NAF: The NetSA Aggregated Flow tool suite

Brian Trammell – CERT/NetSA, Carnegie Mellon University
Carrie Gates[1] – CA Labs

ABSTRACT

In this paper we present a new suite of tools – NAF (for NetSA Aggregated Flow) – that accepts network flow data in multiple different formats and flexibly processes it into time-series aggregates represented in an IPFIX-based data format. NAF also supports both unidirectional and bidirectional flow data by matching uniflows into biflows where sufficient information is available. These tools are designed for generic aggregation of flow data with a focus on security applications. They can be used to reduce flow data for long-term storage, summarize it as the first step in numerical analysis, or as a back-end to flow data visualization processes.

Introduction

Many organizations, including universities, private industry, Internet Service Providers (ISPs) and government organizations, are monitoring and storing network traffic information. Due to the overwhelming volume of network traffic, many of these organizations have chosen flow formats over full packet capture or packet header information. This information is then used for a variety of purposes, such as billing or capacity planning, with security monitoring being particularly common.

A number of tools have been produced in the past few years for processing flow data for both network usage and security analysis. For example, OSU FlowTools [4], argus [10] and SiLK [5] each have a significant user base. However, each of these programs produces flow data in a different format, the tools from each are not interoperable, and there are no programs that can perform consistent analysis across each of the different formats.

In this paper we present the NAF (NetSA Aggregated Flow) suite of tools. These tools have been designed for interoperability with a variety of flow collection systems and analysis tools, with a focus on common security analysis tasks using time-series aggregated flow data. To that end, the nafalize flow aggregator can read raw flow data from multiple input formats (e.g., SiLK, argus) as well as from data sources supporting the IPFIX [1] standard for flow export. To bridge the gap between unidirectional and bidirectional flow data formats, nafalize can match uniflows into biflows if both directions are available. The nafscii flow printer converts aggregate flow data into whitespace-delimited text for simple import into a variety of numerical analysis tools.

nafalize was designed to be a general-purpose flow aggregator. It provides a flexible aggregation facility for generating time-series aggregated flow data, supporting aggregation by any set of flow key fields, and counting of octets, packets, and flows in the aggregate, as well as by distinct source and destination addresses. It includes a facility for sorting aggregated flow data and limiting output to produce time-series Top-N and watch lists. The nafilter tool allows further filtering and sorting of this aggregated flow data.

We begin with an overview of related work, focusing on different suites of flow tools and the functionality they provide. We then describe the NAF suite of tools, including the functionality of each of the tools along with a description of the options available. We present details of the design and implementation of the nafalize and nafilter tools and subsequently provide some usage examples that have been deployed in an operational context. The last section includes concluding comments and plans for future work.

Related Work

The conversion and aggregation of flow data for analysis purposes is certainly not a new area of work, and there exist several other tool sets which address some of the problems the NAF suite was created to address. Most of NAF's functionality can indeed be duplicated by stringing together combinations of these tools; we believe that what NAF adds to the state of the art is the combination of flexible, time-series aggregation and sorting of flow data with multiple input format support in a single, relatively easy to use package. We examine some of these other tools below.

In its default mode of operation, the ipaggcreate tool from the ipsumdump [3] suite, maintained by Eddie Kohler at UCLA, works much like nafalize and nafscii in series, with two important differences. First, ipaggcreate is primarily designed to read various packet trace formats; "flow-like" input support is limited to Cisco NetFlow summary files and Bro connection summaries. Second, ipaggcreate has no notion of time series; it is analogous to running NAF with a bin size larger

[1] This work was performed while with the CERT Network Situational Awareness Group at Carnegie Mellon University.

than the entire scope of its input; using ipaggcreate to generate time-series aggregates would require multiple invocations.

The OSU FlowTools [4] flow-report tool can produce the same type of aggregate reports, with the same caveat that it does not natively support time series data.

The SiLK [5] flow analysis suite can be used to do many of the aggregation tasks performed by nafalize and nafilter, though somewhat less easily.[2] A set of rwfilter invocations can be used to filter flows, simulating flow key masking for limited key spaces. The rwcut tool provides only simple time binning and aggregation. nafalize also supports biflow data natively, and unique host and port counting, which are not presently provided by SiLK.

The FlowScan [2] tool, maintained by Dave Plonka from the University of Wisconsin, is another time-series flow analysis environment. Unlike NAF, it is focused specifically on visualization of summary time series flow data, and as such uses the round-robin database provided by RRDTool [13] for data storage and aggregation. However, it does support import of flow data in multiple formats; including CAIDA's cflowd, the OSU flow-tools package, QoSient's argus, and the Lightweight Flow Accounting Protocol.

NCSA's CANINE [8] tool performs largely the same function as nafalize's first stage, transcoding among a variety of flow formats as a conversion front-end for NCSA's flow data visualization tools. CANINE is also capable of IP address and time sequence anonymization. However, it does not itself provide any support for aggregation of flow data.

Description

The NAF suite presently consists of four tools: nafalize, which aggregates flow data into a common aggregated flow data format (the NAF data format, based upon IPFIX); nafilter, which sorts and filters NAF data; nafscii, which prints NAF data as white-space-delimited ASCII text; and nafload, which loads NAF data into a relational database. The NAF data format itself is handled by libnaf, a library installed with the NAF tools; this arrangement allows the easy creation of tools that interoperate with NAF.

