

# Agile and Scalable Analysis of Network Events

Mike Fisk, George Varghese

**Abstract**—The state of the art in general purpose software systems for large-scale traffic measurement has not progressed much past the venerable *libpcap*. In this paper we describe a new data analysis system that provides a scalable, flexible system for composing ad-hoc analyses of high-speed, streaming data. This agility allows researchers, network security analysts, or network operators to easily compose new analysis functions. A growing tool box of filtering, measurement, and statistical tools allows new approaches to be tested with a minimum of software development. Further, a dynamic type system allows polymorphic analysis modules to operate on arbitrary forms of structured data, thus allowing easy integration of multiple data sources such as packet traces, netflow records, or security logs. In this paper we present this system and demonstrate its capabilities while performing several measurements, such as computing probability density functions, detecting port-scans, and probabilistic counting of traffic traces.

## I. INTRODUCTION

Many network measurements involve some sort of statistic, reduction, or other computation on huge streams of network data. Abstractly, this analysis could be expressed as a relational database query, yet traditional databases are not amenable to large traffic traces, much less real-time measurement of live traffic. So instead, the process of analyzing network data is usually performed by throw-away programs that input streams of raw data and parse, extract, and calculate the desired information.

The venerable *libpcap* has greatly simplified the task of acquiring network packets for measurement. But once a raw packet has been given to the programmer, he or she is left to do virtually everything else by hand. In many cases, parsing complex data structures such as packets is a more complex task than the measurement at hand. Worse, it is something that is repeated over and over again by programmers. This overhead, especially for operational analysts who are rarely systems programmers, is one of the biggest barriers to exploring new ideas.

In evolving systems and research environments, system agility is a key factor determining the ability of researchers and engineers to explore new ideas quickly. Even network operators and security analysts frequently explore new measurements. Thus it seems that measurement is an inherently dynamic field full of ad-hoc analysis that may not be repeated.

During our own analysis of packet traces and audit logs for

M. Fisk is with Los Alamos National Laboratory and the University of California San Diego. E-mail: mfisk@lanl.gov

G. Varghese is with the University of California San Diego. E-mail: varghese@cs.ucsd.edu

This work was supported by a NIST Critical Infrastructure Grant for the Sensitive Project and by the U.S. Department of Energy under contract W-7405-ENG-36 with Los Alamos National Laboratory.

ACM COPYRIGHT NOTICE. Copyright 2002 by the Association for Computing Machinery, Inc. Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Publications Dept, ACM Inc., fax +1 (212) 869-0481, or [permissions@acm.org](mailto:permissions@acm.org).

the purpose of intrusion detection research and operations, we realized the need for a general-purpose data analysis platform. While we had a specific application domain in mind, the statistics we gather, and the context of the problem, are indeed quite common.

The community needs a component-based platform for developing and using analysis functions, I/O functions, and data manipulation. In this paper we present our *System for Modular Analysis and Continuous Queries* which allows users to assemble modules into an efficient system for analyzing multiple types of streaming data. We believed that the power of this sort of component system is, much like Metcalfe's 'network-effect,' based on the square of the number of modules that can be combined.

The system we have developed is unique for four reasons. First, it is optimized for analyzing and filtering large streams of data. Second we make extensive use of polymorphic components that can perform common functions on new and unforeseen types of data without requiring any additional programming. Third, we provide a scheduling infrastructure that can both pipeline data flows and exploit data parallelism while maintaining a simple, single-threaded API for component development. Fourth and finally, we provide a user interface and set of components that allow a non-programmer analyst to quickly assemble and execute new kinds of ad-hoc analysis functions.

## II. RELATED WORK

Previous researchers have developed modular software architectures for extensible systems [1], [2], but few optimized for churning through large amounts of data. Other researchers have built extensible systems for streaming data through protocol layers [3] or routing functions [4], but have not provided an infrastructure that is particularly amenable to analyzing extremely large or broad datasets. Windmill [5] provides a modular system for monitoring network protocol events, but does not provide any significant infrastructure to aid in the analysis of those events, or non-protocol events, once they are acquired. Continuous Query systems [6], [7] from the database field share many of the concerns of acquiring and filtering continuous streams of data, but do not have extensible type systems or the ability to easily add new functions over that data.

