VAST: Visibility Across Space and Time

Matthias Vallentin[1,2] and Himanshu Sharma[1]
{vallentin,himanshu}@cs.berkeley.edu

[1] *UC Berkeley*
[2] *International Computer Science Institute*

Abstract

Key operational networking tasks, such as troubleshooting and defending against attacks, greatly benefit from attaining views of network activity that are unified across *space and time*. This means that data from heterogeneous devices and systems is treated in a uniform fashion, and that analyzing past activity and detecting future instances follow the same procedures.

Based on previous work that formulated principles for comprehensive network visibility [1], we present the practical design and architecture of *Visibility Across Space and Time (VAST)*, an intelligent database that serves as a single vantage point into the network. We implemented a proof-of-principle prototype that can archive and index events from a wide range of sources. Our preliminary performance results indicate that the underlying event engine needs further tuning in order to deploy the system in large-scale operational networks.

1 Introduction

Many administrative tasks in operational networks today rely on descriptions of network activity that are fragmented in both *space and time*. By fragmented in space, we allude to the heterogeneity of disparate data sources and programmatic interfaces that the network operator has to incorporate to obtain a unified view of the network data. By fragmented in time, we refer to the discrepancy in analysis procedures between past and future activity (e.g., processing plain-text IDS logs versus configuring firewall rules; using scripts to extract information from the logs versus pro-actively configuring alert generation).

In this paper, we provide the practical component to previous research which set out principles of comprehensive network visibility [1]. We devised the architecture and implemented the first prototype of *Visibility Across Space and Time (VAST)*, a platform that serves as a single vantage point on network activity. VAST is designed to archive descriptions of activity from numerous sources in the network and to query past activity for retrospective analysis. Based on a policy-neutral core, the architecture allows a site to customize operations to coalesce, age, sanitize, and delete data.

The realization of such a system requires addressing a number of challenges. First, we need a flexible and expressive data model to unify heterogeneous descriptions of activity. This corresponds to principal *D.1* from [1]. A particularly apt model for this purpose is the *publish/subscribe* scheme, which is based on asynchronous communication using events. In this model, interacting peers publish the availability of an event and subscribe to events published by peers according to local interest.

Second, we need to separate mechanism from policy (principle *D.6* in [1]). Many security devices limit their focus only on real-time detection but ignore the additional step of providing policy-neutral output that can be highly useful in the future.

Third, we have to realize that the temporal dimensions of analysis extend beyond the duality of past and future. By considering an additional *what-if* perspective, we seek to automatically explore the suitability of analysis patterns codifying future behavior if applied to past activity (principle *C.2* in [1]). For instance, false positives can be greatly reduced by testing detection procedures against previous network behavior.

Fourth, a large measurement window with respect to time is needed (principle *D.2* in [1]). Most intrusion detection systems operate in real-time today and discard the incoming data after processing it. However, an extensive archive allows re-inspection of activity that became interesting only in retrospect.

Finally, we argue to avoid “clean-slate” designs (principle *B.2* in [1]). In operational practice, the very challenge in creating a coherent picture of network activity is often exacerbated by the chaotic details such as heterogeneity and policy frictions. Therefore, we emphasize *incremental deployability* to achieve practical usability.

The remainder of the paper is organized as follows. In § 2, we sketch possible scenarios that would significantly benefit from the existence of VAST system. Then, we refer to related work in this field in § 3 and compare our approach to existing ones. After introducing the technological building blocks of VAST in § 4, we explain its architecture in § 5 and prototype implementation § 6. We present a performance study in § 7 and give recommendations for future work in § 8.

2 Application Scenarios

Different scenarios in operational networks face substantial problems due to the fragmented nature of descriptions of network activity. In the following application fields presented by [1], we point out practical hurdles and problems and argue why an integrated view of network activity would significantly improve the situation.

Troubleshooting. Troubleshooting network problems requires an operator to gather detailed information from disparate sources. Pin-pointing the exact error source is made difficult by the layered, modular structure of network architecture and protocols. The more context they can draw upon, the easier it becomes. Being able to characterize the problem sources from as many perspectives as possible is of high value.

Also, allowing the operator to integrate analogous analysis procedures for future activity would be a great utility. For example, consider the example of a failing automated remote backup caused by wrong permissions of the SSH configuration directory. An operator could codify the automated monitoring of SSH server logs to trigger an automated inspection of the directory permissions on the failing client.

Detecting Intrusions. Most Intrusion Detection Systems (IDS) focus on a particular type of monitoring data (either host logs or network packets). A *hybrid* system that synthesizes data from a variety of sources (virus scanners, user keystrokes and interactions with peripheral devices, firewall logs, DNS lookups, internal NetFlow records, and honeypot data) enhances the analysis context of the IDS.

Post facto forensics constitute an important aspect of comprehensive incident response. A high quality archive allows the security analyst to assess the real depth of an intrusion and quantify the scope of the caused damage. A unified log archive makes forensic analyses less error-prone and more precise, and saves a significant amount of time, which is key to defend against an ongoing attack.

It is also highly beneficial for a security analyst being able to codify security breaches in a consistent fashion in order to automatically detect future instances of a certain attack pattern. Moreover, the analyst could extend the temporal dimension to *what-if*, i.e., automated exploration of how analyses intended for future activity would have behaved if applied in the past.

Combating Insider Abuse Dealing with security breaches from the inside is similar to intrusion detection, but it is difficult to comprehend the full scope of the attack at the time of detection because the complete attack comprises of a specific sequence of actions, each of which the attacker was authorized to do. For example, an employee accessing a sensitive machine, copying documents to an intermediate machine to conceal the malicious intent, and finally sending the sensitive data to a non-authorized party.