All the tools support common options for input and output routing. They can each run as command-line tools or as daemons. In the latter case, the tools can watch directories for input files, processing them as they appear and forwarding the output to other directories; in this manner, chains of daemons can be built to support specific workflows.

nafalize reads raw flow data in a variety of formats, filtering, aggregating, and sorting it into the NAF data format. The aggregation function performed by nafalize is specified on the nafalize command line by an aggregation expression. This expression maps relatively tightly to the design of nafalize; that is, each "clause" in the expression corresponds to a particular process within nafalize's data flow. The aggregation expression is described by the following pattern:

```
bin [uniform | start | end] <n>
     (sec | min | hr)
[uniflow] [perimeter <perimeter-ranges>]
[<filter-expression>]
aggregate [sip[/<mask>]] [dip[/<mask>]]
            [sp] [dp] [proto] [sid]
   count [hosts] [ports] [total]
         [flows] [packets] [octets]
   [<filter-expression>]
   [<sort-expression>]
   [label <output-label>]
[aggregate ...]
```

Note that multiple aggregate clauses are supported; this is used to specify fanout as in the "Fanout" section in "Stage 3," below. The label phrase serves to differentiate output files in this case.

The first filter expression is applied to the entire data set so that all later processing is performed only on this filtered data (and is described more fully in the "Stage 2" section, below). The second filter expression applies solely to the individual aggregation (see the "Stage 3" section). Thus each individual aggregation can be on data that has been filtered differently. The filter expression itself is described by the following pattern:

```
filter
[time <time-rangelist>]
[sip [not] <ipaddr-rangelist>]
[dip [not] <ipaddr-rangelist>]
[sp [not] <unsigned-rangelist>]
[dp [not] <unsigned-rangelist>]
[proto [not] <unsigned-rangelist>]
[flows <unsigned-rangelist>]
[packets <unsigned-rangelist>]
[octets <unsigned-rangelist>]
[shosts <unsigned-rangelist>]
[dhosts <unsigned-rangelist>]
[sports <unsigned-rangelist>]
[dports <unsigned-rangelist>]
```

The sort expression defines the ordering of records in the output. The output is always sorted in time order first. Within each bin, output records are given in arbitrary order by default. If a sort expression is present, output records within each bin are sorted by the fields given in ascending or descending order. The limit phrase limits the output to the specified number of records per time bin; it can be used to generate top-N lists. The sort expression is described by the following pattern:

```
[sort (flows | packets | octets |
       shosts | dhosts | sports |
       dports | sip | sp | dip |
       dp | proto)
    [asc | desc]]
[sort ...]
[limit <n>]
```

[2] Indeed, one of the initial motivations behind NAF's creation was to provide an easier method of producing time-series aggregates and unique counts from SiLK data.

nafilter reads NAF aggregated flow data, filters and sorts it, writing NAF aggregated flow data. As with nafalize, nafilter's operation is specified by a filter/sort expression, which has the same mapping to nafilter's internals as the aggregation expression above. The filter/sort expression is merely a filter expression followed by a sort expression, as defined above.

Each of the clauses in the expressions corresponds to a process in the data flow of each application; see below for more.

nafscii is the NAF flow printer. It reads NAF aggregated flow data and writes it out as whitespace-delimited text. It is used for simple aggregated flow display, and for exporting NAF aggregated flow data to other analysis tools (*e.g.,* the R [12] statistical computing and visualization environment) which can handle whitespace-delimited data.

nafload is an online loader for inserting NAF aggregated flow data into a relational database. It was built largely to support the internal workflow of a specific project at the CERT Network Situational Awareness Group.

Both nafscii and nafload are extremely simple in design, reading in NAF aggregate flow records and writing them out after transforming them; they will therefore not be considered in further detail in this paper.

Reference the manual pages in the NAF distribution for detailed usage instructions (available at http://www.cert.org/netsa/tools/naf).

Design and Implementation

In this section, we describe the common data model and storage format these tools use, then explore the design of nafalize and nafilter in detail.

Data Model

NAF's internal data model is based on time-binned, aggregated, bidirectional flows. Each NAF record represents a collection of flows sharing a given flow key or subkey within a given finite time period

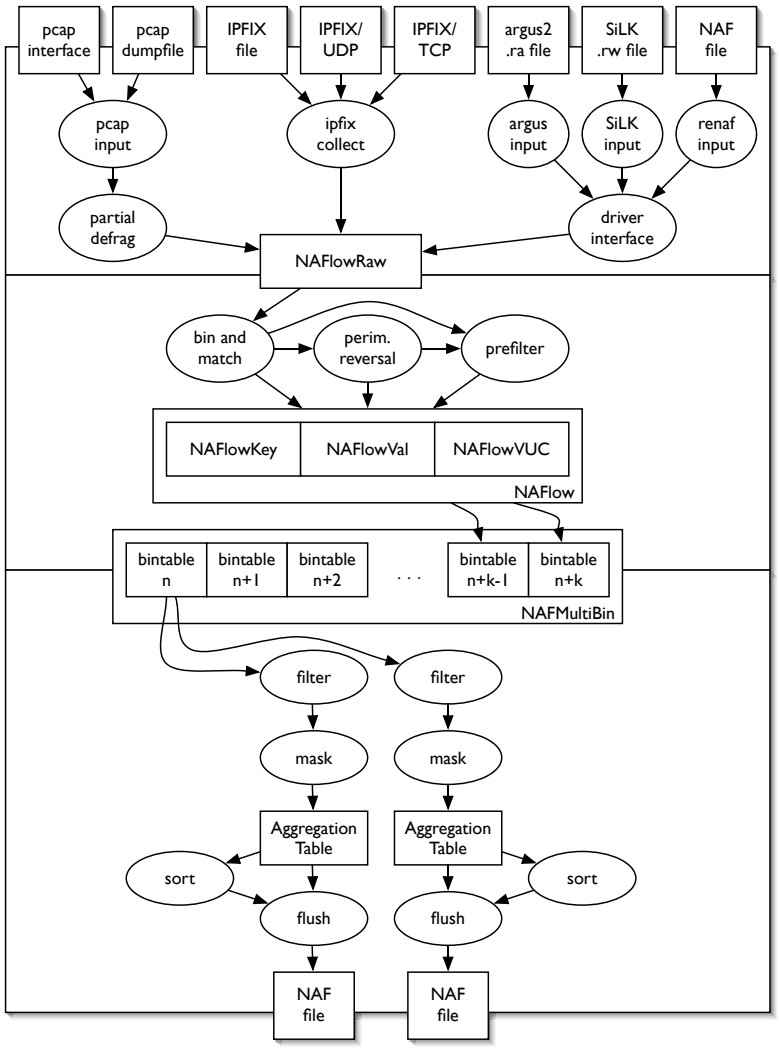

Figure 1: Schematic diagram of nafalize.

