errors. Moreover, the high-level type information makes it easier to achieve robustness without compromising on efficiency. Finally, this approach makes our system easily extensible to support new network protocols.

We presented the results of a comprehensive evaluation of our system by MIT Lincoln Labs [GLCFKWZ98] as part of a larger effort to compare current IDS. These results show that our approach is effective and efficient for detecting low-level network attacks, while producing a very small number of false positives. In the near future, we plan to integrate the IDS described in this system with a second system that operates by intercepting and examining system calls made by processes. Together, we expect the system to provide robust defenses against most attacks known today.

References

[ALJTV95] D. Anderson, T. Lunt, H. Javitz, A. Tamaru, and A. Valdes, Next-generation Intrusion Detection Expert System (NIDES): A Summary, SRI-CSL-95-07, SRI International, 1995.

[Bates95] P. Bates, Debugging Distributed Systems Using Event-Based Models of Behavior, ACM Transactions on Computer Systems, 1995.

[CERT98] CERT Coordination Center Advisories 1988–1998, `http://www.cert.org/advisories/index.html`.

[CM99] S. Chandra and P. McCann, Packet Types, Workshop on Compilers Support for Systems Software.

[Denning87] D. Denning, An Intrusion Detection Model, IEEE Trans. on Software Engineering, Feb 1987.

[FHS97] S. Forrest, S. Hofmeyr and A. Somayaji, Computer Immunology, Comm. of ACM 40(10), 1997.

[GLCFKWZ98] I. Graf, R. Lippmann, R. Cunningham, D. Fried, K. Kendall, S. Webster and M. Zissman, Results of DARPA 1998 Offline Intrusion Detection Evaluation, `http://ideval.ll.mit.edu/results-html-dir`, 1998.

[GM96] B. Guha and B. Mukherjee, Network Security via Reverse Engineering of TCP Code: Vulnerability Analysis and Proposed Solutions, Proc. of the IEEE Infocom, March 1996.

[GSS99] A.K. Ghosh, A. Schwartzbard and M. Schatz, Learning Program Behavior Profiles for Intrusion Detection, 1st USENIX Workshop on Intrusion Detection and Network Monitoring, 1999.

[Guang98] Y. Guang, Real-time packet filtering module for network intrusion detection system, Department of Computer Science, Iowa State University, July 1998.

[Heberlein90] L. Heberlein et al, A Network Security Monitor, Symposium on Research Security and Privacy, 1990.

[Hochberg93] J. Hochberg et al, NADIR: An Automated System for Detecting Network Intrusion and Misuse, Computers and Security 12(3), May 1993.

[Ilgun93] K. Ilgun, A real-time intrusion detection system for UNIX, IEEE Symp. on Security and Privacy, 1993.

[LBMC94] C. Landwehr, A. Bull, J. McDermott and W. Choi, A Taxonomy of Computer Program Security Flaws, ACM Computing Surveys 26(3), 1994.

[LPS99] W. Lee, C. Park and S. Stolfo, Automated Intrusion Detection using NFR: Methods and Experiences, USENIX Intrusion Detection Workshop, 1999.

[LV95] D. Luckham and J. Vera, An Event-Based Architecture Definition Language, IEEE Transactions on Software Engineering, 21(9), 1995.

[LHMBH87] D. Luckham, D. Helmbold, S. Meldal, D. Bryan, and M. Haberler, Task Sequencing Language for Specifying Distributed Ada Systems: TSL-1, PARLE: Conf. on Parallel Architectures and Languages, LNCS 259-2, 1987.

[Lunt92] T. Lunt et al, A Real-Time Intrusion Detection Expert System (IDES) - Final Report, SRI-CSL-92-05, SRI International, 1992.

[Lunt93] T. Lunt, A survey of Intrusion Detection Techniques, Computers and Security, 12(4), June 1993.

[MJ92] S. McCanne and V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture, Lawrence Berkeley Laboratory, Berkeley, CA, 1992.

[MHL94] B. Mukherjee, L. Heberlein and K. Levitt, Network Intrusion Detection, IEEE Network, May/June 1994.

[Paxson98] V. Paxson, Bro: A System for Detecting Network Intruders in Real-Time, USENIX Security Symposium, 1998.

[PN97] P. Porras and P. Neumann, EMERALD: Event Monitoring Enabled Responses to Anomalous Live Disturbances, National Information Systems Security Conference, 1997.

[Ranum97] M. Ranum et al, Implementing A Generalized Tool For Network Monitoring, LISA, 1997.

[SBS99] R. Sekar, T. Bowen and M. Segal, On Preventing Intrusions by Process Behavior Monitoring, USENIX Intrusion Detection Workshop, 1999.

[SP99] R. Sekar and P. Uppuluri, Synthesizing fast intrusion detection/prevention systems from high-level specifications, USENIX Security Symposium, 1999.

[SU99] R. Sekar and P. Uppuluri, Synthesizing Fast Intrusion Prevention/Detection Systems from High-Level Specifications, Technical Report 99-03, Department of Computer Science, Iowa State University, Ames, IA 50014.

[VK98] G. Vigna and R. Kemmerer, NetSTAT: A Network-based Intrusion Detection Approach, Computer Security Applications Conference, 1998.