Insider attacks can manifest over long time intervals. When trying to understand the full extent of a breach, a comprehensive archive that reaches far back in time is particularly helpful to reveal initial attempts to hide the abuse or to uncover similar non-authorized activity.

3 Related Work

The term *visibility* has been used in different contexts in the past. Cooke et al. [6] use the term network-wide visibility to refer to the ability of accessing clear-text data at the network-level, which is hardly possible in the presence of encryption and tunneling. To regain visibility, the authors present *Anemone*, a monitoring infrastructure deployed on endpoints where data flows can be accessed in clear-text.

The *Time Machine (TM)* is the closest to our work. It efficiently records network traffic streams to support inspection of activity that turns out to be interesting only in retrospect [14]. Similar to VAST, the TM aims to improve key operational networking tasks — such as troubleshooting and forensic analysis. While the TM provides high-performance network traffic capture, indexing and retrieval, VAST extends these concepts to arbitrary events that can stem from a wide range of sources. Similar to the TM's architecture, VAST is parallelized from scratch to exploit the performance benefits of modern multi-core CPUs.

One major aspect in the design of VAST is the storage component which continuously archives and indexes incoming events for later retrieval. Approaches geared to off-line network traffic analysis [5, 20] are not designed to deal with live streaming data [8, 21]. In the following, we briefly sketch some of the existing approaches by streaming databases.

The *CoMo project* [10] provides a network monitoring infrastructure featuring both live and retrospective queries on streaming data. In a distributed setup, multiple CoMo systems can propagate queries to pinpoint relevant data locations in the network. Despite its high-speed layout, network traffic is stored in a circular on-disk buffer which imposes functional limitations.

TelegraphCQ [3] builds an adaptive stream processing engine for high-volume data streams based on the open-source database PostgreSQL. Recently, the system has been extended to allow querying both historical and real-time data simultaneously [19, 4]. To work around the expensive I/O operations for fetching historical data, the approach retrieves only a sampled — and thus incomplete — version of the incoming data stream.

An interesting hybrid approach follows *Hyperion* [7], a system for archiving, indexing, and on-line retrieval of high-volume data streams. Their write-optimized streaming file system, StreamFS, poses an alternative to the traditional relational database storage mechanisms. The low-level programming and file system implementation is only available for Linux.

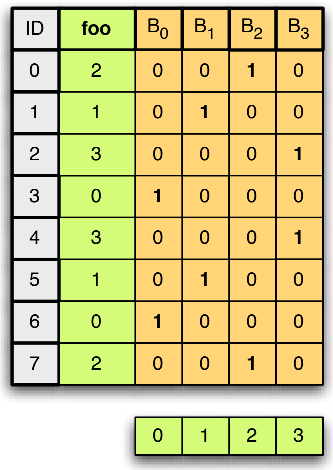
4 Technologies

VAST continuously receives and relays streams of events, and stores them permanently for later querying. Traditional *data base management systems* (DBMS) support *historical queries* over data that already exists before the query is issued. In contrast, *data stream management systems* (DSMS) answer *live queries* that incorporate new arriving data after the query is issued. VAST is a *hybrid* system that answers both historical and live queries.

Queries sent to VAST usually process large quantities of historical data. At the same time, the system has to archive a continuous data stream. Since the high I/O costs of delivering historical query results can prevent the system from processing the incoming stream of data, it is important to accelerate historical query through indices.

On the one hand, *B-Tree* [2] indices exhibit similar computational complexities for searching and updating and are hence apt for *on-line transaction processing applications* (OLTP) where write and update operations occur more often than searching.

On the other hand, *Bitmap indices* [17, 24] perform better than B-tree indices for *on-line analytical processing* (OLAP) applications where search frequency is higher than update frequency [23, 24, 16]. Although updating bitmap indices is known to be expensive [24], it has been shown recently that efficient updates are indeed



ID	**foo**	B_0	B_1	B_2	B_3
0	2	0	0	**1**	0
1	1	0	**1**	0	0
2	3	0	0	0	**1**
3	0	**1**	0	0	0
4	3	0	0	0	**1**
5	1	0	**1**	0	0
6	0	**1**	0	0	0
7	2	0	0	**1**	0
		0	1	2	3

Figure 1: A bitmap index generated from 4 distinct values

possible [19]. As historical data is never modified, new data can be appended without disturbing existing data. Additionally, some indices can be built on unsorted data which allows adding new data in time that is a linear function of the number of new records.

In general, bitmap indices are based on bit arrays (called *bitmaps*) and answer queries by performing bitwise logical operations which are well supported by modern processors. To index a single column in a table, the bitmap index generates c bitmaps, where c is the number of distinct values in the column (also referred to as *column cardinality*). Each bitmap has N bits, with N being the number of records in the table. If the column value in the kth row is equal to the value associated with the bitmap, the kth bit of the bitmap is set to 1, but remains 0 for any other column value.

Figure 4 depicts a simple bitmap index for the column `foo` that consists of integers ranging from 0 to 3. Since the column cardinality is 4, the index includes 4 bitmaps, named B_0, B_1, B_2, and B_4. For example, the third and 5th bit of B_3 are set to 1 because the values in the corresponding rows are equal to 3. Answering a query such as "`foo` < 2" translates into bitwise OR ($|$) operations between successive long-words of B_0 and B_1. The result is a new bitmap that can be used in further operations.