(bin). Flow keys (type NAFlowKey in the NAF source) are composed of some combination of source and destination IP address and prefix length, IP protocol, source and destination port, and source (or sensor) identifier. Flow values (type NAFlowVal) are composed of six aggregate counters (for octets, packets, and flows, each in the forward and reverse directions) and four unique value counters (for distinct source and destination IP addresses, and distinct source and destination ports).

NAF uses an IPFIX-based external data format, as described in the IPFIX Protocol Specification [1] and the IPFIX Information Model [11]. Bidirectional flow information is represented as in Biflow Implementation Support in IPFIX [14]. IPFIX was chosen for its self-describing nature; the ability to dynamically define record templates, useful because nafalize can produce output with any subset of key and value fields; and for ease of present and future interoperability with IPFIX-based flow metering and collection tools. Each NAF data file is a serialized IPFIX message stream containing the IPFIX templates required to interpret the records contained within the file.

nafalize

nafalize is the NAF flow aggregator. It is an input-driven application, capable of converting a variety of flow formats. It can read flow data from files, function as an IPFIX Collecting Process (reading IPFIX messages over TCP or UDP), or capture packets from an interface via libpcap.

nafalize is roughly organized into three stages, with well-defined interfaces between them. These three stages are raw flow input, binning and matching, and aggregation and output; they are described below. A schematic diagram of nafalize appears in Figure 1.

It should be noted that presently, all of these stages run in series – for example, when a packet or flow is received from the first stage that causes a new bintable to be enqueued, and a bintable is dequeued, that bintable is aggregated and flushed (multiple times, in case of fanout) before the next packet or flow is processed. This may lead to dropped data when NAF is collecting data from a live interface or via IPFIX over UDP. The strict layering of these stages was chosen for future performance enhancement; by splitting each stage into its own thread for running on its own processor in a multiprocessor system, for example.

Stage 1: Raw Flow Input

nafalize supports three raw flow input types: multi-format flow input from files; IPFIX over TCP, UDP, or serialized streams via libfixbuf; and packet capture via libpcap [7]. A given nafalize invocation can only read input from a single type.

The file input facility includes flow format drivers for reading from QoSient Argus [10] 2.0.6, CERT/NetSA SiLK [5], and NAF files themselves for re-aggregation. New flow format drivers are relatively simple to add, but at this time require integration into the NAF source code; there is no support for dynamically-loaded flow format drivers.

The IPFIX input facility can read from serialized IPFIX message stream files, and IPFIX messages via UDP or TCP. It is capable of reading both unidirectional and bidirectional flows.

The packet capture facility can read from pcap dumpfiles (as produced by tcpdump -w), or from a live ethernet or loopback interface via libpcap. It does partial fragment reassembly – enough to ensure subsequent fragments of a fragmented datagram are accounted to the correct source and destination UDP or TCP port. When capturing packets, nafalize simulates counting TCP ‘‘flows’’ by looking for SYN or SYN+ACK packets. Each packet, SYN or not, is then treated as a separate raw flow for purposes of binning and matching, as below.

Each of these facilities successively reads flow or packet records from their respective sources, and normalizes them into NAF raw flow records (type NAFlowRaw), which are then passed to the second stage.

Stage 2: Bin and Match

The second stage of nafalize consists of four processes: *binning*, *perimeter reversal*, *prefiltering*, and *matching*. Perimeter reversal and prefiltering are optional, depending on configuration.

Each raw flow is first split into time bins. If a raw flow’s start and end times fit entirely within a single bin, that raw flow is assigned to the bin in which it fits. Otherwise, the flow is assigned to bins according to a user-selectable bin selection strategy. Three of these strategies are presently supported: start, end, and uniform. The ‘‘start’’ strategy bins the raw flow completely into the bin containing the flow’s start time; conversely, the ‘‘end’’ strategy bins the raw flow completely into the bin containing the flow’s end time. The ‘‘uniform’’ bin selection strategy divides the flow’s values equally into each bin covered by the time span between the flow’s start and end time, preserving remainders so that values are robust across re-aggregation.

Note that, as a packet has only a single timestamp, every packet-derived ‘‘raw flow’’ will always fit into a single bin. Therefore, packet capture input has the effect of ‘‘start’’ binning no matter what bin selection strategy is employed.

NAF supports optional perimeter reversal of flows. If the user specifies a network perimeter based on a set of IP address ranges and/or CIDR blocks, all raw flows are conditionally reversed such that addresses external to the perimeter are ‘‘source’’ addresses, and addresses internal to the perimeter are

"destination" addresses. Flows not crossing the perimeter are dropped. This is compatible with the semantics of some security tools, such as snort and QRadar, which typically assign source addresses to the "attacker." This facility is provided for users of such tools, who are more comfortable thinking of networks in terms of "interior" and "exterior."

After perimeter reversal, each binned flow is then subjected to an optional prefilter. See the "Filtering" section below for a detailed description of filtering in nafalize. Prefiltering is provided for performance improvement. Though each flow can also be filtered after matching, matching requires a flow to be held in memory until it is ready for aggregation. Therefore, early elimination of irrelevant flows before matching will increase nafalize's performance.