The Click [4] modular router provides a data-flow system designed for a single data type, the packet, and for the specialized purpose of building router-like systems. The mechanics of managing data flow in our system are similar to Click, but our dynamic type system and support for polymorphic modules allow the system be much more easily applied to new types of data.

The Knit [2] system supports arbitrary interfaces between components and studies how systems can be statically assembled and then optimized. Our system lacks some of this static optimization, but the explicit data-flow graphs in our system create opportunities for the system to utilize run-time pipelining

and parallelism.

Event-based systems such as Bro [8], Glish [9], *listeners* in Java, and *callbacks* in X-Windows, allow components to generate events that are then delivered to any components that have registered an interest in those events. Event-based systems share our goal of allowing users to assemble components in unforeseen ways. But instead of routing messages (events) by their type, we route them based on the data-flow graph that was constructed (with a linear pipeline being a simple form of graph). Components in our system can have any number of output channels that they produce messages on. These channels can be used to separate different types of messages (events), but can also be used to divide messages of the same type between different consumers based on criteria such as load-sharing or connection state. Most event-based systems do not support this type of load distribution. Thus we can assemble the same flows of information as an event-based system, but provide lower-level routing that can be used in ways that most event-based systems cannot.

Special purpose programming languages have been created for the network analysis domain. The network intrusion detection sensors Bro [8] and NFR [10] both let a programmer define small program segments to perform arbitrary analysis on network packets. Our system also provides similarly high-level access to complex data types like packets. In contrast, we provide this interface in both C and Python [11] and operate on data types other than packets.

The Unix environment provides powerful abstractions and conventions for assembling ad-hoc data processing pipelines using small, single-purpose utilities such as *grep*, *awk*, *sed*, *uniq*, *sort*, *head*, *tail*, etc. However, these utilities all operate on lines of text assembled into byte streams. This adds marshaling and parsing overhead to each function and makes it difficult to operate on complex data types. A performance limitation of this set of tools is that data is passed between elements of a pipeline through system calls and system libraries that repeatedly buffer and copy data.

Similarly, *libpcap* [12] packet trace files are a commonly used stream format for storing and passing packets between utilities such as *tcpdump* [13], *tcpdump* [14], *Snort* [15], etc. However, each of these programs must marshal binary data types to and from byte streams. This prevents general purpose tools like *grep*, *sort*, and *uniq* from being used on these files. Further, it is non-trivial to extend systems like *tcpdump* to work on similar, but different data types.

### III. SYSTEM DESIGN

We have built an infrastructure for efficiently passing structured data between software components. Like traditional Unix tools, these components should be single-purpose and require no prior knowledge of each other. But unlike these tools, we should be able to preserve and utilize structure within the data.

The system is implemented as a thread-safe C library that can be embedded in other applications. We provide two command-line interfaces that use the library to accept Unix-like pipeline syntax or queries in a subset of SQL. The library is also being used in an immersive visualization environment written in C++.

As shown in Figure 1, our system is composed of two run-time environments. One is an extensible dynamic type system

