USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

The following paper was originally published in the

Proceedings of the Workshop on Intrusion Detection and Network Monitoring

Santa Clara, California, USA, April 9–12, 1999

Automated Intrusion Detection Using NFR: Methods and Experiences

Wenke Lee, Christopher T. Park, and Salvatore J. Stolfo
Columbia University

For more information about the USENIX Association:
Phone: 1 510 528 8649 FAX: 1 510 548 5738
Email: office@usenix.org WWW: http://www.usenix.org

Automated Intrusion Detection Using NFR: Methods and Experiences

Wenke Lee Christopher T. Park Salvatore J. Stolfo

Computer Science Department, Columbia University
500 West 120th Street, New York, NY 10027
{wenke,cpark,sal}@cs.columbia.edu

Abstract

There is often the need to update an installed Intrusion Detection System (IDS) due to new attack methods or upgraded computing environments. Since many current IDSs are constructed by manual encoding of expert security knowledge, changes to IDSs are expensive and require a large amount of programming and debugging. We describe a data mining framework for adaptively building Intrusion Detection (ID) models specifically for use with Network Flight Recorder (NFR). The central idea is to utilize auditing programs to extract an extensive set of features that describe each network connection or host session, and apply data mining programs to learn rules that accurately capture the behavior of intrusions and normal activities. These rules can be used for misuse detection and anomaly detection. Detection models are then incorporated into NFR through a machine translator, which produces a working detection model in the form of N-Code, NFR's powerful filtering language.

1 Introduction

With the increase of Internet connectivity, there is the ever increasing risk of attackers illicitly gaining access to computers over the network. Intrusion detection is often used as another wall to protect computer systems, in addition to the standard methods of security measures such as user authentication (e.g. user passwords or biometrics), avoiding programming errors, and information protection (e.g., encryption). Intrusion detection techniques can be categorized into *anomaly detection* and *misuse detection*. While accuracy is the essential requirement, its extensibility and adaptability are also critical design criteria in today's network computing environment. There are multiple "penetration points" for intrusions to take place in a network system. For example, at the network level, carefully crafted "malicious" IP packets can crash a victim host; at the host level, vulnerabilities in system software can be exploited to yield an illegal root shell. Since activities at different penetration points are normally recorded in different audit data sources, an IDS often needs to be extended to incorporate (additional) modules that are specialized for certain components (e.g., hosts, subnets, etc.) of a network systems. The large traffic volume in security related mailing lists and Web sites suggests that new system security holes and intrusion methods are continuously being discovered. Therefore, it is imperative that IDSs be updated frequently and rapidly.

Currently building an effective IDS is an enormous knowledge engineering task. System builders largely rely on their intuition and experience to select the statistical measures for anomaly detection [6]. Many IDSs only handle one particular audit data source, and updating these systems is expensive and slow. Some of the recent research and commercial IDSs have begun to provide built-in mechanisms for customization and extension. The Network Flight Recorder (NFR) [8] is one such extensible system that combines data collection, analysis, and storage within a single platform. We discuss NFR in more depth in Section 2.2. Such systems would normally be located between a firewall and an Internet connection, an area aptly named the DMZ. Analysis in NFR is accomplished by scripts based on a language called N-code, NFR's flexible language for traffic analysis. Information is displayed in NFR to a web-based interface with Java support. NFR also has a real time alerting capability and a storage subsystem that allows data to be stored, rotated, and archived to other external devices [8]. However, this does not eliminate the need for experts to first analyze and categorize attack scenarios and system vulnerabilities, and hand-code the corresponding rules and patterns in N-code for misuse detection. Because of the manual and ad hoc nature of the development process, current IDSs including NFR have limited extensibility and adaptability. Our goal is

to substantially reduce this effort by automating:

1) the task of building intrusion detection through data-mining.

2) generating the N-code for NFR to detect intrusions via a machine translator.