The binned flows are then inserted into the multibin (type NAFMultiBin). This structure is a bin-indexed, time-ordered queue of flow tables (type NAFBinTable). The appropriate bin table is accessed by time bin; if no table exists yet for a given bin (because a binned flow is the first one assigned to the given bin), new bin tables are created and inserted at the head of the queue.

The multibin's queue length is determined by the *horizon*. This horizon h is chosen such that the processing of a flow of start time t enables the assumption that no flows with a start time before $t - h$ will be processed subsequently. When processing raw flow or IPFIX data, this is generally set to the active timeout interval of the flow metering process; for packet capture input, the horizon can be the same as the bin size, keeping only one bin table active at once. This design implies that NAF's input must be at least roughly sorted by time.

When new bin tables are enqueued at the head, old bin tables expire from the tail. These bin tables are dequeued, and passed on to the third stage.

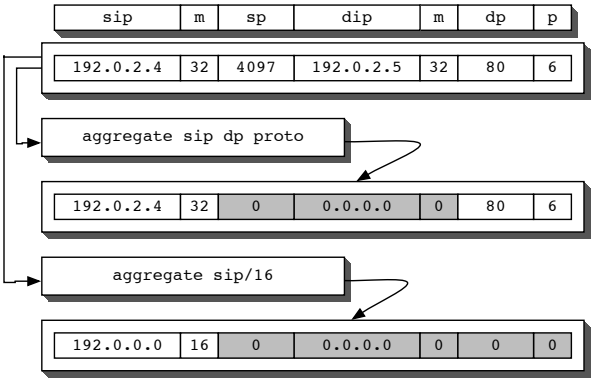

Figure 2: Illustration of mask operation.

Stage 3: Aggregate and Output

The third stage of nafalize consists of five processes: *filtering*, *masking*, *aggregation*, *sorting*, and *output*. As with prefiltering in the second stage, filtering is optional, and is described in the "Filtering" section below.

Each binned flow that passes the optional filter is masked. Masking consists of projecting each record's flow key into a subset of the flow key. Subkeys are derived either by dropping fields from the key or by masking IP addresses by a more restrictive prefix length. The masking process is illustrated in Figure 2.

This subkey is then used to create and update aggregate flow records into an aggregation table. The values of each binned flow are added to the corresponding aggregate flow record, and distinct values for flow keys dropped from the subkey (such as hosts or ports) are counted. Once all the binned flows in a bin table have been aggregated, the aggregation table is optionally sorted as described in the "Sorting" section below, then written to the output file.

Fanout

NAF is capable of *fanout*; that is, it can run multiple third stages off a single second stage. Each of these third stages has a different filter and mask, and a separate aggregation table, and writes to a different file. This can lead to significant performance improvement over processing the same data twice, because in many cases the second stage is much more memory- and CPU-intensive than the third.

Filtering

As noted above, nafalize may optionally filter raw flows before binning and matching, and binned flows before aggregation. The filtering facility is identical in either case. Each filter is built from a user-supplied filter expression, and contains one or more field specifiers (e.g., source IP, protocol) and ranges of acceptable values for the given field. If a field specifier is present in a filter, then that field must have one of the values in the associated range in order for the flow to pass the filter. Flows that do not pass the filter are simply dropped.

This filtering algorithm does not support arbitrary boolean predicates; instead, it can be viewed as the intersection of a set of unions (or "AND-of-OR").

When filtering on time ranges, a flow matches a time range if the flow's start time falls within the time range. For purposes of filtering, the start time of a binned flow is the bin's start time.

Sorting

As noted above, nafalize may optionally sort aggregated flows on output. If a sort expression is supplied by the user for a given aggregation, all aggregated flows in each time bin are placed into an array on output, then sorted using a sort comparator derived from the sort expression. Note that this design constrains the output to be sorted in ascending time order.

The sorting function also supports a limit, which will output only the first N flows per bin in sort order.

In this way nafalize can be used to build time-series Top-N lists.

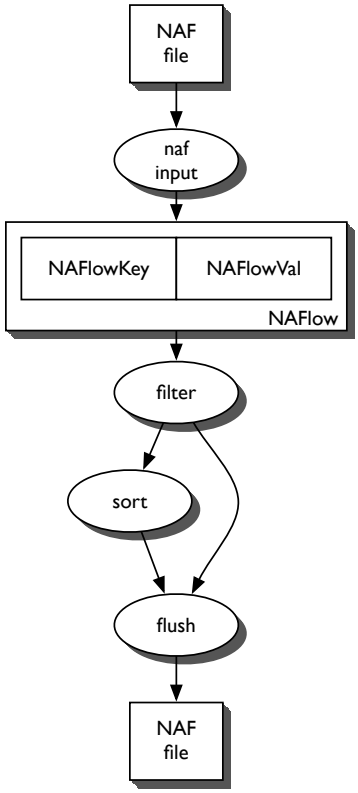

Figure 3: Schematic diagram of nafilter.

nafilter

nafilter is the NAF aggregated flow filter/sorter. Like nafalize, it is an input-driven application, though it is limited to only reading files written by nafalize. It is roughly organized into two stages: filtering and optional sorting. A schematic is shown in Figure 3.

Filtering operates as described in the ''Filtering'' section; indeed, the filter implementation is shared between nafalize and nafilter. It is important to note that the filter operates on raw flows in nafalize but on aggregated flows in nafilter, so filtering on value fields will have different results in each application. Consider the example of a filter that passes only records with a packet count of one. In nafalize, this filter will build aggregates of single-packet raw flows, while in nafilter, it will only pass aggregate flows that themselves only have a single packet.