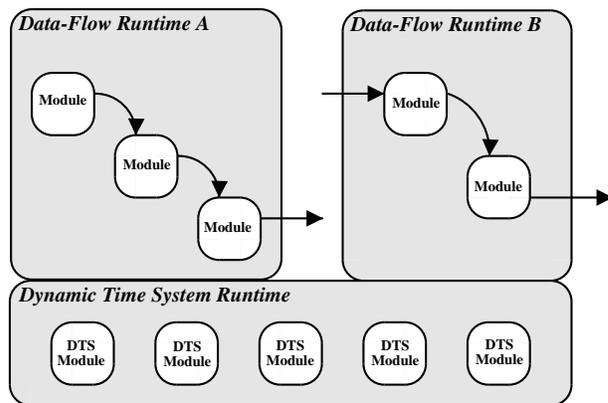


Fig. 1. Example Run-time Instantiation

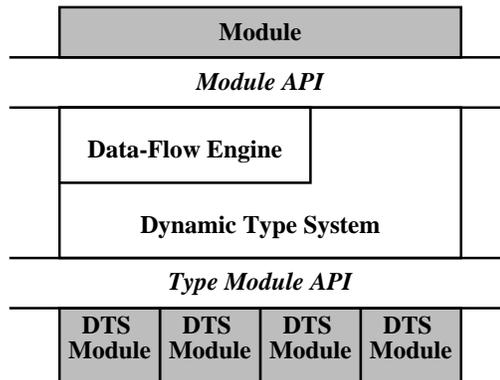


Fig. 2. Modular Programming Environment

that supports polymorphic analysis, and the other is a data-flow and scheduling environment for passing data objects. The dynamic type system can be instantiated and used separately, or as shown in the figure, it can be used to share data between multiple instantiations of the data-flow runtime. Figure 2 shows the programming environment and the two APIs for writing type modules and analysis modules. In the remainder of this section, we will describe these two runtime systems further.

#### A. Dynamic Type System

A key portion of the system is a dynamic type system implemented in C. All data that flows through the system is typed. Data records are reference-counted objects with meta-data such as type information. Types are dynamically loaded modules that provide access to an atomic value, such as an *int*, or a structure, like a *packet*, of any number of named, typed data fields. A type may provide a header file that exports the internal storage structure so that modules can access data directly and efficiently. This is particularly important for modules that interface with existing software that operates only on structures (such as *libpcap*-based programs).

We have exported this type system (including the annotations described below) to Python [11], a dynamically-typed scripting language. This allows analysis modules to be written in Python and utilize its native object syntax for accessing fields of data objects.

## A.1 Polymorphism

It is particularly important to us that general purpose analysis modules be polymorphic so that they may operate on any type of structured data that might be used in the system. This is implemented by allowing types to provide a list of named fields of the type. A polymorphic module can request any of these fields by name. This enables the creation of modules that perform general-purpose operations such as comparison, summation, averaging, and display of named fields requested by the user. However, modules that understand a data type can access that type directly without using function calls to named fields. This allows type-specific modules to have more efficient access to data.

A type can register named fields either by location within a structure, or with a callback function that must be used to access that field. This is necessary for fields with variable locations or sizes that can only be determined by further inquisition of the data record. For instance, the location of fields of a TCP header of an IP packet are determined by the variable size of the IP header. However, a polymorphic module that accesses fields by name is unaware of this difference.

## A.2 Annotation

In addition, we value the ability to annotate data types with new fields that are not part of a pre-defined type. This form of mix-in inheritance [16], [17] is important when creating and propagating meta-data such as probabilities, counts, or alerts to data. Annotations can be made either directly to a known record or addressed to sets of records.

Direct annotations are useful when adding meta-data or interpretations of data. For example, an intrusion detection system might want to annotate a packet with an alert that was caused by that packet. Or a measurement system might want to annotate a packet with the fraction of bandwidth that its source was consuming.

However, it may be more useful to address an annotation to a set of records. For instance, it might be more useful to address an intrusion alert to all other packets from the same source address. Annotation addressing is expressed as one or more field criteria. Each criterion is a field name (such as *srcip*), an operator (such as equality or existence), and a value (for comparison operators such as equality).

Criteria are evaluated lazily when a module asks for a named field of a record. If the type of that record does not have a field by that name, pending messages are checked to see if they match this record. Messages can be delivered once (an *anycast*) or as many times as they match (a *multicast*). An anycast message will be held in the system until it matches once and will then be thrown away. Multicast messages will be held in the system indefinitely. In the future we plan to add the ability to cancel multicast messages in order to prevent them from consuming too much memory.