While using such methods, system builders and administrators will still have to maintain and fine-tune the respective IDS. However a large amount of their work will be automated, thus effectively reducing time and manpower in fielding an effective IDS.

2 Techniques for Intrusion Detection

There are two major categories of techniques that are used by most IDS's:

- Anomaly detection attempts to determine whether deviation from an established normal behavior profile can be flagged as an intrusion [4]. A profile typically consists of a number of statistical measures on system activities, for example, the *CPU usage* and the *saturation of bandwidth at a given time period*. Deviation from a profile can be computed as the weighted sum of the deviations of the constituent statistical measures. Profiles can be updated periodically (aged) so that shifts of normal behavior are accounted. The key advantages of anomaly detection systems is that they can detect unknown intrusions since they require no *a priori* knowledge about specific intrusions. However, defining and maintaining "normal" profiles is a nontrivial and error-prone task.

- Misuse detection refers to techniques that use patterns of known intrusions (for example, *more than three consecutive failed logins within 2 minutes* can be classified as a penetration attempt). The sequence of attack actions, the conditions that compromise a system's security, as well as the evidence (e.g., damage) left behind by intrusions can be represented by a number of general pattern matching models. The key advantage of misuse detection systems is that once the patterns of known intrusions are stored, future instances of these intrusions can be detected effectively and efficiently. However, newly invented attacks will likely go undetected.

In Section3.1, we discuss the data-mining algorithms that can be used to build both anomaly and misuse detection models.

3 Systematic Framework

Our framework consists of data-mining programs for learning detections models, a translator for converting learned rules to real-time models, and NFR for capturing network traffic and applying the real-time N-code modules for ID.

3.1 Data-Mining Algorithms

Data mining generally refers to the process of (automatically) extracting models from large stores of data [3]. The recent rapid development in data mining has made available a wide variety of algorithms, drawn from the fields of statistics, pattern recognition, machine learning, and databases. Several types of algorithms are particularly useful for mining audit data.

Classification : maps a data item into one of several pre-defined categories. These algorithms normally output "classifiers", for example, in the form of decision trees or rules. An ideal application in intrusion detection will be to gather sufficient "normal" and "abnormal" audit data for a user or program, then apply a classification algorithm to learn a classifier that can label or predict new unseen audit data as belonging to the normal class or the abnormal class. We use the package RIPPER [2] as our classification rule-learner.

Link analysis : determines relations between fields in database records. Correlations of system features in audit data can serve as the basis for constructing normal usage profiles. A programmer would have "emacs" highly associated with "C" files, for example. Observed deviations from these automatically learned associations may suggest an attack. We use the association rules algorithms [1] for this particular type of analysis.

Sequence analysis : models sequential patterns. These algorithms can discover what (time-based) sequence of audit events frequently occur together. These frequent event patterns provide guidelines for incorporating temporal statistical measures into

duration	the length (in seconds) of the connection
protocol_type	type of protocol being used (e.g. tcp, udp, icmp, etc.)
protocol	if the protocol is privileged ($\leq$ 1024) or not
flag	normally SF (successfully connected and terminated according to the protocols), but can be an error status such as REJ, S0, S1, etc.
urgent	is the "urgent" flag used in any of the data
wrong_size_rate	if packet is fragmented, how many are "wrong" fragments per second

Table 1: Within Connection

count	the count of such connections
rej_count	the count of connections to a service that get the flag "REJ" (i.e. a packet that has a flag SYN which is met by an RST packet from the receiving end)
S01_count	the count of connections to a service that receive an ACK on a SYN packet that they never sent
diff_hosts	the count of unique (different) destination hosts
diff_rate	diff_hosts / count

Table 2: Same Service

intrusion detection models. We use the frequent episodes algorithms [7] for this analysis.

A framework has been developed, first proposed in [4], of applying data mining techniques to build intrusion detection models. This framework consists of programs for learning classifiers as well as a support environment that enables system builders to interactively and iteratively drive the process of constructing and evaluating improved detection models. The end product of this process is a set of concise and intuitive rules (that can be easily inspected and edited by security experts when needed) that can detect intrusions. The rules are then subsequently ported over to N-code as sub-routines or independent functions.