FastBit To archive, index, and query historical data, VAST uses *FastBit* [22] for its storage engine. FastBit implements many efficient bitmap indexing methods with various encoding, compression, and binning strategies. A distinctive feature of FastBit is its efficient compression method, the *Word Aligned Hybrid* (WAH) method [23, 24]. WAH employs *run-length encoding* (RLE) to reduce the size of each bitmap in a bitmap index so that the total size of the index remains modest regardless of the type of data.To minimize I/O operations on index lookups, FastBit stores bitmaps in one index in linear order.

Bro To describe network activity, VAST leverages the event model of the flexible open-source Bro NIDS [18] developed and maintained by Vern Paxson. Raw packets from the network interface are received through *libpcap* [13]. Employing libpcap enables flexible deployment in various UNIX environments and greatly reduces the traffic in the kernel by applying a *BPF* filter expression [15]. The pre-filtered packet stream is then handed up to the next layer, the *event engine*, which re-assembles the packet streams and elicits events.Since the resulting events represent a *policy-neutral* snapshot of visible network activity, they perfectly fit into our event model.

Broccoli To enable third-party applications partaking in the event communication, the light-weight client library *Broccoli*[1][12] has been developed, allowing other applications to request, send, and receive Bro events.

5 VAST Architecture

This section describes the architecture of VAST. After framing the technical challenges we follow throughout the system design, we give a high-level overview about VAST's components and then delve into each component in detail.

5.1 Challenges

VAST can be seen as an intelligent fusion of DBMS and DSMS. Consequently we face challenges and limitations of both technologies. It is an ambitious undertaking to merge the functionality of both approaches as they are geared towards different application domains.

Real-Time Nature. As a single sink for incoming data in form of a continuous stream of events from all components interfacing with the VAST, the system must provide enough resources to archive the arriving events quickly and entirely, without foregoing new data.

High Concurrency. System must be able to adequately process the incoming event stream, while at the same time offering enough resources to perform parallel tasks, such as answering retrospective queries, dispatching event feeds, reorganizing index structures, and constantly aggregating old events to maximize storage capacities; emphasizing the need for a *concurrent* and *asynchronous* system architecture.

[1]*Broccoli* is the healthy acronym for "Bro Client Communications Library".

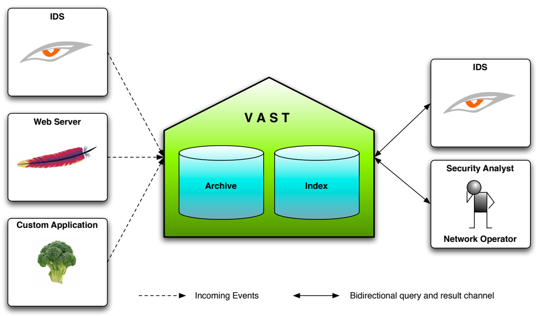


Figure 2: High-level overview of VAST.

Compatibility. The system should provide the ability to transform internal state of arbitrary devices into events that can be sent to the VAST system yielding a powerful mechanism to unify the communication of heterogeneous components.

Commodity Hardware. Finally, the system should not require expensive custom hardware to achieve the desired performance. Instead we prefer to employ commodity hardware to enable cost-effective deployment. Ideally, performance bottlenecks can be countered by adding a new machine that overtakes a dedicated task to unburden a component that reached its resource limits.

5.2 Overview

VAST is a scalable, distributed system that archives events from a wide range of sources and offers a query interface to extract a stream of events. As illustrated in Figure 2, an event source could be a NIDS, web server, logging daemon, or an arbitrary *Broccoli-enabled* application. VAST archives the incoming event streams by storing a raw copy in the *archive* and indexing the event details in a separate *index*.

An interactive query interface allows users to extract historic activity in form of events. For example, the interface supports retrospective queries like "which events include activity of IP address *X* today" or "which events in the last month included access of URL *Y*". The query result consists of a list of events that match the given condition. In a further step, the user can inspect and parse the event to extract the desired details.

Further, users should have the ability to install *live queries* that remain active. By subscribing to a specific event pattern, such as "all URLs of this form", VAST instantiates a filter for all future events matching the given condition. The corresponding query is registered as an *event feed* which continuously relays the qualifying events to the user or a remote application.

5.3 Data Representation

VAST leverages the event model of Bro and uses *Broccoli* to outfit third-party applications with the capability to communicate in the same event model. VAST acts both as sink and source of events, that is, it represents a *transparent* actor in the network since events can be sent to and received from the system in a uniform fashion. This flexibility is particularly beneficial when integrating the VAST system into larger setups that span across the boundaries of the local administrative domain.

5.4 Event Specification

Events in the Bro event model consist of a name and zero or more typed *arguments*. To minimize the space-overhead when sending events across the network, only event name, argument data, and the type information are transferred, but not the argument names. We refer to these compact representations that do not contain the argument names as *raw events*.

The missing semantic context is codified in the *event specification*, a document that VAST maintains to manage event meta data. Separating the event structure from their semantics gives us a flexible mechanism to change the meaning and behavior of events when they are processed. For example, it is possible to rename arguments, change the way they are indexed, or link together similar arguments of different events. VAST's specification language is similar to the Bro scripting language and offers both primitive types such as `addr`, `count`, `string`, etc., and compound types such as `record`, `vector`, `set`, and `table`.