We use the terms *anycast* and *multicast*, but it may be easier to think of these messages as being posted to a white-board that other analysis modules can look at in the future. Because messages are published to the system rather than held privately by a module, the system can optimize when these messages are

bound to new records. In particular, we can apply aggressive pre-fetching of messages when transferring a datum to a remote cluster node.

## B. Data-Flow System

The data-flow runtime handles flow-control between modules. Data-handling efficiency is important to processing large streams of data quickly. As a result, our system uses zero-copy passing of data records through callbacks that we require modules to register for production, consumption, initialization, and shutdown.

Data is provided to a module through the *consume* callback. The *consume* function indicates through its return value whether the data it was given should be filtered out or passed on to children in the flow graph. If it is to be passed, the module can also specify which output channel the data should exit on, or that the record should be broadcast to all channels. The flow graph can have different children associated with each output channel.

In addition, the *consume* call can indicate whether the module has any new records to produce. If it does, then the system must call the *produce* callback when it is ready for new data, but before passing any new data to the *consume* call.

Because a given data record may be destined for multiple modules, modules are not allowed to modify records directly. Instead they must request a modifiable copy<sup>1</sup> that they can change. The module must indicate that the original data should be filtered out and then produce the new record.

## C. Scheduling and Parallelism

Our system has a modular scheduler design and currently has three implemented schedulers. One uses a recursive, depth-first graph traversal, another uses an iterative breadth-first traversal, and the third uses a multi-threaded model.

For small per-module workloads, the single-threaded schedulers perform better. However, multi-threading is useful for pipelining modules on multi-processor systems and to accommodate modules that process at different, sometimes bursty, rates and can benefit from asynchronous scheduling buffering between modules. An input queue, in the form of a FIFO ring of record pointers, is associated with each module. Individual queue sizes can be changed to allow for different amounts of buffering and asynchrony. Mutual exclusion locks on the queue allow multiple producers to add to the same queue. Condition variables allow consumers to sleep until there is something in the queue as well as allowing producers to sleep while the queue is full. The host operating system's POSIX thread implementation handles the scheduling of runnable threads. In the future we will provide more control over threading so that specific modules can be run in separate threads while other modules share a thread and avoid the performance penalty of passing data between threads.

## D. Dynamic Instantiation

Data-flow graphs need not be static. The system allows new modules to be added and removed dynamically. In particu-

<sup>1</sup>As an optimization, if the record is not in any other queues, then the original record is returned rather than a copy

lar, we allow a module to create a new instantiation of one of its children, and optionally all of that child's children (recursively). This allows us create a new instance of a whole sub-graph of modules. A user may want to calculate a per-host or per-connection statistic. While our system library provides routines that allow modules to keep a table indexed by one or more named fields, we would rather not require that every statistic module know how to keep such a table. Instead, our *split* module can keep a single table, create new instantiations when necessary, and route the record to the appropriate instantiation.

#### E. Distributed Flow

Some data-flow problems require extensive computation. A multi-threaded engine can execute different portions of the pipeline in parallel. To scale beyond what can be performed on a single SMP, distributed data-flow is essential. We already have a `SOCKET` module can be used to join data-flow pipelines on different systems. In the near future we plan to build a more tightly clustered system as well.

#### F. Iterative Programming Interface

Scheduling of analysis modules is inherently tied to the flow of data through the system. The low-level programming interface for module authors is therefore based on producer-consumer callbacks, much like *libpcap* callbacks. This is however a clumsy interface, especially for analysts that lack systems programming expertise. For example, local variables are not preserved between invocations and modules should support poly-instantiation, which precludes the use of global variables. Thus, state has to be explicitly stored in a persistent data structure that is passed to each invocation.

To remove this limitation, we have provided a POSIX-thread shim that allows modules to be written as a single loop that performs blocking reads of new data items and signals when it is done with each item. This provides programmers with a simple iterative model where local variables persist across data objects.

We expect to have large numbers of modules that often perform very quick operations. For example, one may instantiate a reassembly module for each new connection that is seen. Because of the large number and quick operation of modules, we plan to investigate more efficient closure and coroutine mechanisms in the future.