3.2 Mining Data to Construct Attributes

In order for data-mining programs to compute effective intrusion detection models, we must first process and summarize packet-level network traffic data into "connection" records. We initially start out with the raw audit data (commonly `tcpdump` binary output) of the designated network we wish to monitor. This is then subsequently preprocessed into individual packets/events in ASCII format. As the packets are summarized according to their separate connections, we record their within-connection features which may be deemed as "traditional attributes" of a connection record. Refer to to Table 1 for examples of these attributes. We use the mined patterns from network connection records as guidelines to construct temporal statistical attributes for building classification models [5]. We performed pattern mining and comparisons using intrusion data of several well-known attacks, e.g., port-scan, ping-sweep, etc., as well normal connection records. Each of the unique intrusion patterns are used as guidelines for adding additional features into the connection records to build better classification models. These temporal and statistical attributes are shown in Table 2, and Table 3. *Same Destination* attributes deal with all connections to a particular host, and is not concerned with the number of ports or services being accessed; it only keeps track of the connections for the specified host. *Same Service* attributes deal with all connections to a particular port or service being sent throughout the network; it keeps track of the connections for the specified service. An N-code filter has been written for each of these attributes.

With the addition of these specific attributes to the standard features of connections, "rules" of intrusion detection models can be produced in NFR by machine learning via RIPPER.

3.3 Learning Detection Rules

We apply RIPPER to the connection records to generate the classification rules for the intrusions. Like other rule learning systems, it is used for classifications problems.

count	the count of such connections
rej_count	the count of connections that get the flag“REJ” met by a particular host
S01_count	the count of connections that send a SYN packet but never get the ACK packet (S0), or receive an ACK on SYN that they never have sent (s1)
diff_services	the count of unique (different) services
diff_srv_rate	diff_services / count

Table 3: Same Destination

Figure 1: Example tcpdump File

```
...
12:22:18.336681 im.a.hacker.com.2019 > your.machine.org.talk: udp 28 (frag 242:36@0+)
12:22:18.336681 im.a.hacker.com > your.machine.org: (frag 242:4@24)
12:22:18.356681 im.a.hacker.com.2019 > your.machine.org.talk: udp 28 (frag 242:36@0+)
12:22:18.356681 im.a.hacker.com > your.machine.org: (frag 242:4@24)
12:22:18.376681 im.a.hacker.com.2019 > your.machine.org.talk: udp 28 (frag 242:36@0+)
12:22:18.376681 im.a.hacker.com > your.machine.org: (frag 242:4@24)
...
```

A “training” period is initially required for RIPPER to gather the necessary data on the network to compute models. The purpose is two-fold:

1) establishing “normal” traffic patterns and variants that the network may encounter to establish anomaly detection,

2) introducing known intrusion methods and attack scripts into the network in order to inductively learn the classification models of intrusions

Here we provide example rules used to detect known attacks. In particular, we illustrate how to detect and recognize an attack which is categorized as *denial-of-service*.

Let us suppose that a hacker launches “teardrop” from machine im.a.hacker.com and is attempting to bring down the server your.machine.org. Teardrop transmits a number of overlapping fragmented UDP packets to a specified host. Fig. 1 shows the tcpdump data of such an attack.

Here, the data is first processed into connection records with the attributes described in section 3.2 along with the class label “teardrop”. RIPPER is then applied to these records to produce a teardrop rule shown to Table 4, for monitoring the teardrop intrusion.

While this may seem intuitive to a system builder, it is important to distinguish that our data mining programs has *automated* the process of generating such heuristics. Our system would then call upon the machine translator to compile the teardrop rule, and generate the appropriate N-code that filters out all network traffic except for the information concerning this particular type of attack.