To group related or similar semantic activity, an event is assigned to zero or more *classes*. A class consist of a list of arguments and the contributing events. By default, every event generates a separate class because a computer cannot deduce semantic relationships between events reliably. The following examples illustrates the benefit of classes:

```
event http_request (
    src: addr,
    dst: addr,
    request: string
)

event ftp_request (
    client: addr,
    server: addr,
    command: string
)
```

From a user perspective, the query "which events from the last hour involve communication between host *A* and host *B*?" should ideally consider both events. The corresponding query that connects (or *joins*) the two events would be similar to:

```
(http_request.src = A OR ftp_request.client = A)
                    AND
(http_request.dst = B OR ftp_request.server = B)
```

With hundreds of events, this explicit connection *at query time* not only introduces a significant overhead for the query issuer, but is also quite error-prone. Therefore, the event specification allows factoring out related information. The following change uses the `&map_to` attribute in the specification to create a separate class `conn` which holds IP endpoint information:

```
event http_request (
    src: addr &map_to="conn:orig"
    dst: addr &map_to="conn:resp"
    request: string
)

event ftp_request (
    client: addr &map_to="conn:orig"
    server: addr &map_to="conn:resp"
    command: string
)
```

Although not explicitly listed as such in the specification, the class `conn` has the following internal structure based on the above mappings:

```
conn ( orig:  addr, resp:  addr )
```

Our initial query looks now much simpler and expresses the intent of the user in a more natural fashion:

```
conn.orig = A AND conn.resp = B
```

5.5 Archiving Events

A high-level overview of the event archival process is visualized in Figure 3. All three components involved in the archival process are designed to operate in a distributed fashion. That is, each component can be deployed on a separate machine and they communicate over a network connection. Because each component is self-contained and has an associated event queue, scaling components across multiple machines is naturally enabled by the system design.

In order to interact with VAST, event producers connect to the *event dispatcher* and send their events through a Broccoli communication channel. The dispatcher assigns each incoming event a unique monotone 64-bit

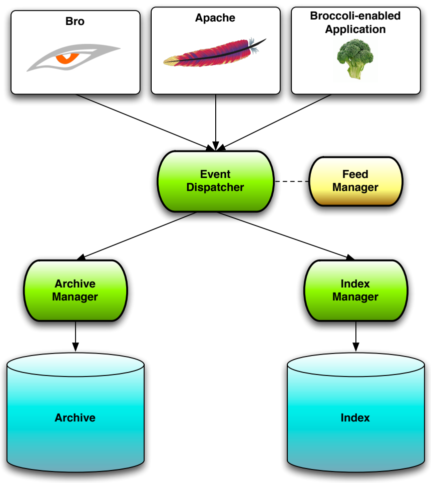


Figure 3: Archiving events from multiple sources.

identifier and creates two copies of the event: The first copy is the *raw event*, an unmodified version of the serialized Broccoli event augmented with the obtained 64-bit ID that is compressed and forwarded to the *archive manager*. The second copy is the *VAST event* which is converted from a raw event into an internal data structure. It is forwarded to the *index manager* which access the event arguments through a well-defined interface. We will discuss the *feed manager* later in the context of live queries (see §5.6.2).

The archive represents a comprehensive record of all events that have ever been sent to the VAST system. It stores events unmodified, i.e., in their serialized form as obtained from Broccoli. Archiving raw events rather than the VAST events has the advantage that query results can directly be sent over the network using Broccoli. The archive consists of a collection of *containers* offering a table-like interface to append data. Figure 4 visualizes an archive and index container. The former has two columns to store evens associated with their ID, the latter has an ID column and one column for each argument of the class that this container represents.

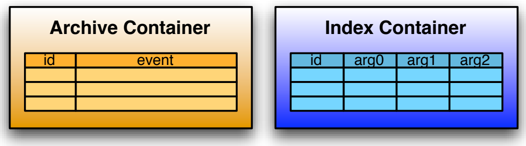


Figure 4: Archive and index containers.

Common among archive and index is a specific *slicing* scheme. A slice is an abstraction that maps an object with an identifier to specific bucket. The archive uses *event ids* as identifier and the objects are *containers*. The index uses *timestamps* as identifier that map a *set of containers* (also referred to as *index slice*). Slicing is similar to the *binning* strategy employed by FastBit, only at a higher level of abstraction.

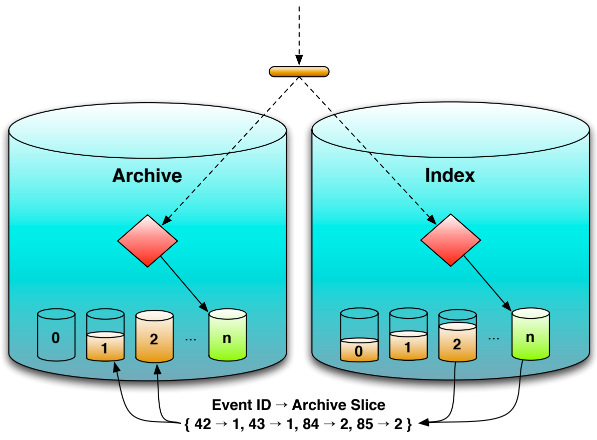


Figure 5: Slicing of archive and index.

This technique partitions data horizontally. As illustrated in Figure 5, an incoming event is both sent to the archive and index, where the corresponding slice is determined. Because object identifier are monotone, the event is usually appended to the last slice.[2] When looking at the index to find specific events, the list of returned event ids allows us to efficiently locate the corresponding events in the particular archive slice (see §5.6.1).

Slicing helps only to bypass FastBit's maximum container size, but we also expect to use it in conjunction with VAST's aging and aggregation framework to expire slices with no more valid entries.[3] The slice interval is customizable by the user and denotes the maximum number of events to be stored per container for the archive, and a time interval in seconds for the index. For example, if the slice intervals were 20,000 and 60, an archive slice would then hold 20,000 events and all events arriving in the same 60 seconds interval would belong to the same index slice.