### IV. MODULE TOOLBOX

Our system currently contains 23 modules, an additional 16 type modules, and is growing rapidly. Some of the more widely used modules are summarized in this section. We organize them by I/O or analysis functionality, but the system does not make this distinction itself.

#### INPUT/OUTPUT MODULES

**TABULARINPUT.** Read records from STDIN. Field names (specified as arguments, or column number by default) are assigned to columns. Field types can be specified by appending a colon and the type name to the end of the field name. If no type is specified for a field, it is treated as a double if possible, or a string otherwise.

**PRINT.** Display specified fields of a data record

**PCAPFILE.** This module reads or write records of type 'packet' to/from a tcpdump-style output file. Multiple input files can be specified and each can be compressed. Output files can be rotated after a specified number of bytes.

**PCAPLIVE.** Sniff packets from a network using *libpcap*.

**SOCKET.** Supports distributed pipelines. A receiver can operate in daemon mode and multiplex data from multiple simultaneous or consecutive clients. Type systems in different pipelines are translated across network connections.

#### ANALYSIS MODULES

**COUNT.** Count the number of records or, if one or more fields are specified, the number of occurrences of each value of the tuple of those fields. Annotate the record with a `count` field accordingly.

**PDF.** Consume a stream of records with `count` fields. When the data stream ends or a `refresh` record is received, compute the total count for all records, annotate each record with its fraction of the total as a `probability` field, and produce each of those records.

**TOP.** Given a set of named fields, pass only those records for which the tuple of fields has occurred more than some constant factor above average. An optional limit to the amount of memory to be used can be specified. This causes a hash of counters to be used which passes some probabilistically small number of extra records.

**ENTROPY.** Assemble a vector of probabilities from all records that have a `probability` field. When the last record is received, or a `refresh` record is received, generate a record with the Shannon entropy annotated to it.

**DERIVATIVE.** Given named fields for  $x$  and  $y$ , compute  $dy/dx$  for each pair of consecutive records. Annotate the result to the latter record.

**FILTER.** Filter records that do not match a list of named field criteria including existence, equality and inequality.

**UNIQ.** Given a tuple of named fields, pass only the records where the tuple containing those fields is unique. Unlike the Unix `uniq` program, data need not be pre-sorted.

**ENCRYPT.** Encrypt a named field based on a hash of any number of other named fields.

**INTERVALS.** Record the time of the first and last records of each value of the specified tuple of fields.

**LAST.** Filter out all but the last record of each value of the specified tuple of fields.

**HEAD.** Terminate the system after  $n$  packets have been processed.

**PYTHON.** Runs a specified Python script on each record. Global variables are preserved across records.

**REASSEMBLE.** Use Snort's reassembly preprocessor to generate reassembled packets.

### V. EXAMPLE APPLICATIONS

In this section we present some example analyses performed with the system. These analyses all use straight-forward combinations of these modules to generate meaningful results. While

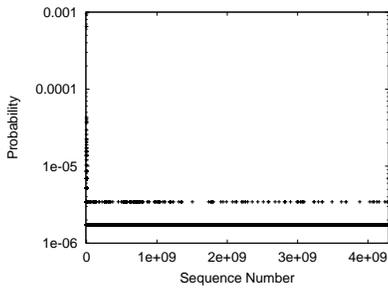


Fig. 3. PDF of TCP Seq. Numbers

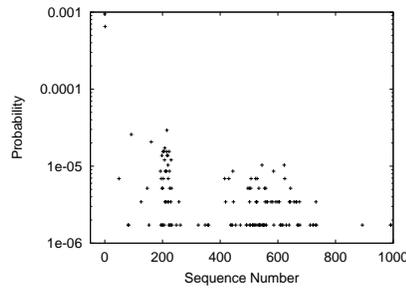


Fig. 4. PDF of TCP Seq. Numbers

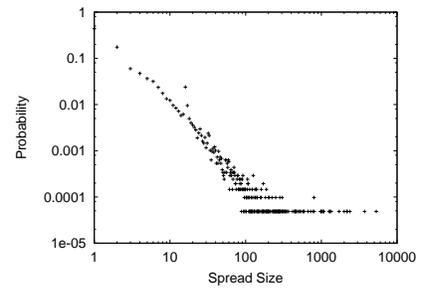


Fig. 5. PDF of Destinations per Source

we present graphical results, the system itself currently only generates tabular data.

