Efficient Network-wide Flow Record Generation

Joel Sommers*, Rhys Bowden[†], Brian Eriksson[‡], Paul Barford[‡], Matthew Roughan[†], and Nick Duffield[§]

*Colgate University (jsommers@colgate.edu), [†]University of Adelaide, [‡]University of Wisconsin, [§]AT&T Labs–Research

***Abstract*—Experiments on diverse topics such as network measurement, management and security are routinely conducted using empirical flow export traces. However, the availability of empirical flow traces from operational networks is limited and frequently comes with significant restrictions. Furthermore, empirical traces typically lack critical meta-data (*e.g.*, labeled anomalies) which reduce their utility in certain contexts. In this paper, we describe *fs*: a first-of-its-kind tool for automatically generating representative flow export records as well as basic SNMP-like router interface counts. *fs* generates measurements for a target network topology with specified traffic characteristics. The resulting records for each router in the topology have byte, packet and flow characteristics that are representative of what would be seen in a live network. *fs* also includes the ability to inject different types of anomalous events that have precisely defined characteristics, thereby enabling evaluation of proposed attack and anomaly detection methods. We validate *fs* by comparing it with the ns-2 simulator, which targets accurate recreation of packet-level dynamics in small network topologies. We show that data generated by *fs* are virtually identical to what are generated by ns-2, except over small time scales (below 1 second). We also show that *fs* is highly efficient, thus enabling test sets to be created for large topologies. Finally, we demonstrate the utility of *fs* through an assessment of anomaly detection algorithms, highlighting the need for flexible, scalable generation of network-wide measurement data with known ground truth.**

I. Introduction

Careful and comprehensive evaluation is a critical step in the process of developing new network algorithms, protocols and systems. Standard evaluation objectives include testing over a range of realistic operating conditions in a controlled and repeatable fashion. To that end, idealized analytic models are usually supplemented by more realistic tests. The most realistic of these use live traffic traces gathered from operational networks. While trace-based evaluation offers the benefit of realism, the traces themselves are inherently inflexible, are usually limited in their availability (*e.g.,* to members of a particular organization) and often lack meta-data (*e.g.,* labels of events like routing changes or anomalies). Also lacking is the ability to control such data to answer "what if" questions.

As a result, research groups often turn to laboratory-based testbeds or simulation to supplement data gathered from real networks. However, existing methodologies do not satisfy current requirements. Most simulations (and all testbeds) work at the packet level. However, the traffic datasets that currently appear to represent the best tradeoff between cost of collection and level of detail are *flow export traces*. Empirical flow export traces have been widely used in studies on traffic-matrix estimation and anomaly detection, amongst other topics. In principle it is trivial for a packet level simulation to generate flow records, but simulating packets when all you really require is flows is wasteful and limits scalability.

Our work is motivated by the need for representative flow export traces for the evaluation of new network algorithms and systems. Specifically, we desire flow export traces to be (1) generated in a scalable fashion from arbitrary network configurations, (2) representative in terms of packet/byte/flow and IP address range characteristics, and (3) inclusive of comprehensive meta-data on "anomalous events" that exist within the data. We know of no prior existing sources of flow export data that meet these goals.

Tools for *traffic and workload generation* [17] come closest to meeting the above goals. These approaches generally create realistic data sets or live traffic streams that can be used in simulations or testbed-based experiments. Standard examples are packet traffic generators (*e.g.,* [27], [30]) that are used to assess the performance and capacity of network systems such as routers and firewalls. The key challenges in the development of a workload generator is to identify a parsimonious foundational model that enables the production of traces or traffic streams with specific features. These features can be simple, like fixed inter-packet spacing, or more complex, like self-similarity [22], depending on the target applications. There are many such models now, though it would be fair to say that no one model has won universal acceptance. Thus, it is important to be able to flexibly implement multiple models for various tasks. The problem is that a workload generator alone is not enough. A critical aspect of a traffic trace is the surrounding network configuration. The challenge is to implement the model in a system that is scalable and robust so that it can be used over a range of configurations and deployment scenarios.

In this paper we describe a new flow record generation tool called *fs* that is designed to meet the above challenges and requirements. *fs* is based on a modeling framework that synthesizes file transfers into origin-destination (OD) flows for a target network. *fs* also includes a flexible anomaly generation capability that enables outages, volume anomalies and scans to be injected into flow traces. While *fs* is technically a simulator, it differs from prior simulation tools that are designed to enable simulation for the sake of a controlled simulation setting. *fs*, in contrast, is designed to generate flow export records and SNMP-like byte, packet, and flow counts; it simply uses simulation techniques to do so. Further, *fs* is designed to efficiently use system resources, thereby enabling generation of flow records for all nodes in large network topologies.

We validate the data generated by *fs* by comparing it with data produced by the well-known ns-2 simulator [24] and with data produced in a controlled laboratory environment using the Harpoon traffic generator [27]. Our results show that byte, packet, and flow volumes produced by *fs* are statistically indistinguishable from those produced by ns-2 or Harpoon over time scales of 1 second and longer. We also show that measurement data can be generated using *fs* significantly faster and with much lower memory requirements than in ns-2.

As one illustration of the value of *fs* we include an example of its use in assessing anomaly detection algorithms. We conduct 1000 independent simulations with *fs*, and show, using statistically-derived confidence intervals, that two simple anomaly detection techniques are indistinguishable unless one uses a very large number of tests (nearly 800). This case study underscores the need for fast, controlled trace generation. In the area of anomaly detection, where techniques have evolved toward whole network tests, trace generation must be highly scalable but still provide the level of detailed information needed for the current crop of anomaly detection algorithms, *i.e.*, the type of flow records produced by *fs*.

II. Related Work

There are many possible definitions of a *network traffic flow*. A widely embraced definition is described by Claffy *et al.* in [11] in terms of traffic characteristics observed at internal points in a network. Monitoring traffic flows is of significant importance to Internet Service Providers who use this information for both accounting and network management. To that end, flow measurement has been the subject of standardization efforts (*e.g.*, [6], [12]) and is widely implemented in network devices (*e.g.*, Cisco's Netflow [28]).