Sorting operates as described above; the sorting implementation is also shared with nafalize.

Usage Examples

Here we describe two examples of actual NAF deployments in operational contexts. The first is as part of an internal data collection project at the CERT Network Situational Awareness Group. The second deployment occurred on the network of a large computing conference as part of a security support effort using SiLK and NAF.

NetSA Preanalysis

NAF is used in the preprocessing of Argus 2.0.6 flow data from a distributed collection infrastructure operated by the Network Situational Awareness group at CERT. The generated aggregate flows support ''at-a-glance'' visualization and statistical anomaly detection.

Raw Argus flow files (generated by argus and preprocessed themselves through ragator) are aggregated into four separate labeled files (using the fanout feature described below). These files are then loaded via nafload into a relational database, from which further analyses are done.[3] The nafalize command line for this output is:

```
nafalize --daemon --lock --intype argus2
 --nextdir argus-cache --faildir naf-fail
 --in "argus/*.rag" --out naf-out
 bin 5 min
 aggregate count flows packets octets
 label volume
 aggregate count hosts
 label talkers
 aggregate sip count flows octets
 label sources
 aggregate dp proto count hosts octets
        packets flows
 filter proto 6,17
 label pdb
```

The four labeled files contain, in 5-minute time series: total flow, packet and octet volume (example nafscii output in Figure 4); total distinct source and destination hosts (Figure 5); total flows and octets per source IP address (Figure 6); and total distinct source and destination hosts, flow, packet, and octet counts from per destination port and protocol (Figure 7).[4]

The relational database into which the aggregate flow data is loaded is presently used for two purposes.

First, periodic queries are run against the relational database, and the results are fed into Tobias Oetiker's rrdtool [13] to generate time-series graphs for each of the variables produced (e.g., total data volume per sensor in octets, talkers per sensor, etc.). This provides a simple ''dashboard'' visualization for the distributed collection system.

Periodic queries are also used to select all the variables available for a given time bin; these are used

[3] This is not precisely how this works in production. First, the nafalize command line is slightly different due to deployment concerns. Second, nafalize is run twice for site-specific reasons, with the second nafalize instance processing the output of the first instance, using naf's ability to reaggregate its own output. Third, the aggregation expressions in production use the srcid field to aggregate data from multiple sensors.

[4] The example data was not generated from production data from the distributed collection system; it is presented as an example of output from the command-line shown.

as independent-variable input into a statistical anomaly detection process based upon Mahalanobis distance [6], which compares each bin to a baseline derived from a larger window of recent aggregate data, and detects time bins which deviate significantly and therefore may benefit from further analysis of the full-flow data. The result of this process is a single time-series ''deviance'' metric, which is itself fed into rrdtool as above.

Note that we store time-series summary flow data in the relational database, and keep raw flow data in its native (Argus) format for a period of time to permit detailed flow analysis. NAF was deployed in this environment to replace a system which inserted raw flow data into the database for intermediate-term storage, and where all aggregation was done via SQL queries. This system did not scale adequately for our needs; a more detailed look at the use of relational database technology in raw flow storage is given in [9].

Conference Security

SiLK [5] and NAF[5] were deployed as part of an effort to provide security support for a large computing conference in late 2005. The example usage and output results presented here were all gathered from this conference. We have represented IP addresses internal to the security conference facility as 192.168.128.0/17, while external addresses have been randomly chosen from 241.0.0.0/8.

NAF is used here as a post-processor for SiLK raw flow data; the per-key binning provided by nafalize

[5]This deployment used an earlier version of the NAF tools which did not support fanout and consequently used a slightly different aggregation expression; the examples have been corrected to use aggregation expressions that will operate with NAF as it is presently available.

date	time	flo	rflo	pkt	rpkt	oct	roct
2006-03-03	13:20:00	363	12	4873	5552	739388	5956007
2006-03-03	13:25:00	279	16	7026	7612	1337665	8042156
2006-03-03	13:30:00	343	11	2599	2208	639824	1616504
2006-03-03	13:35:00	190	9	1355	1077	311763	729521
2006-03-03	13:40:00	223	6	1631	1422	320408	1040939
2006-03-03	13:45:00	223	7	5031	6147	653908	7736319

Figure 4: Argus preprocessing example volume output.

date	time	shosts	dhosts
2006-03-03	13:20:00	35	62
2006-03-03	13:25:00	37	60
2006-03-03	13:30:00	37	70
2006-03-03	13:35:00	31	38
2006-03-03	13:40:00	34	50
2006-03-03	13:45:00	28	48

Figure 5: Argus preprocessing example talkers output.

date	time	sip	flo	rflo	oct	roct
2006-03-03	13:20:00	10.146.0.13	1	0	64	0
2006-03-03	13:20:00	10.146.0.73	27	2	91604	433619
2006-03-03	13:20:00	10.146.0.74	14	0	1436	837
2006-03-03	13:20:00	10.146.0.77	23	0	9286	15266
2006-03-03	13:20:00	10.146.0.82	27	3	7766	5544
2006-03-03	13:20:00	10.146.0.83	14	0	4647	22963
2006-03-03	13:20:00	10.146.0.91	11	0	23202	31724
2006-03-03	13:20:00	10.146.0.95	8	0	3618	35124
2006-03-03	13:20:00	10.146.0.99	2	0	56	0

Figure 6: Argus preprocessing example sources output.