When issuing queries like "which events include hosts that accessed IP address X more than n times in the last hour", we usually formulate a query condition that refers to a particular event argument. In the above example, we are interested in events that have an argument that

[2]Due to multi-threading or components distributed across multiple machines, it is possible that *out-of-order* events arrive that have to be sent to a previous slice.

[3]At the time of this writing, aggregation and aging only exists conceptually. In this context, slicing helps us to overcome the append-only nature of FastBit partitions by deleting a partition entirely after all its entries are marked as invalid through an aggregating process.

represents a destination IP address. One approach to extract the relevant events would to iterate over the entire archive and inspect each event. This is clearly an inefficient strategy for large event archive. To improve query performance, we therefore index each argument value by associating it with the corresponding event ID.

Continuing the above example, consider the fictitious event `transfer` with the following structure:

```
event transfer (
    from: addr,
    to: addr,
    data: string
)
```

An event with such a simple structure translates to a class that consists of the same arguments. The corresponding container would contain columns for the three arguments, plus one column for the event ID. That is, the column names would be `id`, `from`, `to`, `data`. When indexing an instance of this event, the ID and each argument are written in one container row. Unfortunately, we encounter also more complex scenarios in practice: events can include arguments which belong to several classes (e.g manually remapped arguments).

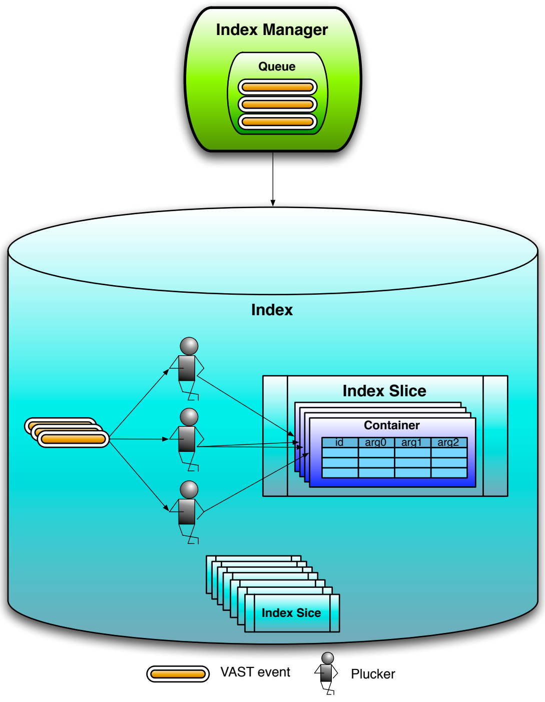


Figure 6: Pluckers that index events.

To counter these intricacies, we introduce the *plucker*, a dedicated entity to distribute all arguments of one event to the corresponding containers. Upon slice creation, the plucker consults the event specification to generate a list of containers that have to be accessed for a particular event. When an event arrives, the plucker performs the following steps. First, it sends the event ID to all containers that have to be accessed for the given event. Second, the plucker iterates over the event arguments and writes the data of each argument to the corresponding container. Similar to a database transaction, the plucker finally commits the involved containers when all arguments have been processed successfully.

5.6 Event Retrieval

VAST offers operators and security analysts a centralized vantage point where they can retrieve information about past and future activity. Moreover, remote applications such as a NIDS can interact with VAST to obtain a policy-neutral event stream to analyze or subscribe to specific set of events that are forwarded when they enter the system. To issue a query, a client connects to the *query manager*. This component provides a well-defined interface to access to archive and index over the network. We will cover the query manager in-depth later in this section.

Queries sent to VAST are represented as events as well. The advantage of this approach is that we can employ Broccoli to create an asynchronous, bi-directional communication channel. While the client uses this channel to issue queries, VAST uses the same channel to send the qualifying results to the client.[4]

Furthermore, a client can specify a query target different from itself to redirect the query result to a different machine. For example, this proves beneficial when an analyst issues a manual query to feed a stream of events into an intrusion detection system. In the following, we first discuss the design of historical queries and thereafter turn to live queries.

5.6.1 Historical Queries

The schematic process of a historical query is sketched in Figure 7 and involves the six following steps.

1. Both users and remote application can issue queries by creating a *query event* locally and sending it to the query manager. Each incoming query is associated with a dedicated *query queue* that is processed asynchronously to deliver matching events back to the client.

2. Executing the query instructs the query manager to parse the event, extract the query condition and access the index to obtain a list of event ids that match the condition.

[4]Note that query events take a different data path into the system than events that are archived and indexed. Rather than connecting to the event dispatcher, clients that issue queries connect to the query manager which listens on different port.