Over the past decade, there have been many studies on the measurement, analysis and characterization of network traffic flows (*e.g.*, [15], [21], [32]). These studies have implications for effective network management and capacity planning, and provide a foundation for our work. A related topic is *traffic matrix estimation*, which seeks to identify traffic volumes for origin-destination flows through a network based on partial information (*e.g.,* [25], [29], [34]). A motivation for our study is to develop a tool that could be used to systematically assess and evaluate traffic matrix estimation techniques.

Another important motivation for our work is the recognition that certain research areas are under-served in terms of the availability of tools and data for comprehensive analysis. Ringberg *et al.* highlight some of the difficulties in assessing anomaly detection algorithms, and make the case for improved simulation capability [26]. That paper highlights the need for data sets with *ground truth*, *e.g.*, flow data that includes labels that identify *all* anomalous events. *fs* is specifically designed to enable generation of flow data with labeled anomalies that could be used to assess either single node (*e.g.,* [2]) or network-wide (*e.g.,* [21]) detection algorithms. Related to our effort is that of Brauckhoff *et al.* [4] in which they describe a tool called FLAME designed to augment an existing flow export record trace with controlled anomalies. *fs* is more general and comprehensive than FLAME in that it is designed to generate *new* flow export traces with known characteristics.

Prior efforts in the area of workload generation also provide an important perspective for our study. Network workload generators can generally be divided into two categories, depending on whether they are based on replaying empirical traces (*e.g.,* [10]) or based on distributional models of key characteristics of a particular workload (*e.g.,* [1], [27], [30]). In each case, the general aim of workload generators is to be able to reproduce a range of characteristics. *fs* is based on a set of distributional models for traffic behavior using the model for single node behavior specified in Harpoon [27]. However *fs* differs from Harpoon since it is focused on flow record generation and adds a broader framework to enable consistent and scalable network-wide flow record generation.

Finally, our work builds on the various network simulators that have been widely used in past studies [9], [13], [24]. *fs* differs from these tools in its focus on flow record generation for potentially large networks.

III. Design and Implementation

In this section we describe the design, configuration, and implementation of *fs*.

A. Design Goals

fs is designed with three key considerations in mind. First is the goal to generate representative network measurements similar to those that can be collected from operational routers today. In particular, *fs* generates flow export records (*e.g.*, Cisco Netflow records) and SNMP-like counters (*e.g.*, packet and byte counters from router interfaces).

The second goal is to ensure sufficient realism in the measurements that *fs* generates. *fs* is designed to generate measurements from *benign* flows as well as particular types of anomalous flows. For benign flows, *fs* builds on the Harpoon model for traffic generation [27]. Similar to Harpoon, *fs* creates flows between a given source and destination that have particular distributional properties. Namely, flows are initiated between a source and destination according to one distribution, and flow sizes are drawn from another distribution.

The third goal of *fs* is to scale to large network configurations, not only to generate measurements quickly, but to use modest memory resources while doing so. Because the kinds of measurements that *fs* can generate do not contain fine-grained information (*e.g.*, packet-level timings), we ignore many packet-level details. This design decision results in major computational and memory savings while generating realistic data over time scales of 1 second and longer, as shown below.

B. Configuration

Another design goal for *fs* is to provide a familiar and easy-to-use method of configuration. To that end, *fs* uses a declarative syntax based on Graphviz DOT files [3]. We now describe through examples how *fs* is configured.

1) Node and link configuration: Listing 1 shows an example that will create a simple three-node network. In the DOT language, there are three basic entities: graphs, nodes, and links. In this example, there is one graph (`threenodenetwork`), three nodes (`a`, `b`, and `c`), and three links (`a--b`, `b--c`, and `a--c`). Nodes are declared by simply providing an identifier along with an optional set of attributes enclosed in square braces. Links are declared by stating two node identifiers separated by two hyphens (`--`). An optional set of square-brace-enclosed attributes can also be specified for links. A set of attributes in the DOT language essentially consists of a series of key/value pairs, with the key and value separated by the = symbol. Below, we describe various valid attributes that *fs* supports for nodes and links. Note that since DOT is a declarative language, the specific ordering of nodes and edges within the file does not matter.

a) Node attributes: In *fs*, a DOT node and a router are synonymous. Routers may be configured with a set of IP destination prefixes that are reachable through the router; observe that in Listing 1, each node has an `ipdests` attribute for this purpose. This attribute can consist of multiple white-space delimited address prefixes (either IPv4 or IPv6). These prefixes may be considered, in a sense, as external sources or destinations for the network being modeled. They are primarily used in the traffic generation process to determine the egress node of the network for a given flow.

Additional valid attributes for *fs* nodes relate entirely to how network traffic is configured. We describe traffic generation separately, below.

b) Link attributes: A link in *fs* can be considered either a logical or physical link between two routers. Notice in Listing 1 that for each link, `weight`, `capacity`, and `delay` attributes are configured. Capacity is specified in bits per second, and delay is specified in seconds. Presently in *fs*, the configured weights are fed into Dijkstra's algorithm for computing shortest-path routes across the network (if there is more than one egress node for a given destination address the closest node is used, mimicking hot potato routing).

There are additional attributes that can be configured to induce link (un)reliability. A `reliability` attribute can be used to specify how and when a link can be disabled during a simulation. One way that the reliability attributes can be specified is to give a time after which a link should fail, as well as how long the link should remain down. Listing 1 shows that the link between routers `a` and `b` should go down 30 seconds into the simulation and remain down for 10 seconds, after which it will be reenabled. Another way that the reliability attributes of a link may be specified is to supply a time to failure (TTF) and a time to recovery (TTR). The link between routers `b` and `c` in Listing 1 shows that the TTF is exponentially distributed with a mean of 600 seconds between outages. Once the link is down, the time to recover is also exponentially distributed with a mean of 5 seconds. In *fs*, when a link goes down, routes are instantaneously recomputed using Dijkstra's algorithm. In future work, we intend to investigate more flexible and realistic routing behavior.