date	time	dp	proto	shosts	dhosts	flo	rflo	pkt	rpkt	oct	roct
2006-03-03	13:20:00	22	6	1	1	4	0	42	47	3910	7894
2006-03-03	13:20:00	80	6	5	15	81	0	1033	1141	107152	1120657
2006-03-03	13:20:00	143	6	1	1	2	0	48	49	2534	34490
2006-03-03	13:20:00	443	6	4	7	40	0	423	431	64404	282673
2006-03-03	13:20:00	445	6	3	1	3	0	3	0	144	0
2006-03-03	13:20:00	993	6	1	1	2	0	4	2	320	266
2006-03-03	13:20:00	53	17	8	6	53	0	91	55	6411	11104
2006-03-03	13:20:00	67	17	5	2	5	0	9	4	3024	1312
2006-03-03	13:20:00	137	17	7	3	9	0	54	0	4203	0
2006-03-03	13:20:00	138	17	5	1	6	0	7	0	1615	0
2006-03-03	13:20:00	5353	17	3	1	6	0	11	0	1365	0

Figure 7: Argus preprocessing example pdb output.

is more convenient to use than the equivalent operations using the SiLK rw tools alone, and SiLK did not at the time of this deployment support unique host or port counting as in the third example.

In our first example, we first use the SiLK tools (rwfilter) to filter the data to include only those flows that originated from within the internal network, but that were not destined for an internal address. One hour of such TCP data is piped into the following command:

```
nafalize -t silk bin 1 hr
 aggregate sip
 count hosts | \
nafscii | \
gawk '{if ($1 !~ /date/) { split($3, a, ".");
 printf "%d | %d0, a[1]*256*256*256 +
 a[2]*256*256 + a[3]*256 + a[4], $4}}'
```

The nafalize command aggregates the results from the SiLK commands into one hour bins by source IP address and counts the total number of hosts contacted. These results are converted into ASCII (via nafscii). The gawk command takes the IP address, which is provided in dotted quad notation, and converts it back to its integer form, printing this value along with the number of hosts contacted by that IP. The results from this command have the following form:

```
2363326977 | 1
2363326978 | 1
2363326980 | 2
2363327150 | 32
2363327161 | 7
```

These results are then presented graphically (hence the requirement for an integer representation of the addresses) so that a user can gain a sense over time of what was considered normal activity for the network. We present an example graph in Figure 8. This figure indicates five outliers that are potentially worth further investigation.

Figure 9 demonstrates a second command run periodically on the conference network. In order to detect unusual TCP activity, we select TCP traffic that was inbound to the network and that did not originate from within it over a one day period, restricted to only those flows representing failed connection attempts (i.e., where the SYN flag was set but not the ACK flag). This selection is done via the SiLK rwfilter command. We again nafalize this into one hour bins by source IP address, counting packets and bytes. nafscii then converts the output for processing by gawk to print the average number of bytes per packet observed, the number of packets, the start date and time for the record and the source IP address in dotted quad. As the selected flows represent failed connection attempts, we would expect the majority of traffic to be 40, 44, 48 or 52-byte single-packet flows. However, here we observe unusual activity. We expect that the cases where there are a large number of packets that average 40 or 44 bytes per packet are indications of scanning activity. For example, the case where IP address 241.194.61.230 has 14,374 packets with an average of 59 bytes per packet might indicate someone who was scanning and trying an exploit against those hosts that responded. This would indicate an IP address whose traffic warrants further investigation.

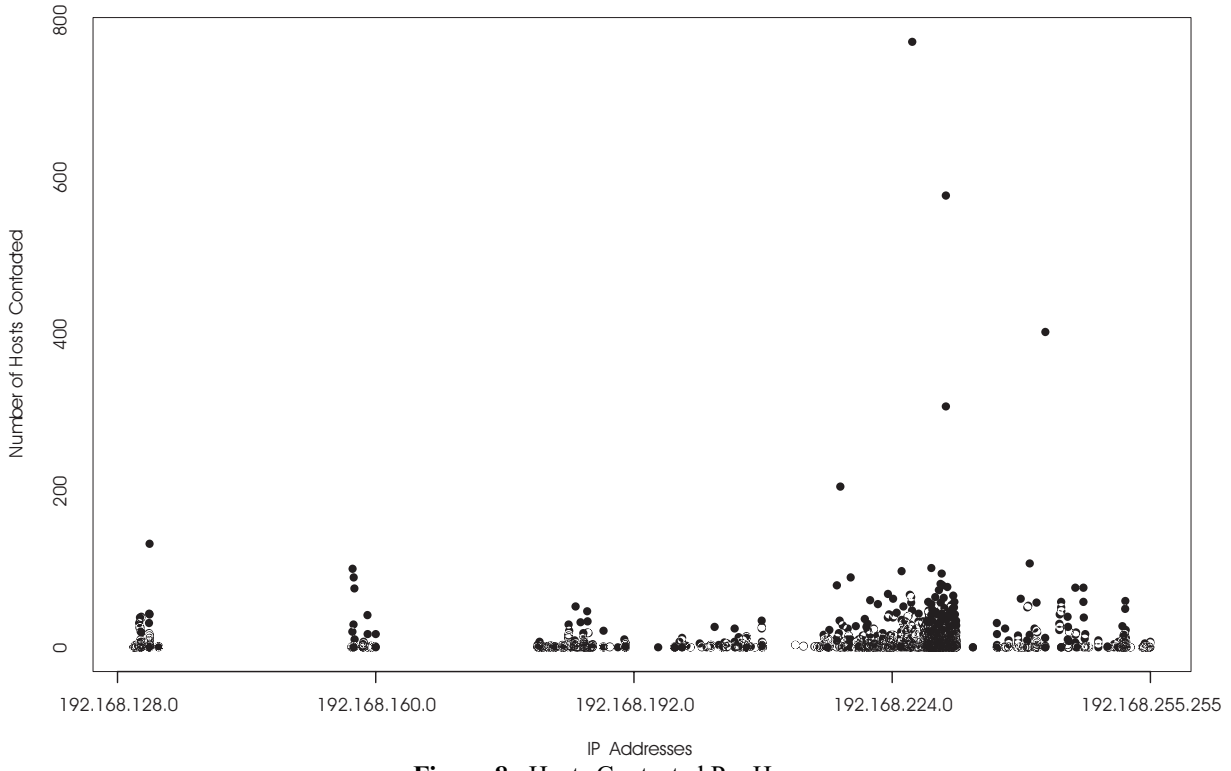

Figure 8: Hosts Contacted Per Hour.