### A. Field Distribution

Consider a large set of captured network packets and the following question: ‘What is the distribution of TCP sequence numbers?’ This is an interesting question to ask given the hypothesis that although operating systems should pick initial sequence numbers at random, older operating systems may not, and packets generated by hacker tools may use non-random sequence numbers and could thus be detected.

We compose a data-flow graph in the form of a pipeline that prints out the final probability of each sequence number in our trace. To avoid over-counts due to retransmits, we filter out duplicates of the same sequence number in the same connection tuple (source IP, destination IP, source port, destination port). The COUNT module annotates each packet with a running count. This is useful for monitoring the real-time status of live traffic, but for a finite trace we use LAST to filter out all but the last count for each sequence number.

```
pcapfile - | uniq seq srcip dstip srcport dstport
| count seq | last seq | pdf | print seq probability
```

The discrete probability density function computed over a trace of 1 million packets is shown in Figure 3. The bulk of the large values are at the low end of the sequence space, as shown in Figure 4. Clearly, the distribution is not uniform and a few values are used much more often than average. This discovery deserves follow-up that is beyond the scope of this paper.

### B. Port Scans

Port scan detection is a common task for intrusion detection systems. In order to analyze the complexity of this task or apply probabilistic algorithms, we need to model the frequency and size of port scans. The following pipeline computes the probability density function of *spreads*, where the *spread* of a source is the number of unique destinations (measured as host/port combinations) that it has communicated with. In this example we count the spread for each host and then count the frequency of each spread value and build a PDF from that. The resulting Figure 5 shows that spread has a Pareto distribution [18], [19].

```
pcapfile - | filter srcmac=00:01:02:03:04:05
| uniq srcip dstip dstport
| count -f spread srcip | last srcip | count spread
| last spread | pdf | print spread probability
```

Our packet trace includes traffic traveling in both directions on the Internet connection of a site. If no distinction was made

based on who talked first, a popular web server would exhibit a large spread, even though it is merely responding to external queries. Since our focus is on the behavior of external hosts towards our network, we use the FILTER module to restrict traffic to that coming from the external router interface.

## VI. PROBABILISTIC METHODS

One of the unique contributions of this system is the use of probabilistic algorithms designed to enable statistical measurement of very large traces. For large traces, per-host or per-sequence-number counting consumes prohibitively large amounts of memory. As a result, we resort to probabilistic algorithms and filters that enable us to answer the same questions with high probabilities of correctness.

**Unique Filter:** One fundamental tool is the Bloom summary [20], which creates a bit vector that summarizes the contents of a set. Our UNIQ module has two modes of operation: a precise mode in which elements of the set are placed in a chained hash table and a probabilistic mode in which a Bloom summary is used. The Bloom summary will never allow a duplicate to pass to the next module, but can, with low probability, over-aggressively filter new records. The result in our previous examples would be slight under-counting of events.

**Top Talkers Filter:** The TOP module can also operate in either a precise or probabilistic mode. Probabilistic mode involves a hash of counters in which each entry contributes to multiple counters. The result is an algorithm that, with low probability, may falsely think some sources have large counts. Rather than setting some absolute threshold value, the TOP module filters based on the deviation from an average count. Additional work in this area by our group is presented in [21] and we will continue to add new forms of probabilistic counters and filters to the system.

## VII. AVAILABILITY

The software distribution for this System for Modular Analysis and Continuous Queries is available on the web at [smacq.sourceforge.net](http://smacq.sourceforge.net). We invite community feedback and participation on the project.