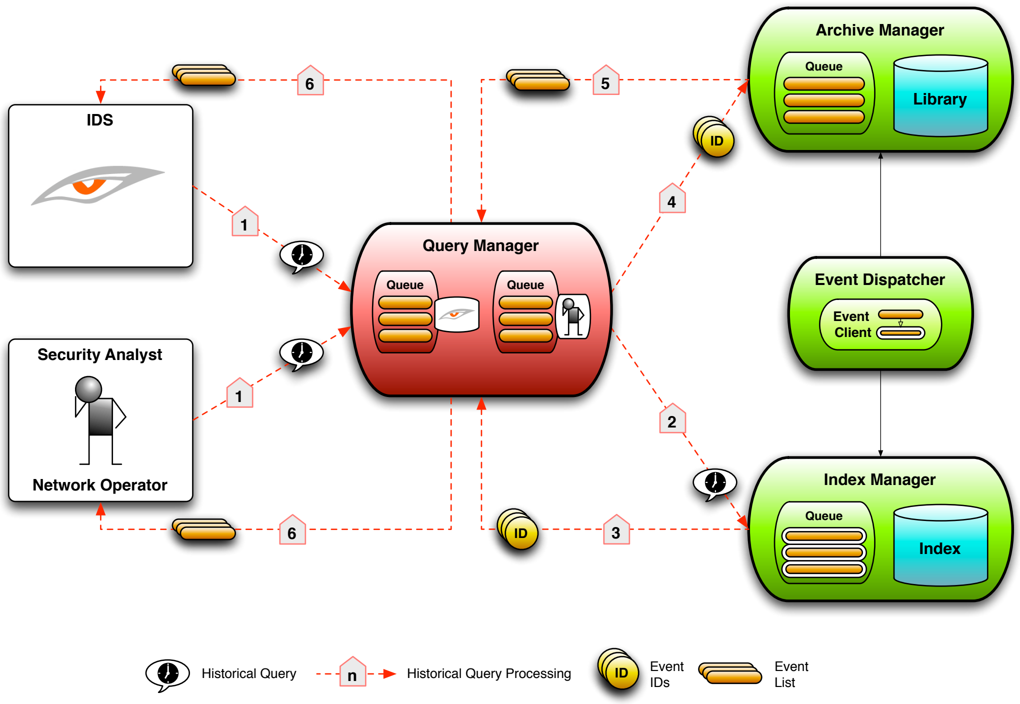


Figure 7: Historical query processing.

3. The index evaluates the condition and sends a list of matching ids back to the index manager.

4. If the list was empty the query manager signals the client that the query did not yield a result and terminates the connection. Otherwise, it consults the archive to extract the events with the given list of ids.

5. The archive manager then sends the extracted events back to the query manager which inserts them in to the corresponding query queue.

6. As soon as events are inserted into the client queue, the query manager forwards the events over the existing Broccoli connection to the client.

5.6.2 Live Queries

In contrast to historical queries, *live queries* do not consult the event archive. Instead, they install an *event feed* which continuously relays matching events to the query issuer.

On the client side, the only difference to historical queries is a flag indicating the query type. On VAST side, live queries differ significantly from their historical counterpart. While historical queries can simply be converted into one or more index lookups, live queries exhibit a different nature. Instead of "pulling" data from disk, a constant stream of events is "pushed" into the system. This subtlety has significant implications for the design of the live query architecture. In particular, a very high or bursty event arrival rate can impose an unpredictable workload and thus imposes limitations on the system resources required for processing the event stream.

In historical query architectures, queries are brought to the event data. Live queries have diametrically opposed characteristics: data is brought to the queries. This twist necessitates a different architecture for live queries. Our design is depicted by Figure 8. Issuing a live query consists of the following steps:

1. Both users and remote applications can issue queries by creating a *query event* and sending it to the query manager.

2. The query manager parses the event and forwards it to the feed manager which creates a live query based on the given condition. Each VAST event is "pushed" into the feed manager and compared against all live queries.

3. If a match was successful, the feed manager requests the corresponding raw event from the dispatcher and sends it to the associated query queue of the query manager.

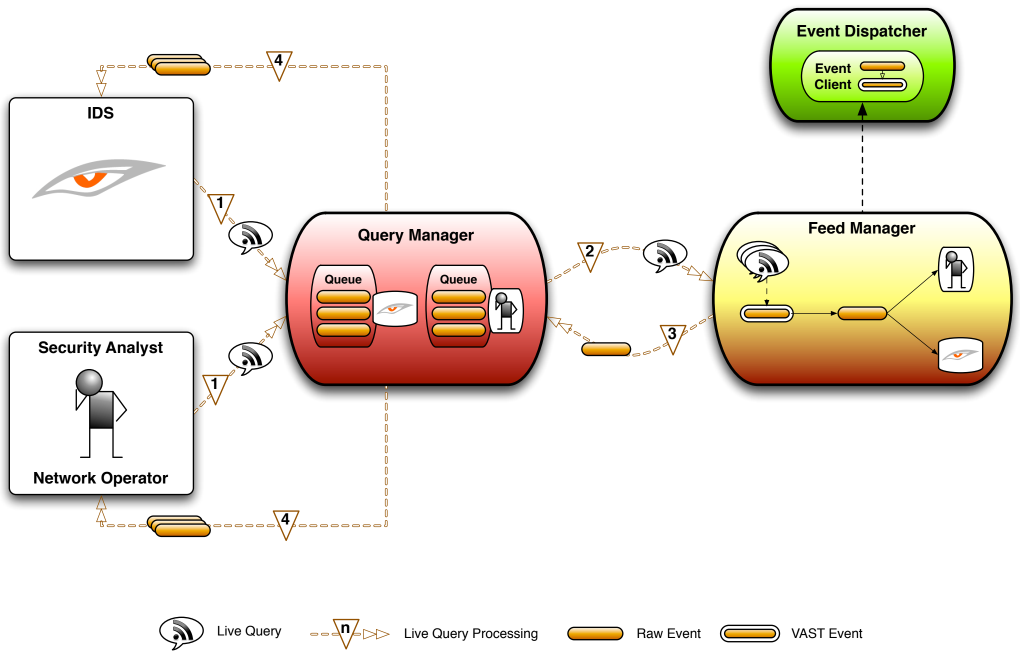


Figure 8: Live query processing.

4. As soon as events are inserted into the client queue, the query manager forwards the events over the existing Broccoli connection to the client.

A key challenge in the design of the live query architecture is avoiding big performance penalties. Because each incoming event has to be matched against all live queries, the processing overhead can be non-negligible when facing large event volumes. To cope with a very high event load, concurrent query execution could substantially ameliorate the scalability of the system.