3.4 Network Flight Recorder

NFR is a powerful software package that monitors and provides information concerning network traffic. It also analyzes the traffic and generates statistics that can then be displayed in a graphical format. We primarily chose NFR for the following reasons:

1) it does not interfere with network activity, necessary for accurate data analysis

2) it possesses a language flexible and portable enough to be programmed on an internal basis, rather than hard-coded in the monitoring application

3) it has real-time alert capability

NFR runs primarily on a packet-sniffing engine that is responsible for filtering and reassembling. While packets are passed through the NFR daemon, they are checked against a list of filters for evaluation. The filters, written in “N-code”, (an interpreted programming

Rule	Translation
if (protocol = UDP and wrong_size_rate ≥ 3)	(the current connection has succession of fragmented packets coming in at a rate over 3 packets/sec and the fragmentation is "wrong")

Table 4: Tear Drop Rule

language) perform the various functions and tasks that are activated by the incoming packets. [8]

In Appendix we have an example of the wrong_size_rate attribute concerning UDP packets.

3.5 Generation of N-Code Filters

As discussed earlier, the attributes of connection records are implemented as subroutines that may be called upon to check the rules that were generated by machine-learning.

Observe that a RIPPER rule simply consists of a sequence of attribute value tests. With each attribute implemented as an N-code filter, a rule can be automatically translated into an N-code filter that consists of a sequence of function calls to the N-code filters. For example, the "teardrop" RIPPER rule is translated into a "teardrop" N-code filter, as shown in Figure 2.

4 Experiments

Our network for intrusion detection research runs primarily on six hosts connected to a T1 subnet of a larger domain of a university LAN. NFR 1.6, runs on a Solaris X86 machine where all data recorded is contained on an external 6.0 GB SCSI HD. While a training period of 2-3 days has been allotted for this network, it is important to establish that training periods were continuously being conducted to fine-tune the accuracy of the detection. Traffic is relatively light in saturation, rarely ever going over 60-70 Megabits/sec.

The types of intrusions that we are primarily concerned with fall into 4 main categories:

1) denial-of-service (e.g., ping-of-death, SYN flood, smurf, teardrop, etc.)

2) unauthorized access from a remote machine (e.g. via guessing password)

3) unauthorized access to local superuser privileges by a local unprivileged user (e.g., various buffer overflow attacks)

4) surveillance and probing (e.g., port-scan, IPscan, etc)

We are gathering training data on these attacks, producing RIPPER rules, and converting the rules into NFR N-code filters.

4.1 Details

A connection filter is used to generate and store a record for each network connection. These connection records are kept in a global array, and are ordered by their timestamps (that is, the start time of the connections). This array is used for temporal and statistical analysis on the connections and for feature construction.

A connection record consists of a timestamp, the source and destination hostnames, the port numbers, the "traditional" attributes, and the temporal and statistical attributes.

4.2 Issues

The biggest issues we face are optimizing the connection filter, and excluding certain types of connections that may occur within the LAN that are proven to be harmless, but may overload the buffer allocated by the connection filter with the number of packets being generated.

Connections within a network may remain open for long periods of time and create large amounts of packet traffic. This can occur, for example, when exporting X window system and browser applications to a remote host. We consider all traffic generated by such remote applications to be part of one connection; consequently, keyboard, mouse, and display update operations can easily overflow the single connection's buffer. Currently our only workaround is to forbid such exported applications.

```
filter teardrop_rule(ip.src, conn_id, ip.protocol) {
 if (wrong_size_rate (conn_id) >= 3 && ip.protocol ==17)
           {
         $message = cat("This is a TEARDROP!* sent from " ip.src,   }
    }
  echo ($message);
           }
```

Figure 2: Automatically Generated N-code Filter for "teardrop"

To minimize dropped packets, NFR filters should reduce the volume of incoming packet traffic before forwarding to a backend; *en masse* recording is best done in the NFR engine itself. Our present connection filter consumes a substantial portion of the NFR host CPU, subsequently increasing the number of dropped packets from the engine.