Listing 1. fs configuration example.

```
graph threenodenetwork {
  // three node declarations, each with ipdest attributes
  a [ ipdests="10.1.0.0/16 10.128.0.0/9" traffic="m1"

      // build up and withdrawal of source s1: 10 srcs
      // for 60 sec, followed by 20 sources for 60 sec ...
      m1="modulator start=0.0 generator=s1
          profile=((60,),(10,20,30,20,10))"

      // a basic Harpoon traffic setup
      s1="harpoon ipsrc=10.1.0.0/16 ipdst=10.3.1.0/24
          flowsize=pareto(10000,1.2)
          flowstart=exponential(100.0)
          sport=randomchoice(22,80,443)
          dport=randomunifint(1025,65535) mss=1460
          lossrate=randomchoice(0.001) tcpmodel=msmo97" ];

  b [ ipdests="10.2.0.0/16" ];
  c [ ipdests="10.3.0.0/16 10.0.0.0/8" ];

  // three link declarations
  a -- b [ weight=5, capacity=1000000000, delay=0.020,
           reliability="failureafter=30 downfor=10" ];
  b -- c [ weight=13, capacity=1000000000, delay=0.010,
           reliability="mttf=exponential(1.0/600.0)
           mttr=exponential(1.0/5.0)" ];
  a -- c [ weight=30 capacity=1000000000 delay=0.123 ];
}
```

2) Network traffic: For traffic, there are three node-level items that must be configured: a traffic definition, a traffic modulator, and a `traffic` attribute. The `traffic` attribute refers to one or more traffic modulators to be activated and each modulator describes how a number of traffic sources with identical characteristics should be varied over time. In Listing 1, the `traffic` attribute for node `a` refers to the identifer `m1`. `m1` in turn refers to a traffic modulator that manages a traffic generator source, identified by `s1`.

Traffic modulators in *fs* specify when a traffic source (or multiple, identical sources) should start, and how a number of these traffic sources should evolve over time. In Listing 1, `m1` specifies that traffic generator `s1` will start at time 0, and will have a modulation `profile` such that 10 sources will be active for 60 seconds, then 20 sources will be active for 60 seconds, then 30 for 60 seconds, and so forth. The syntax for specifying a traffic `profile` consists of two ordered lists of numbers, enclosed in parentheses. (Jumping ahead to some implementation details, these values must simply conform to valid syntax for Python tuples.) The first list indicates a series of 1 or more time durations (in seconds), and the second list indicates the number of instances of a particular traffic source that should be active. The time durations and number of sources are matched in a circular fashion; in the example, the time duration 60 is repeated for each of the five values for number of sources.

The final piece is the description of how each traffic generator instance should behave. In Listing 1, `s1` is referred to by the modulator `m1` and describes the characteristics of a Harpoon-like traffic generator [27]. The generator `s1` has IP source and destination address prefixes from which addresses are chosen for new flows, a distribution of flow sizes, and another distribution that determines when individual flows start. While a Harpoon generator is active, a new flow is started after a time duration chosen from the `flowstart` distribution; that flow will have random IP source and destination addresses chosen from the source and destination prefixes, and a random flow size chosen from the `flowsize` distribution. The next flow will start after another random value chosen from the `flowstart` distribution, and so forth. The source and destination prefixes can be constructed in such a way as to recreate a particular distribution, similar to the way that Harpoon can be configured.

For one instance of a Harpoon flow, source and destination ports are also chosen from the configured settings, *e.g.*, in Listing 1, a source port is a random selection from a list of three choices, and a destination port is randomly chosen from an integral range. For this example, each flow will have a fixed TCP maximum segment size (MSS) (though the MSS may also be randomly chosen). We note that the settings shown are simply meant to be illustrative of how a Harpoon source within *fs* can be configured.

One of the most important differences between Harpoon as implemented in *fs* with the implementation in [27] is that the *fs* version does not use a real implementation of TCP as a transport. Moreover, *fs* does not even attempt to emulate any detailed packet-level interactions of a flow. Instead, we employ existing TCP throughput models to estimate the throughput (and consequent time duration) of a given flow. Two remaining configuration parameters for the generator `s1` in Listing 1 are used for this purpose: `tcpmodel` and `lossrate`. For each flow, a random loss rate is chosen from the configured parameter (as with other settings, this could just be a single value, if desired; there are no restrictions placed on how this is configured). This value, along with a computed round-trip time based on the link configurations are used as input to a TCP throughput model to estimate the average rate of the flow. The RTT computed for input to the model only considers propagation delay; transmission and queuing delays are ignored. For the TCP throughput model, *fs* incorporates the simple Mathis *et al.* model from [23] and the more complex Cardwell *et al.* model from [8]. We used these two models in order to investigate the sensitivity of *fs* to the TCP model used.

Once the average flow throughput and duration are computed, an individual flow's progress is simulated. At successive time intervals, portions a flow's volume are emitted along a path from the source node to destination node. These portions are referred to as *flowlets* in *fs*. Each flowlet represents the volume of the flow (bytes and packets) that would be transmitted over a given time interval (*e.g.*, over a 1 second interval). For small flows, the entire flow may only consist of a single flowlet and be completed within one interval. Longer flows may consist of multiple flowlets, spanning multiple intervals. Each flowlet has the appropriate TCP flags set (*e.g.*, if a TCP flow is broken into multiple flowlets, only the first flowlet has the SYN flag set); when all flowlets comprising a flow have been emitted, transmission of the flow is complete. Note that similar to a packet being the main entity to be scheduled within ns-2, a flowlet is the main entity to be scheduled and moved through the network in *fs*.