## VIII. ACKNOWLEDGEMENTS

The authors’ approach to this problem was greatly influenced by experiences with the McGruff forensics system written by Dan Ridge at NASA. We thank him for an ongoing dialog on system design for large-scale measurement. Paul Weber assisted with the Python type system glue and Bill Barber implemented the interface to Snort’s reassembly functions.

## IX. CONCLUSIONS

In this paper we have presented a general purpose software platform for analyzing large quantities of streaming data. The system can transparently operate on multiple types such as packet traces, netflow data, or logs. Using this system we were able to perform several ad-hoc analyses using a simple syntax and no additional programming. This agility makes the system attractive to a wide audience from researchers and engineers to network operators and security analysts.

Data flow graphs, including simple pipelines, provide a clean abstraction between simple to write modules and the underlying flow of data. This allows the system to be implemented in a variety of ways that improve performance, including zero-copy data passing, multi-threading, and even parallelism across clusters.

## REFERENCES

- [1] D. Parnas, "On the criteria to be used in decomposing systems into modules," *Communications of the ACM*, vol. 14, no. 1, pp. 221–227, 1972.
- [2] Alastair Reid, Matthew Flatt, Leigh Stroller, Jay Lepreau, and Eric Eide, "Knit: Component composition for systems software," in *Proceedings of the 4th Symposium on Operating Systems Design and Implementation*, Oct. 2000, pp. 347–360.
- [3] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An architecture for implementing network protocols," *IEEE Transactions on Software Engineering*, vol. 17, no. 1, pp. 64–76, 1991.
- [4] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek, "The Click modular router," *ACM Transactions on Computer Systems*, vol. 18, no. 3, pp. 263–197, Aug. 2000.
- [5] G. Robert Malan and Farnam Jahanian, "An extensible probe architecture for network protocol performance measurement," in *Proceedings of ACM SIGCOMM '98*, Sept. 1998, pp. 215–227.
- [6] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang, "NiagaraCQ: A scalable continuous query system for internet databases," in *Proceedings of ACM SIGMOD 2000*, 2000, pp. 379–390.
- [7] Samuel Madden, Mehul Shah, Joseph M. Hellerstein, and Vijayshankar Raman, "Continuously adaptive continuous queries over streams," in *Proceedings of ACM SIGMOD 2002*, 2002.
- [8] Vern Paxson, "Bro: A system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23–24, pp. 2435–2463, Dec. 1999.
- [9] V. Paxson and C. Saltmarsh, "Glish: A user-level software bus for loosely-coupled distributed systems," in *Proceedings of the Winter 1993 USENIX Conference*, San Diego, California, 1993, pp. 141–156.
- [10] "Network flight recorder," <http://www.nfr.com/>.
- [11] Mark Lutz, *Programming Python*, O'Reilly and Associates, 1996.
- [12] Van Jacobson, Craig Leres, and Steven McCanne, "libpcap," 1994, <http://www-nrg.ee.lbl.gov/>.
- [13] Van Jacobson, Craig Leres, and Steven McCanne, "tcpdump," <ftp://ftp.ee.lbl.gov/>.
- [14] Vern Paxson, "tcpslice," <ftp://ftp.ee.lbl.gov/tcpslice.tar.gz>.
- [15] Martin Roesch, "Snort - lightweight intrusion detection for networks," in *Proceedings of the 13th Systems Administration Conference*. 1999, USENIX.
- [16] Gilad Bracha and William Cook, "Mixin-based inheritance," in *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications*. 1990, pp. 303–311, ACM Press.
- [17] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen, "Classes and mixins," in *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 1998, pp. 171–183, ACM Press.
- [18] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos, "On power-law relationships of the internet topology," in *Proceedings of ACM SIGCOMM '99*, 1999, pp. 251–262.
- [19] Lada A. Adamic, "Zipf, power-laws, and pareto - a ranking tutorial," Tech. Rep., Xerox Palo Alto Research Center, 2000.
- [20] Burton Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.
- [21] Cristian Estan and George Varghese, "New directions in traffic measurement and accounting," in *Proceedings of ACM SIGCOMM 2002*, Aug. 2002.