In the third example (Figure 10), we examine all TCP traffic for a single day that is incoming to the conference network and that did not originate from within it. Again we aggregate in one-hour bins by source IP address. We convert to ASCII and process the results, printing out the integer value for the IP address, the number of destination hosts contacted, the start date and time for the record, and the dotted-quad version of the IP address. As there is little reason for an external IP address to connect to a large number of internal IP addresses, this analysis indicates likely scanning activity. Note that IP 241.194.61.230 appears again in this data, lending credence to our hypothesis above regarding their activity.

Our last example demonstrates commands run periodically to detect potentially compromised internal machines. In this case we select TCP flows from an external host to an internal host on port 445, lasting more than thirty seconds, representing completed connections where more than 1400 bytes were sent. This should extract those hosts that might have compromised the SMB port on a Windows machine. We then use NAF in order to bin on an hourly basis, extracting the start date and hour, the source and destination IP address, the number of flows and the number of bytes. The results from running this command are provided in Figure 11. In this case we find two IP addresses that have potentially compromised a single internal host each.

During the conference, we performed a similar analysis on ports 135 and 22. While much of the traffic destined for port 22 was legitimate, we examined the number of internal IP addresses to which different external IP addresses had connected. We also examined where the external IP addresses were registered, to ensure that they matched known participants rather than likely compromisers.

```
% rwfilter --syn=1 --ack=0 --daddr=192.168.128.0/17 \
    --not-saddr=192.168.128.0/17
    --pass=stdout --proto=6 --start=2005/11/15 | \
    nafalize -t silk bin 1 hr aggregate sip count packets octets | \
    nafscii | \
    gawk '{if ($1 !~ /date/) printf "%d | %d | %s %s %s \n", \
        $5/$4, $4, $1, $2, $3}'
 40 |      1 | 2005-11-15 00:00:00 241.192.13.14
 52 |      2 | 2005-11-15 00:00:00 241.10.21.189
 48 |      1 | 2005-11-15 00:00:00 241.11.246.197
 44 |   1929 | 2005-11-15 00:00:00 241.34.98.164
 59 |  14374 | 2005-11-15 00:00:00 241.194.61.230
 40 |  23347 | 2005-11-15 16:00:00 241.192.100.123
740 |      1 | 2005-11-15 23:00:00 241.71.1.154
```

Figure 9: Inbound bytes-per-packet by source.

```
% rwfilter --daddr=192.168.128.0/17 --not-saddr=192.168.128.0/17 \
    --pass=stdout --proto=6 --start=2005/11/15 | \
    nafalize -t silk bin 1 hr aggregate sip count hosts | \
    nafscii | \
    gawk '{if ($1 !~ /date/) { \
        split($3, a, "."); printf "%d | %d | %s %s %s \n", \
       a[1]*256*256*256 + a[2]*256*256 + a[3]*256 + a[4], $4, $1, $2, $3}}'
1000079635 | 1 | 2005-11-15 21:00:00 241.156.1.19
1006778434 | 2 | 2005-11-15 23:00:00 241.2.56.66
1022911850 | 3 | 2005-11-15 03:00:00 241.248.101.106
3514611848 | 283 | 2005-11-15 20:00:00 241.124.184.136
3703717350 | 14509 | 2005-11-15 00:00:00 241.194.61.230
1022903300 | 15881 | 2005-11-15 16:00:00 241.248.68.4
```

Figure 10: Inbound destination host count by source.

```
% rwfilter --bytes=1400-99999999 --dur=30-3600 --dport=445 --ack=1 \
    --flags-initial=S/SA --start=2005/11/15 --proto=6 --pass=stdout \
    --not-saddr=192.168.128.0/17 --daddr=192.168.128.0/17 | \
    nafalize -t silk bin 1 hr aggregate sip dip count total flows octets | \
    nafscii | \
    gawk '{if ( $1 !~ /date/) { if ($6 > 10000) \
        printf "445 | %s %s | %s %s | %d | %d\n", $1, $2, $3, $4, $5, $6}}'
445 | 2005-11-15 10:00:00 | 241.146.88.212 192.168.190.233  | 18 | 1209855
445 | 2005-11-15 11:00:00 | 241.146.88.212 192.168.190.233  |  1 | 44824
445 | 2005-11-15 16:00:00 | 241.214.211.244 192.168.190.248 | 3  | 231372
```

Figure 11: Inbound potential SMB compromise detection.

Conclusions and Future Work

We have introduced a new flow aggregation tool suite, the focus of which is interoperability with multiple flow sensor technologies and the reduction of flow data for network security analysis purposes. These tools are designed to be reasonably generic, and apply to a wide variety of analysis tasks. We have explored the design of two of these tools in detail, and provided examples of their usage in two real-world applications.

NAF is under continuing active development at the CERT Network Situational Awareness Group. We have planned several enhancements for the tool suite that will appear in future releases:

NAF's internal data model and aggregation operations presently only support flows with IPv4 addresses; we plan to add support for IPv6 addresses, as well.

To support NAF's use in data sharing applications, future releases of the tool suite will include support for data anonymization, when the aggregation operations do not discard sufficient information to meet an organization's dissemination policy needs. Indeed, the extent to which aggregation operations obfuscate data needs to be better quantified; this is an area for future research.