Listing 2. More complex traffic generation example.

```
graph morecomplextrafficexample {                                 1
  a [ ipdests="10.1.0.0/16 10.128.0.0/9"                          2
      traffic="m1 m2 m3 m4 m5"                                    3
                                                                  4
      // very short TCP flows.                                    5
      m1="modulator start=0                                       6
          generator=s1 profile=((120,),(1,))"                     7
      // flow consists of one flowlet; flows emitted              8
      // at exponential intervals, mean of 1 sec                  9
      s1="rawflow ipsrc=10.1.1.0/24 ipdst=10.3.2.0/24            10
          sport=80 dport=randomunifint(1024,65535)               11
          ipproto=tcp flowlets=1 bytes=normal(3000,500)          12
          pktsize=normal(1000,200) tcpflags=SYN|FIN|ACK          13
          interval=exponential(1.0)"                             14
                                                                 15
      // UDP variable bitrate flow.                              16
      m2="modulator start=0 generator=s2                         17
          profile=((120,),(1,))"                                 18
      // flow consists of 100 flowlets, one emitted every        19
      // sec.  variable bitrate comes from a random num          20
      // of packets and bytes for each emitted flow.             21
      s2="rawflow ipsrc=10.1.1.5/32 ipdst=10.3.2.5/32            22
          flowlets=100 ipproto=udp dport=4444                    23
          sport=randomunifint(1024,65535) pkts=normal(10,1) 24
           bytes=normal(1000,100) interval=1.0"                  25
                                                                 26
      // SYN flood.  at t=10 sec, ramp up as a step              27
      // function from 1 source up to 100, 10 new sources        28
      // at a time.  sustain 100 sources for 30 sec, then        29
      // then withdraw down to 0, in similar way to start..30
      m4="modulator start=10 generator=syns                      31
          emerge=((1,),frange(0,100,10))                         32
          sustain=((30,),(100,))                                 33
          withdraw=((1,),frange(100,0,-10))"                     34
      syns="rawflow ipsrc=10.1.0.0/16 ipdst=10.4.5.0/26          35
            dport=80 sport=randomunifint(1,65535)                36
            ipproto=tcp pkts=1 bytes=40 tcpflags=SYN             37
            flowlets=1 interval=exponential(1/1.0)"              38
                                                                 39
      // Randomly "subtracts" flow records along a path.         40
      m5="modulator start=30 generator=sub1                      41
          profile=((10,),(1,))"                                  42
      sub1="subtractive dstnode=b                                43
            ipsrcfilt=10.1.0.0/16 ipdstfilt=10.3.0.0/16          44
            ipprotofilt=tcp action=removeuniform(0.1)" ];        45
                                                                 46
  b  [ ipdests="10.3.0.0/16 10.4.0.0/16" ];                      47
  a -- b [ weight=10, capacity=1000000000, delay=0.043 ];        48
}                                                                49
```

a) More complex traffic configurations and anomalies: In addition to the Harpoon generator, there is another "raw" flow generator that can work as a one-flowlet generator (the entire flow is represented as one flowlet) or as a simple constant or variable bit-rate flow (*e.g.*, to mimic a UDP-based media stream). Examples are shown in Listing 2 (see `rawflow` generator configurations starting on lines 10, 22, and 35).

For some types of traffic, *e.g.*, SYN floods, it may be useful to separately describe how the traffic begins, how it behaves once active, and how it ends. In other words, the traffic *emerges*, is *sustained* for some time, then *withdraws*. These behaviors may be specified in *fs*. An example is shown on lines 31–34 of Listing 2. The values for each of the settings `emerge`, `sustain`, and `withdraw` are the same as for the Harpoon example, above. (The `frange` expression on line 50 is a shortcut for specifying `(0,10,20,30,...,100)`. Note also that `sustain` is synonymous with `profile`.)

This modulation capability enables the creation of traffic whose onset and withdrawal have specific, controlled behaviors. In concert with the raw flow generation capability, SYN floods, port scans, and a variety of other interesting traffic flows are easily constructed in a controlled fashion.

Another type of anomaly that has direct support in *fs* is a *subtractive* anomaly. A `subtractive` generator removes some fraction of flow records stored at routers and can be used to mimic problems with flow collection in a live network, *e.g.*, dropped or corrupted export packets. Such measurement problems can have a significant effect on different algorithms. Settings can be specified to only remove flows whose source or destination addresses are within specified prefixes or that match other criteria. For example, starting on line 43 of Listing 2, the subtractor will probabilistically remove 90% of flows that match the given criteria.

Finally, another sort of anomaly can arise from links going in and out of service. When this happens, routes are automatically recomputed within *fs*, and any flowlets in transit are rerouted. However, the re-routing is done instantaneously in *fs*, so at present there is no chance for looping or other types of related anomalies. Incorporating a more realistic routing model is the subject for a future enhancement of *fs*.

b) Measurement collection and export in fs*:* Flow records are collected at each router by aggregating flowlets as they traverse network paths. Routers in *fs* export flows when a FIN or RST is observed in a flowlet, after a fixed duration, or when the number of stored flows exceeds a certain threshold; each of these behaviors are configurable. *fs* can currently export records in a simple text format, or in Cisco Netflow version 5 binary format [28]. There are plans to support (at least portions of) the IPFIX standard [12].

Sampling can also be enabled in routers in *fs*. Currently implemented are 1-in-n packet sampling (effectively, packets within flowlets are sampled), and uniform probabilistic sampling. *fs* could easily be extended to support additional types of sampling, *e.g.*, trajectory sampling [14].

A further measurement capability in *fs* is that routers can be configured to collect and export byte, packet and flow counters at each interface, similar to basic SNMP counters. These counters can be exported at configurable intervals.

Finally, we note that the measurement capabilities described above are presently configured on the command-line to *fs*, outside the context of the DOT configuration file.

C. Implementation

At its core, *fs* is a discrete-event simulator and is implemented in the Python programming language. The current implementation consists of about 2,000 lines of Python. *fs* is openly available to the research community (see http://cs.colgate.edu/faculty/jsommers/).

As noted above, a key point of differentiation between *fs* and other discrete-event network simulators like ns-2 is that instead of simulation events revolving around packet-level activity in a network, *fs* is concerned with *flowlets*. The key issue that determines performance of a discrete-event simulator is the rate at which events are fired in the simulator. Two features of *fs* have an important effect on the number of events scheduled, and thus the event rate. First, the use of flowlets as an abstraction greatly reduces the number of events required to simulate a single flow since individual packets comprising a flowlet are not simulated, only the aggregate. Second, specifically with respect to the Harpoon generator, the configured interval at which flowlets that comprise a single flow are emitted also has an important impact. For example, with longer intervals, fewer flowlets need to be emitted for a given flow (and thus fewer events).

IV. Validation Experiments

In this section we examine the behavior and performance of *fs*, focusing on comparing results from running *fs* and the widely used ns-2 simulator (version 2.34) with equivalent configurations. We also evaluate the impact of different *fs* configuration choices on the resulting measurements. Finally, we have performed experiments in a laboratory emulation setting using Harpoon. Due to space limitations, we do not show results from those experiments, but they are qualitatively similar to the results comparing ns-2 and *fs*.