Finally, an online IDS such as ours cannot afford the luxury of deferring the analysis of packet traffic; this analysis must be done in real-time and consumes additional CPU over offline methods. However, we hope to continuously improve the efficency of our filters by considering the computational cost of the attributes and prioritizing the rules that are being called.

5 Conclusions and Future Work

In this paper, we have outlined a data mining framework for building ID models. We describe how models produced via on-line analysis of audit data that can be automatically translated into NFR, a real-time IDS. Our experiences thus far show that our approaches are very effective.

We plan to port our system to the latest 2.0.3 commercial release of NFR, to take advantage of its new features, such as it's new functions and capabilities in N-code, a faster packet-sniffing engine, and the provision of a mini-Web Server which eliminates the need for an independent Web daemon to be running. Preliminary steps have already been taken to detect simple intrusions (ie portscan, ping-of-death, synflood) and while these rules derived from the attributes have proven to be successful, we would like to implement more complex attributes for sophisticated methods of intrusions.

6 Acknowledgments

We are very grateful to the NFR engineering team who have continuously provided us with support concerning the NFR software. We would like to also thank Matthew Miller from Columbia University for his help and encouragement.

References

[1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 207–216, 1993.

[2] W. W. Cohen. Fast effective rule induction. In *Machine Learning: the 12th International Conference*, Lake Taho, CA, 1995. Morgan Kaufmann.

[3] U. Fayyad, G. Piatetsky-Shapiro, and P. Smyth. The KDD process of extracting useful knowledge from volumes of data. *Communications of the ACM*, 39(11):27–34, November 1996.

[4] W. Lee and S. J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[5] W. Lee and S. J. Stolfo. A data mining framework for building intrusion detection models. In *1999 IEEE Symposium on Security and Privacy.*, Oakland, CA, May 1999.

[6] Teresa F. Lunt. Detecting intruders in computer systems. In *Proceedings of the 1993 Conference on Auditing and Computer Technology*, 1993.

[7] H. Mannila, H. Toivonen, and A. I. Verkamo. Discovering frequent episodes in sequences. In *Proceedings of the 1st International Conference on*

Knowledge Discovery in Databases and Data Mining, Montreal, Canada, August 1995.

[8] Inc. Network Flight Recorder. Network flight recorder. http://www.nfr.net, 1997.