Another issue represents the retention strategy for raw events. Because the feed manager asks for any raw event after encountering a positive match, the dispatcher would have to keep the raw events in memory until the corresponding VAST event is processed. One possible solution would be to forward the raw event to the feed manager as well. While it would remove the dependency to the dispatcher, it greatly increases the data sent across the network in a distributed setup.

Note that the live query architecture is still in a very fledgling stage. Hence we have not yet found an ideal solution but would highly appreciate feedback to come up with a tractable solution.

6 Implementation

We now delve into technical facets of VAST and describe interesting aspects of the implementation. VAST is implemented entirely in C++ and currently consists of 7,102 LOC including comments. Since our long-term goal is not only to create a prototype but an application to be employed by large institutions, we emphasized a careful development process and distilled separate modules that have small interfaces. Our implementation has a layered architecture with lower layers not depending on higher layers.

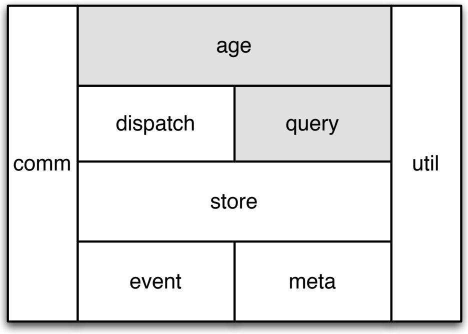


Figure 9: VAST implementation: higher layers only depend on lower layers.

As shown in Figure 9, the building blocks of the system are the `meta` and `event` layer. While the former implements the event specification with event and type meta information, the latter contains data structures used during event processing, such as raw events and VAST events. On top resides the `store` layer which provides storage abstractions for archive and index, the slicing logic, and an enhanced FastBit interface for data streaming. Both the `dispatch` and `query` layer make use of the storage engine. Finally, aging and aggregation is the highest layer as it act like a client that performs queries and inserts new events. Note that the shaded layers (`query` and `age`) are not yet implemented.

6.1 Event Compression

When archiving a high event volume over a very long time window, already small efforts to keep data growth under control have a substantial effect in the long run.

Recall that the event dispatcher creates a raw byte copy of the incoming Broccoli event. Storing raw data (also referred to as *blob* data) in a FastBit is only supported by means of NULL-terminated C-strings. However, the contents of the raw Broccoli event can include NULL bytes and thus cannot be stored in its pristine form. In order to avoid the NULL bytes, we encode the raw bytes constituting the Broccoli event. To this end, we created a custom run-length encoding that achieves a compression rate of approximately 30%.[5]

Our byte-stuffing algorithm is simple and straightforward: each non-NULL character is copied without modification. NULL bytes are replaced with a special *encoding symbol* followed by a *null symbol*. The byte after the null symbol holds the number of consecutive zeros in the raw byte stream. If this number becomes 256, we write the value 255 (or `0xFF`), reset the NULL byte counter to 1, and advance to the next byte. We continue this procedure until the number of zeros remains less than 256. If we encounter the encoding symbol in the raw byte stream, we simply duplicate it in the encoded byte stream.

Although we could use more sophisticated algorithms with higher compression rates, we explicitly chose a simple encoding scheme given that disk space is cheap as opposed to CPU cycles.

7 Evaluation

In the following we present our preliminary performance results of VAST's building blocks, the event and storage layer. At first we explain our methodology in § 7.1, then evaluate the performance of the event layer and storage layer in § 7.2, and finally discuss the storage overhead in § 7.3.

[5]Raw Broccoli events contain apparently enough consecutive NULL bytes to yield such a compression rate

7.1 Methodology

In order to generate reproducible measurements we resort to trace-based offline analysis. As event source, we run Bro on a `pcap` trace, enabling a large fraction of its analyzers to induce a high event load. VAST listens on a TCP socket at port 42,000, waiting for event producers to connect. Although VAST supports multiple event sources in parallel, we only use one connecting Bro instance in our measurements.

Generally, trace-based analysis entails a "compressed" notion of time because the trace is processed as fast possible, without respecting the inter-packet arrival gaps. To generate a more realistic workload, we use Bro's *pseudo-realtime* mode which insert processing delays while receiving packet input from a trace. These delays equal to the natural inter-packet arrival gaps. Internally, Bro defers packet processing until the time difference to the next packet timestamp has elapsed. All our measurements are conducted with a pseudo-realtime factor of one, thus reflecting real-time trace replay.

We conduct our measurements with a packet trace of traffic from the *International Computer Science Institute (ICSI)* in Berkeley. The trace has been recorded on November 6 and spans 24 hours, ranging from 01:45:02 AM to 01:45:04 the next day. It contains 32,923,230 packets from 447,893 IP flows (avg. 73.51 pkts/flow) that incorporate 155,394 distinct source addresses. The observed average rate is 1.81 Mbps (standard deviation 3.97 M) and the peak rate amounts up to 92.45 Mbps. 95.15% bytes are TCP traffic, whereas 4.56% constitute UDP traffic. The most prevalent protocols in terms of byte are HTTP (66%) and HTTPS (2.81%).

7.2 Performance Analysis

To get a detailed picture of the resource usage, we employ the Goggle perftools [9] to both understand memory footprint and CPU usage. The heap profiler indicates that VAST has very low memory footprint of 29.7 MB. Our expectation was that most of CPU time would be spent in VAST's storage layer. However, it turned out that 52% of time was spent in the event layer. We then conducted a new measurement to analyze this layer in isolation. To this end, we turned off the storage engine by uncommenting the relevant portions of code in the Broccoli event handler that is executed when event arrives.