A. Experiment Setup

For the majority of experiments described in this section, we focus on measurements collected using a simple network setting. In particular, the topology we use is the same as shown in the example *fs* configuration in Listing 1. In this setup, there are three nodes, each with 1 Gb/s links connecting them. Note that because of the configured link weights, the shortest path between `a` and `c` goes through node `b` (*i.e.*, the direct link between `a` and `c` is unused). (Note that we do not configure reliability parameters for links in these experiments.)

For network traffic, we configured the Harpoon generator in *fs* to generate 100 flows per second, started at exponentially distributed intervals. We used two different flow size distributions: exponentially-distributed and Pareto-distributed. The latter distribution was configured to result in a heavy-tailed distribution of flow sizes. ns-2 was configured in an equivalent manner using the PackMime web-like traffic generator [7].

B. Traffic Volume Comparisons

We first compare traffic volumes produced by ns-2 and *fs*. In this setup, we configured *fs* to use the Cardwell *et al.* TCP throughput model [8] (referred to below as CSA00). We used the SNMP-like data produced by *fs*. Figure 1 shows cumulative distribution functions (CDFs) of byte and packet volumes produced over 1 second bins for the scenario with exponentially-distributed flow sizes, and Figure 2 shows CDFs of byte volumes for the scenario with Pareto-distributed flow sizes (CDFs for packet volumes are not shown but are qualitatively similar). Note that the x-axis is log scale. The plots show a good match between ns-2 and *fs*. 95% confidence intervals plotted for different quantiles of the ns-2 curve include nearly all *fs* values at those quantiles. Due to space limitations, we do not show results comparing the number of flows generated per second in ns-2 and *fs*, but they are indistinguishable.

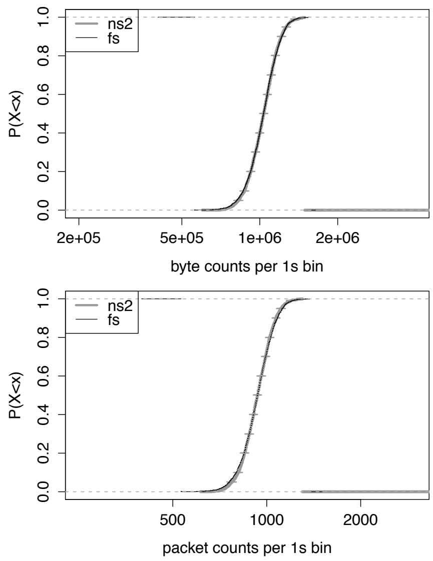

Fig. 1. Results with exponentially-distributed flow sizes for baseline setup.

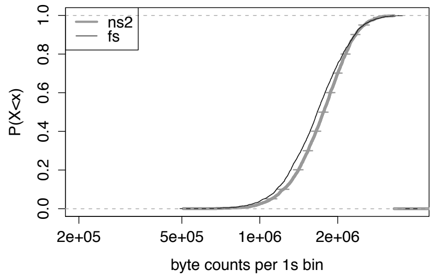

Fig. 2. Results with Pareto-distributed flow sizes for baseline setup.

For the previous plots, we used the CSA00 TCP throughput model, with a configured loss rate of 0.001. In Figure 3, we compare flow durations in ns-2 with three different TCP throughput model settings in *fs*: the Mathis *et al.* [23] model (MSMO97) with a configured loss rate of 0.001, the CSA00 model with a loss rate of 0.001, and the CSA00 model with a loss rate chosen between 0.001 and 0.01 with uniform random probability. (While there is no packet loss in the ns-2 simulation, we found through a series of experiments that a non-zero loss rate was needed with the CSA00 model in order to achieve reasonable accuracy in matching the flow durations measured in ns-2. Note also that using the MSMO97 model *requires* a non-zero loss rate since $\sqrt{p}$, where p is the loss rate, is a divisor in the throughput estimation formula. Both of these issues are due to limitations with the CSA00 and MSMS97 TCP throughput models. Exploring modified or new TCP models for *fs* is a subject for future work.) For this plot, the flow durations are taken directly from the flow export records generated by *fs*. We see that for the MSMO97 model, there is a poor match for flows of short duration (consider that the MSMO97 model only considers flows in congestion avoidance). For the CSA00 configurations, the short flow durations match well for each setup, but the configuration with a fixed 0.001 loss rate tends to underestimate the tail of the ns-2 durations. Surprisingly, the *fs*+CSA00 setup in which a randomly chosen loss rate is used provides a good match to ns-2 flow durations. Note that even if the flow durations do not match well the overall flow volumes still closely match.

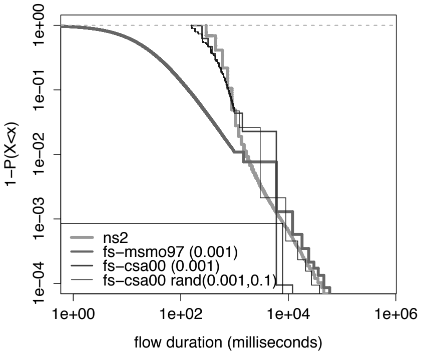

Fig. 3. Complementary cumulative distribution function (log-log scale) for flow durations in ns-2 and *fs* for three different TCP throughput model settings in *fs* and with Pareto-distributed flow sizes.

C. Volume aggregation effects

The baseline results compared packet and byte volumes over fixed 1 second bins. Since *fs* does not attempt to model or simulate low-level packet interactions, an important question to consider is how packet and byte volumes match over shorter bins, *e.g.*, 100 milliseconds. We compared byte volumes for bin sizes of 10 milliseconds, 100 milliseconds, and 1 second. The results of these experiments showed that the match in volumes becomes poorer as the the interval becomes shorter (plots not shown due to space limitations).