Appendix

```
#   UDP wrong_size_rate Detection contains two filters:
#   First is the udpfrag filter which keeps an array indexed by
#   cat (ip.dst, "-", ip.src)
#   which keeps track of how many fragmented udp packets have been sent
#   between src and dst.
#
#  Second is the wrong_size_rate filter wich checks udpFrags every second to check
#  to see if hostpair has had more than m udpfrags where m is ALERT_NUM.

ALERT_NUM = 1;  # The number of frags in TEAR_FREQ seconds that should
                # trigger an alert.

TEAR_FREQ = 1;  # The frequency (in seconds) to check the frag count.

# This filter will be triggered by only udp packets
filter udpFragFilter udp () {
    # byte(packet.blob, 14) is the first byte of the ip header
    #   inside the ethernet frame
    # so: byte(packet.blob, 20) is the 7th byte of the ip header.
    # We only want to continue if more fragments bit is set.

    # The 18th and 19th bytes of the packet are the 5th and 6th bytes of
    # the ip header.  This is the Identification, which is only used if there
    # is fragmentation.  Thus, if it is zero, then there is no fragmentation,
    # and we don't need to look at the packet.
    if (short(packet.blob, 18) == 0) {
        return;
    }

    #   record system.time,
    #       $sport, $dport,
    #       ip.src, ip.dst
    #       to ipfrags_recorder

    $message = cat (ip.src, ":", $sport, " sent ",
                    ip.dst, ":", $dport,
                    " a fragmented UDP packet.");
    echo ($message);
    # assemble the index into udpFrags
    $index = cat (host(ip.dst), ":", $dport);

    # No one should be sending fragments whose size is not evenly
    # measurable in bytes!!

    if (ip.len % 8 != 0) {
        if (udpFrags[ [$myIndex, "badCount"] ] == NULL)
            udpFrags[ [$myIndex, "badCount"] ] = 1;
        else
        udpFrags[ [$myIndex, "badCount"] ] =
                udpFrags[ [$myIndex, "badCount"] ] + 1;
```

```
    #   Get together important info about the packet:
    #   We want to know the length of the packet, and we want the
    #   3-bit flags and 13-bit offset.  These are bytes 6 and 7
    #   of the IP header, or 20 - 21 of the ethernet packet.

    $pktInfo["ip.len"] = ip.len;
    $pktInfo["fragShort"] = short(packet.blob, 20);

    # We need 2 pkts to be able to check the offset correctedness.
    # If this is the first fragmented packet, we need to keep track
    # of the "fragShort" in order to be able to compute the offset in
    # subsequent packets, so we'll record it.

    if (udpFrags[$myIndex] == NULL) {
        udpFrags[$myIndex] = listadd(udpFrags[$myIndex], $pktInfo);
        udpFrags[ [$myIndex, "firstShort"] ] = $pktInfo["fragShort"];
        return;
    } else { # now we know it's a subsequent packet.
        udpFrags[$myIndex] = listadd(udpFrags[$myIndex], $pktInfo);
        $fragList = udpFrags[$myIndex];

        $n = listlen($fraglist);
        while ($n > 0) {
            $pkt1 = elem($fraglist, $n - 1);
            $pkt1Short = $pkt1["fragShort"];
            $pkt1Size = $pkt1["ip.len"];
            $pkt2 = elem($fraglist, $n);
            $pkt2Short = $pkt2["fragShort"];
            $pkt2Size = $pkt2Short["ip.len"];

            # Here we figure out the fragmentation offsets by subtracting
            # the "firstShort".  This eliminates the values of the 3-bit
            # flags, which we aren't concerned with at this point.

            $offset1 = $pkt1Short - udpFrags[ [$myIndex, "firstShort"] ];
            $offset2 = $pkt2Short - udpFrags[ [$myIndex, "firstShort"] ];

            # Here's where we test the offsets and increment
            # the count of udpFrags if they don't line up.
            if (($offset1 + $pkt1Size) != $offset2)) {
                if (udpFrags[ [$myIndex, "badCount"] ] == NULL)
                    udpFrags[ [$myIndex, "badCount"] ] = 1;
                else
                    udpFrags[ [$myIndex, "badCount"] ] =
                        udpFrags[ [$myIndex, "badCount"] ] + 1;
             } else # we're fine!!
                return;
            $n = $n - 1;
        } # close while
    } # close else
} # close func.

# Here we free up udpFrags.
func purge_udpFrags timeout (sec: (TEAR_FREQ + 1), repeat) {
```

```
    # if there aren't any entries, then exit
    if (!udpFrags) {
        return;
    }

    # empty all the entries.
    foreach $myIndex inside (udpFrags) {
        udpFrags[$myIndex] = NULL;
    }
}

# This is the main filter to check for the teardrop attach using the rule:
#      if ((udpFrags in TEAR_FREQ) >= ALERT_NUM)

$myIndex=$conn_Id
filter wrong_size_rate timeout (sec:1, repeat) {

    if (!udpFrags) {
        return;
    }

    foreach $myIndex inside (udpFrags) {
        # Here we give the alarm
        if (udpFrags[ [$myIndex, "badCount"] ] >= ALERT_NUM) {
            $message = cat ("*Wrong_Size_Rate*: ", ip.src, " sent ", $index,
                            " more than ", ALERT_NUM,
                            " wrongly fragmented UDP packets in ",
                            TEAR_FREQ, "seconds.");
            echo ($message);
        }
    }
  # Here we call our function to clear udpFrags.
    purge_udpFrags();
}
```