The use of a single data format at the core of the NAF tools' design also makes it reasonably easy to build new consumers for that data; currently planned is a NAF-to-SVG ''printer'' analogous to nafscii or nafload. This would allow the generation of time-series graphs from NAF data without the present need to convert the data into rrdtool round-robin databases.

Likewise, the layered architecture of nafalize eases the addition of new types of flow input to the tool. Additional flow input drivers will continue to be added to the distribution on an ''on-demand'' basis.

Furthermore, as NAF's output format is based upon IPFIX, it will be reasonably simple to add support for using nafalize as an IPFIX Exporting Process; that is, to allow it to send output over the IPFIX Protocol. When combined with existing Collecting Process support, nafalize will be deployable as a ''drop-in'' aggregating IPFIX proxy.

Present experience with nafalize suggests that its performance is bound by I/O (how fast records can be read from disk or the network) and the size of the active flow table. While the performance is obviously dependent on both the data and the aggregation expressions used, nafalize run on a stock Dell 1850 tends to process between 100 k and 250 k records per second with between 5 k and 10 k concurrent flows in the second-stage NAFMultiBin. We plan on performing detailed profiling and optimization in order to increase this throughput.

One performance enhancement suggested by nafalize's three-stage design would be to split the stages into separate threads. This may increase throughput on multicore hardware, but more importantly, it will isolate output delay from input processing; especially important in the aggregating IPFIX proxy case above.

The NAFBinTable is presently limited to the size of available memory. nafalize may be extensible to work with extremely large datasets by replacing the underlying bintable implementation with one that can utilize on-disk storage when necessary, sacrificing performance for flexibility as needed.

Author Biographies

Brian Trammell is the Engineering Technical Lead at CERT Network Situational Awareness Group. He designs, builds and maintains software systems for the collection and analysis of security-relevant measurement data for large-scale networks. He is also an active contributor to internet-measurement related working groups in the Internet Engineering Task Force. He received his bachelor's degree in computer science in 2000 from the Georgia Institute of Technology, where he also worked as the UNIX systems administrator for the School of Civil Engineering for three years. He can be reached at bht@cert.org .

Carrie Gates is a Research Staff Member with CA Labs where she performs research in enterprise security. She received her Ph.D. in May, 2006, from Dalhousie University. While completing her dissertation, she spent three years working with CERT Network Situational Awareness at Carnegie Mellon University doing security research for large-scale networks. Previous to her doctoral studies, Carrie was a systems administrator for six years. She can be reached at carrie.gates@ca.com .

Bibliography

[1] Claise, B., *IPFIX Protocol Specification*, (work in progress), June, 2006, Internet-Draft draft-ietf-ipfix-proto-22 .

[2] Plonka, Dave, ''FlowScan: A network traffic flow reporting and visualization tool,'' *Proceedings of the 14th Large Installation Systems Administration Conference (LISA 2000),* New Orleans, Louisiana, USENIX Organization, pp. 305-317, December, 2000.

[3] Kohler, Eddie, *ipsumdump and ipaggcreate*, 2006, http://www.cs.ucla.edu/ kohler/ipsumdump/ , (Last Visited: 11 May 2006).

[4] Fullmer, M., and S. Romig, ''The OSU flow-tools package and Cisco Netflow logs,'' *Proceedings of the 14th Systems Administration Conference (LISA 2000)*, New Orleans, Louisiana, USENIX, pp. 291-303, December, 2000.

[5] Gates, C., M. Collins, M. Duggan, A. Kompanek, and M. Thomas, ''More NetFlow tools: For performance and security,'' *Proceedings of the 18th Large Installation Systems Administration Conference (LISA 2004)*, Atlanta, Georgia, USENIX, pp. 121-132, November, 2004.

[6] Lazarevic, A., L. Ertoz, V. Kumar, A. Ozgur, and J. Srivastava, ''A comparative study of anomaly detection schemes in network intrusion detection,'' *Proceedings of SIAM International Conference on Data Mining*, San Francisco, California, Society for Industrial and Applied Mathematics, May, 2003.

[7] LBNL Network Research Group, *TCPDUMP public repository*, 2005, http://www.tcpdump.org , (Last Visited: 9 May 2006).

[8] Luo, K., Y. Li, A. Slagell, and W. Yurcik, ''CANINE: A NetFlows converter/anonymizer tool for format interoperability and secure sharing,'' *FloCon 2005: Proceedings*, Pittsburgh, Pennsylvania, CERT Network Situational Awareness Group, September, 2005, http://www.cert.org/flocon/2005/proceedings.html .

[9] Navarro, J.-P., B. Nickless, and L. Winkler, ''Combining Cisco NetFlow exports with relational database technology for usage statistics, intrusion detection and network forensics,'' *Proceedings of the 14th Large Installation Systems Administration Conference (LISA 2000)*, New Orleans, Louisiana, USENIX, pp. 285-290, December, 2000.

[10] QoSient, LLC, *argus: network audit record generation and utilization system,* 2004, http://www.qosient.com/argus/ , (Last Visited: 9 May 2006).

[11] Quittek, J., S. Bryant, B. Claise, and J. Meyer, *Information Model for IP Flow Information Export*, June, 2006, Internet-Draft draft-ietf-ipfix-info-12 (work in progress).

[12] R Project, *The R Project for Statistical Computing*, 2006, http://www.r-project.org , (Last Visited: 12 May 2006).

[13] Oetiker, Tobias, *RRDtool*, 2006, http://oss.oetiker.ch/rrdtool/ , (Last Visited: 11 May 2006).

[14] Trammell, B., and E. Boschi, *Bidirectional Flow Export using IPFIX*, August, 2006, Internet-Draft draft-ietf-ipfix-biflow-00 (work in progress).