Figure 4 plots variance estimates versus time scale for a range of time aggregations of byte counts (1 millisecond through 100 seconds) for both ns-2 and *fs* and each flow size distribution. 95% confidence intervals are drawn for each curve. Observe that at time scales below 1 second, the variance estimates of ns-2 and *fs* differ substantially, but at 1 second and greater aggregations they match very well. These results suggest that measurements produced by *fs* at aggregations of 1 second and greater are statistically indistinguishable from ns-2 (and also measurements produced in a laboratory setting; results not shown due to space limitations). The plots referred to above also highlight a boundary of *fs*, suggesting that one should not use traffic volumes produced by *fs* for aggregation levels below 1 second. Indeed, *fs* is not designed to accurately capture sub-second traffic dynamics.

D. Network congestion effects

In this last set of traffic-oriented experiments, we compare ns-2 and *fs* in a congested network scenario (*i.e.*, there is packet loss in the ns-2 simulation). In *fs*, if the aggregate volume of flowlets crossing a given link exceeds the configured link bandwidth, flowlets are queued. This queueing is clearly a coarse approximation of what would happen in a live or simulated router that operates on packets. Thus, in the

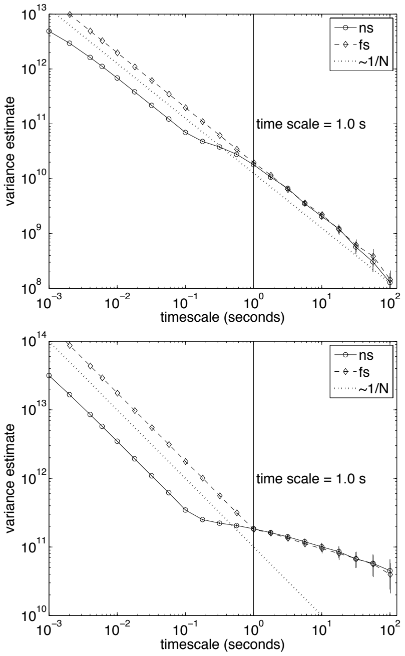

Fig. 4. Variance/time plots for exponentially-distributed flow sizes (top) and Pareto-distributed flow sizes (bottom).

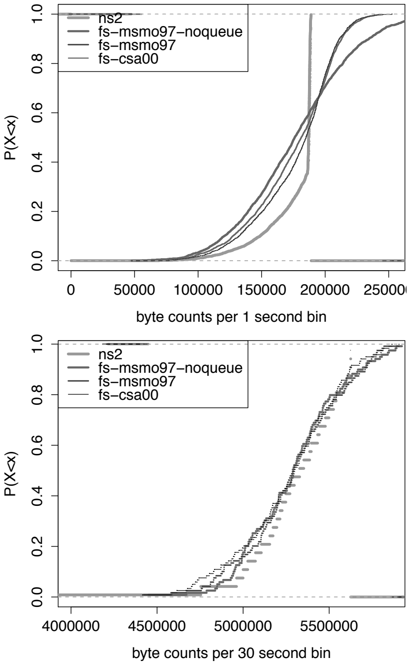

Fig. 5. Comparison of byte volumes produced by ns-2 and *fs* in a highly congested scenario, for 1 second and 30 second bins.

TABLE I
RUNTIME AND MEMORY COSTS FOR NS-2 AND *fs* USING PARETO-DISTRIBUTED FLOW SIZES AND A RANGE OF AVERAGE FLOW ARRIVAL RATES.

New flows per sec	**Runtime (seconds)**		**Peak memory consumption (kB)**	
	ns-2	*fs*	ns-2	*fs*
100	1358	149	163232	27956
200	4440	297	315800	27952
400	15894	597	623404	27956
800	59661	1185	1227900	27952

experiments below we examine another implication of *fs* being oriented around network flows rather than low-level packets.

Figure 5 compares byte volumes over two different aggregation levels (1 second and 30 seconds), computed for ns-2 and three configurations of *fs*. In these experiments, we created a configuration with low-bandwidth links (1.5 Mb/s), Pareto-distributed flow sizes, and an average flow arrival rate such that in the ns-2 simulation there was a rather high overall loss average of 10%. For *fs*, we plot results for configurations in which we (1) ignore link bandwidths and use the MSMO97 TCP throughput model, (2) respect link bandwidths, implement flowlet queues, and use the MSMO97 TCP throughput model with a configured loss rate of 10%, and (3) respect link bandwidths (with queues) and use the CSA00 TCP throughput model with a configured loss rate of 10%. We see from the plot that for the finer aggregation level, the *fs* results diverge from the ns-2 results, especially for the simplistic configuration that ignores link bandwidths. For the 30 second aggregation plot, the *fs* results are much closer to ns-2, as we should expect.

E. Scalability and performance of fs

Finally, we examine memory and runtime requirements for a set of experiments with increasing number of simulated flow arrivals per second. We use Pareto-distributed flow sizes and a range of exponentially distributed flow arrival averages per second: 100, 200, 400, and 800. These experiments were run on a quad-core Intel Xeon host with 4 GB of memory.

Table I shows the memory consumed by ns-2 and *fs* for these experiments and the runtime required to complete a 3600 second simulation. We see that for the scenario with an average flow arrival rate of 400 per second, it takes ns-2 nearly 4 1/2 hours to complete the simulation while consuming nearly 600 MB of memory. *fs* generates data from a comparable configuration in about 10 minutes and uses around 27 MB of RAM. Note that the memory and runtime requirements for ns-2 appear to scale linearly with the number of sources, while for *fs* there is no increase in memory consumption and only a modest increase in runtime.

Although we do not show detailed results here, *fs* also performs well for larger topologies and traffic configurations. We configured *fs* with a topology matching the Internet2 IP backbone, with origin-destination flows configured in a full mesh across the network. *fs* was able to generate flow records for a simulated hour with relatively low levels of traffic (*e.g.*, about 10 Mb/s between each pair of routers) in about 5 minutes. Experiments to generate flow records over an entire simulated day and with much higher traffic volumes are still quite expensive, but we are strongly encouraged by the performance of *fs* and its potential to handle significantly more complex configurations.

V. Application: Anomaly Detection

Automated detection of anomalies in computer networks has been of interest for a number of years, *e.g.*, [16], [18]–[20], [33], [34]. However, quantitative assessment of anomaly detection algorithms remains challenging. The preferred approach in the literature is to use ground-truth data, *i.e.*, a list of manually labeled "anomalies", and to compare a detector's output to this list. However, manual labeling is insufficient for many reasons [26]: it may be impossible to share the underlying traces due to privacy concerns; domain experts are flawed and may miss true anomalies; reliance on a fixed trace prevents a sensitivity analysis, *e.g.*, to determine how "large" an anomaly must be for it to be detected; and manual processes do not scale to the magnitude necessary to estimate sensitivity and specificity of anomaly detection techniques accurately.

Only simulation can provide all of the features needed to complement real data in the evaluation of anomaly detection[1]. Much of the interesting recent work on anomalies has focused on network wide measurements. Moreover, many techniques aim to exploit the full detail of the measurements available, which are often at the flow level. In this section, we expand on the issue of accuracy of performance estimates for anomaly detectors, and show that *fs* provides a suitable simulator environment for many such tests.

Anomaly detection algorithms are often assessed using two primary metrics: *false alarm* and *detection* probability. Estimation of probabilities is a textbook problem: given N trials and X successes, the estimated probability is $\hat{p} = X/N$. However, estimates have errors and we cannot make valid comparisons unless we account for these potential errors.

A common approach for quantifying errors is to use a confidence interval. Confidence Intervals (CIs) are defined with respect to *coverage probabilities*, *i.e.*, we might specify the 95th percentile CI to contain (or cover) the correct value with coverage probability 0.95. The calculation of confidence intervals for a probability estimate such as the detection probability appears in many textbooks, but much of the wisdom in this area is outdated. Standard Gaussian confidence intervals (on the assumption that the central limit theorem applies) have been shown to be invalid. More recent careful consideration has shown that even for large n, Gaussian CIs do not give stable or accurate coverage probabilities [5]. There are a number of reasons, but perhaps most telling is the fact that the variance used in the Gaussian CIs is also an estimate. If one replaces this estimate with the null estimate then a better CI estimate is obtained, usually named the Wilson interval after its first known proponent [31]. There are competing estimators of such CIs, but the Wilson interval performs almost as well or better for most cases, is simpler to calculate, and its width is proportional to $1/\sqrt{N}$ for large numbers of tests N.

Note, however, that the intervals are wide. Given $N = 100$, and a value of $p = 0.1$ the CI's width is > 0.1, and this decreases only with the square root of N. One hundred times as many trials are needed to reduce this interval width by a factor of 10. The width of these intervals is a critical factor in assessing anomaly detection algorithms. If we wish to make comparisons between two detectors, then the width of these intervals must be smaller than the difference between the detection probabilities. In order to achieve this, we may need to conduct *a large number of trials*. It would be very costly to manually classify a large enough dataset to achieve the types of results we require.

The situation is further complicated by the need for *independent* trials. If the trials are correlated in some way (say because they are datasets from the same network in a similar time period), then the width of the confidence intervals should increase yet further. *The only practical way to generate a large number of guaranteed independent trials is simulation.*

In order to illustrate this problem we use *fs* to simulate and test two simple anomaly detection algorithms. We use a similar simulation setup to the previous examples with a small and conceptually simple network with 3 nodes and 3 high capacity links across which we generate traffic using Pareto sources, but with the addition of a large but short-lived flow at a randomly chosen time point.

We test two simple anomaly detection algorithms: (1) a thresholded differencer, and (2) a thresholded high-pass filter. We evaluated both the false alarms, and detection probabilities for these two techniques, but for illustrative purposes we only show detection probabilities. Note that we make no claims that these are good approaches to anomaly detection. The point is to illustrate the need for a tool like *fs*.

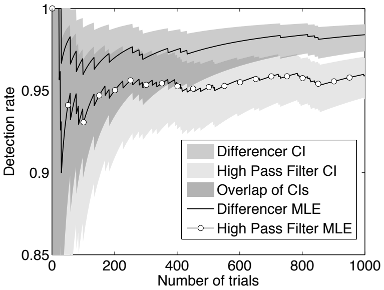

Fig. 6. A formal comparison of the accuracy of two detection probabilities as a function of the number of trials.

Figure 6 shows the estimated probability of detection for the two anomaly detection techniques with 95th percentile confidence intervals. We can see that the confidence intervals overlap up to 750 trial simulations. After that, we can see that the difference outperforms the high-pass method but nearly 800 trials were required before this distinction could be made.

It is tempting to dismiss these types of results as a statistical nicety; we eventually came up with the same result that we might have after 100 trials. Consider, however, the performance until about the 30th trial: the differencer detects every anomaly. It would be tempting to call it perfect, but we later see that it misses detections. The CIs convey the uncertainty about our belief in the "perfection" of the estimator given only

[1] Of course we do not suggest that only simulation should be used. Real labeled data is still a valued *component* of evaluation.

30 trials, even though they are all successful. Likewise, we can see that the estimators for both approaches jump around. This high variation means we cannot be certain about performance, and the CIs consistently allow for this variability.

The take-away point is that the number of trials needed to discriminate the two detectors was very large. Add in the requirement that they be independent, and it becomes clear that measurement data derived from a tool such as *fs* are needed to robustly and effectively assess anomaly detection algorithms.

VI. Summary and Conclusions

The ability to synthesize network measurement datasets with known characteristics is critical for comprehensive evaluation of anomaly detectors, traffic matrix estimators, and other types of algorithms. In this paper, we describe the *fs* tool that can generate network-wide flow export records and SNMP-like measurements that are consistent with measurement collected from live network systems. We described how *fs* can be configured and examined its fidelity and sensitivity in relationship to measurements collected from the ns-2 simulator. We then demonstrated an application in which a tool like *fs* is highly beneficial, anomaly detection.

There are a number of directions we are considering for future work with *fs*. For example, while *fs* can easily produce datasets for use in traffic matrix estimation studies, we are considering how to use traffic matrix data for the purpose of automatically parameterizing *fs* so that it can generate a new set of measurements that are statistically similar to the original TM data. This capability should enable *fs* to be used for introducing particular anomalies into an existing traffic matrix trace in realistic way, and for better understanding macroscopic network behavior and performance.

Acknowledgments

This work was supported in part by NSF grants CNS-0716460, CNS-0831427, CNS-0905186, an NSF CAREER award, and ARC grant DP110103505. Any opinions, findings, or conclusions are those of the authors and do not necessarily reflect the views of the NSF or the ARC.

References

[1] P. Barford and M. Crovella. Generating Representative Web Workloads for Network and Server Performance Evaluation. In *Proceedings of ACM SIGMETRICS*, Madison, WI, June 1998.

[2] P. Barford, J. Kline, D. Plonka, and A. Ron. A Signal Analysis of Network Traffic Anomalies. In *Proceedings of ACM Internet Measurement Workshop*, Marseilles, France,, November 2002.

[3] A. Bilgin, J. Ellson, E. Gansner, Y. Hu, Y. Koren, and S. North. Graphviz-Graph Visualization Software, 2010.

[4] D. Brauckhoff, A. Wagner, and M. May. FLAME: a flow-level anomaly modeling engine. In *Proc. of the Workshop on Cyber Security Experimentation and Test (CSET '08)*, 2008.

[5] L.D. Brown, T.T. Cai, and A. DasGupta. Interval estimation for a binomial proportion. *Statistical Science*, 16(2):101–133, 2001.

[6] N. Brownlee, C. Mills, and G. Ruth. Traffic Flow Measurement: Architecture. IETF RFC 2063, January 1997.

[7] J. Cao, W. Clevelan, Y. Gao, K. Jeffay, F. Smith, and M. Weigle. Stochastic Models for Generating Synthetic HTTP Source Traffic. In *Proceedings of IEEE INFOCOM '04*, Hong Kong, March 2004.

[8] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP Latency. In *Proceedings of IEEE INFOCOM '00*, Tel Aviv, Israel, March 2000.

[9] X. Chang. Network simulations with OPNET. In *Proceedings of the Winter Simulation Conference*, volume 1, pages 307–316, 1999.

[10] Y. Cheng, U. Holzle, N. Cardwell, S. Savage, and G. Voelker. Monkey See, Monkey Do: A Tool for TCP Tracing and Replaying. In *Proceedings of USENIX Technical Conference*, Boston, MA, June 2004.

[11] K. Claffy, H-W Braun, and G. Polyzos. A Parameterizable Methodology for Internet Traffic Flow Profiling. *IEEE Journal on Selected Areas of Communication*, 13(8), October 1995.

[12] B. Claise and Editor. Specification of the IP Flow Information Export (IPFIX) Protocol for the Exchange of IP Traffic Flow Information. IETF RFC 5101, January 2008.

[13] J. Cowie, H. Liu, J. Liu, D. Nicol, and A. Ogielski. Towards Realistic Million-node Internet Simulations. In *International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[14] N. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions on Networking (TON)*, 9(3):292, 2001.

[15] N. Duffield, C. Lund, and M. Thorup. Estimating Flow Distributions from Sampled Flow Statistics. In *Proceedings of ACM SIGCOMM '03*, Karlsruhe, Germany, August 2003.

[16] C. Estan, S. Savage, and G. Varghese. Automatically inferring patterns of resource consumption in network traffic. In *ACM SIGCOMM*, pages 137–148, Karlsruhe, Germany, 2003.

[17] S. Floyd. Traffic Generators for Internet Traffic. http://www.icir.org/models/trafficgenerators.html, 2010.

[18] L. Huang, X. Nguyen, M. Garofalakis, J. Hellerstein, M. Jordan, M.Joseph, and N.Taft. Communication-efficient online detection of network-wide anomalies. In *Proc. of IEEE INFOCOM*, 2007.

[19] R.R. Kompella, S. Singh, and G. Varghese. On scalable attack detection in the network. In *ACM Internet Measurement Conference*, pages 187–200, New York, NY, USA, 2004.

[20] A. Lakhina, M. Crovella, and C. Diot. Mining anomalies using traffic feature distributions. In *ACM SIGCOMM*, pages 217–228, Philadelphia, Pennsylvania, USA, 2005.

[21] A. Lakhina, K. Papagiannaki, M. Crovella, C. Diot, E. Kolaczyk, and N. Taft. Structural Analysis of Network Traffic Flows. In *Proceedings of ACM SIGMETRICS '04*, New York, NY, June 2004.

[22] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the Self-Similar Nature of Ethernet Traffic (extended version). *IEEE/ACM Transactions on Networking*, 2(1), February 1994.

[23] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance aAlgorithm. *ACM SIGCOMM Computer Communication Review*, 27(3):82, 1997.

[24] S. McCanne, S. Floyd, K. Fall, K. Varadhan, et al. Network Simulator ns-2, 1997.

[25] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot. Traffic matrix estimation: Existing techniques and new directions. *ACM SIGCOMM Computer Communication Review*, 32(4):174, 2002.

[26] H. Ringberg, M. Roughan, and J. Rexford. The Need for Simulation in Evaluating Anomaly Detectors. *Computer Communications Review*, 38(1), January 2008.

[27] J. Sommers and P. Barford. Self-Configuring Network Traffic Generation. In *Proceedings of ACM Internet Measurement Conference*, Taormina, Italy, October 2004.

[28] Cisco Systems. Cisco Netflow. http://www.cisco.com/go/netflow, 2010.

[29] Y. Vardi. Network Tomography: Estimating Source-Destination Traffic Intensities from Link Data. *Journal of the American Statistical Association*, 91(433), 1996.

[30] K. Vishwanath and A. Vahdat. Realistic and Responsive Network Traffic Generation. In *Proceedings of ACM SIGCOMM '06*, Pisa, Italy, September 2006.

[31] E.B. Wilson. Probable inference, the law of succession, and statistical inference. *Journal of the American Statistical Association*, 22(158):209–212, 1927.

[32] Y. Zhang, L. Breslau, V. Paxson, and S. Shenker. On the Characteristics and Origins of Internet Flow Rates. In *Proceedings of ACM SIGCOMM '02*, Pittsburgh, PA, August 2002.

[33] Y. Zhang, Z. Ge, A. Greenberg, and M. Roughan. Network anomography. In *ACM Internet Measurement Conference*, Berkeley, California, USA, October 2005.

[34] Y. Zhang, M. Roughan, W. Willinger, and L. Qui. Spatio-Temporal Compressive Sensing and Internet Traffic Matrices. In *Proceedings of ACM SIGCOMM '09*, Barcelona, Spain, August 2009.