According to the perftools profiler, the function `__bro_ht_free()` and its callees spent 51.7% of CPU time. We assumed that the small object allocation of numerous empty tables caused this high figure and hence worked with the Bro developers to investigate the issue. After patching `__bro_ht_free()` to prevent the allocation of slots when the hash table is empty, the CPU consumption of this particular function could be reduced to to 6.3%. As a result, the maximum processing rate of events doubled from roughly 1,500 to 3,000 events per second.

We now turn to the actual event processing overhead of VAST's storage layer. As baseline, we run VAST with empty event handlers as above, thereby disabling the storage component. The event processing rate is displayed in Figure 10(a). The corresponding event queue size on the Bro side is shown on the right side in Figure 10(b). The average event arrival rate per second is 420.3 with a standard deviation of 460.9, and a peak rate of 2,959. Then, we enabled the storage component of VAST and observed event rates and queue sizes as shown as in Figure 10(c) and Figure 10(d). In this case, the average event arrival rate was 307.8, the standard deviation 243.3, and the maximum number of events per second 1707.

Unfortunately, we could not run our measurements to completion. The sheer volume of the event stream caused Bro to terminate the network connection. Too many events on the VAST side arrived that could not be consumed by Broccoli and handed up to VAST. After Bro reaches a threshold of 250,000 events, it closes the network connection. Therefore, our measurement with enabled storage engine reflects the first 10% of the measurement with empty event handlers. That is, Bro terminates the connection after reaching the first peak in Figure 10(a).

Our results confirm an earlier result: 52% of CPU time is spent in Broccoli, thus reducing the event arrival rate by a factor of two. Generally speaking, the analysis suggests to concentrate on further improvements in the event layer before optimizing higher layers.

7.3 Storage Overhead

We now investigate the storage overhead of raw events on disk. VAST could archive and index 1,180,000 events until Bro closed the network connection. The size of the archive amounts to 4.2 GB and the index occupies 377 MB (11.48%) of disk space. Archive meta data (64bit identifier and container meta data) constitutes 0.098% of the entire archive volume.

Figure 11 characterizes the types of events that have been archived. The event with the highest frequency was `conn_weird` (428,778 times). Bro generates this event whenever it encounters a "weird" connection, such as an unexpected TCP state change. The second highest frequency has `packet_contents` (249,013 times) which contains the raw packet contents of a particular connection.

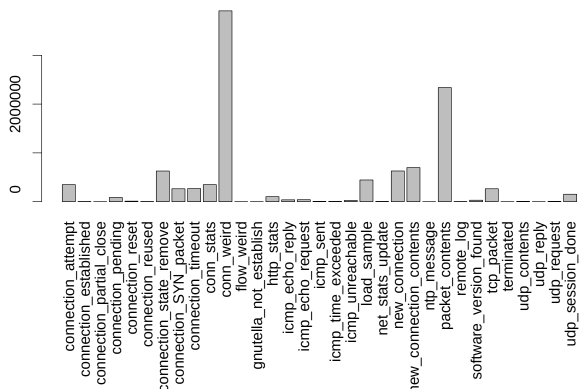


Figure 11: Distribution of different event types.

8 Future Work

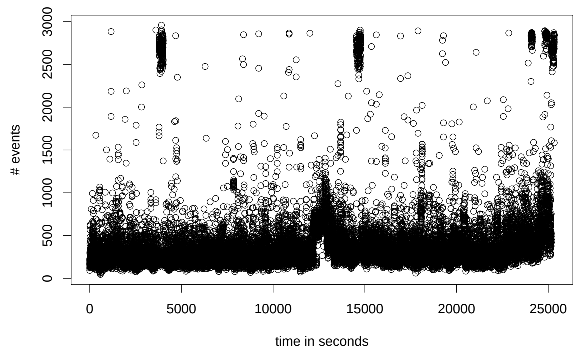
As illustrated by our performance study, the bottleneck of the system represents the event layer. Given that our long-term goal is operational deployment of VAST, we will focus on increasing the peak event processing rate of Broccoli to keep up with a high-volume event stream. We currently experiment with different tuning parameters of the Broccoli that reads from the network socket in rounds,in which each reads with fixed number of bytes.

Our next near-term goal is to implement the query engine. As described, it will support SQL-like queries and allow users to install event feeds. In the long run, we envision to provide graphical user interface to further improve the usability of the system.

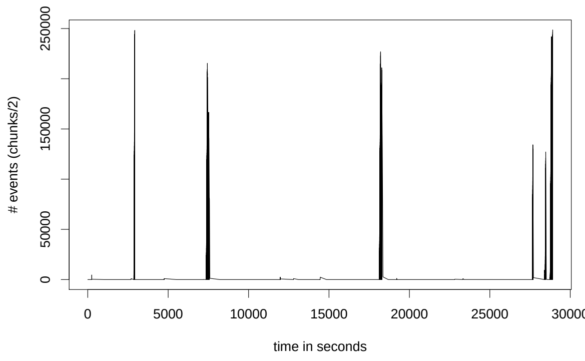
Future work further includes the implementation of an aging and correlation layer which performs continuous housekeeping tasks to optimize space utilization and query response times. The idea is to elevate events to higher semantic abstractions by condensing them into a more succinct form. For example, events reporting numerous unsuccessful connection attempts, could be combined to a summary like "N connections in last T seconds".

To cope with a high-volume event stream over a long period of time, we could use degradation mechanisms that gracefully transform data into more compact but still useful forms. We can have different levels of abstraction. Despite the loss of data over time, abstracted information can constitute the missing link in a chain of previous activities.

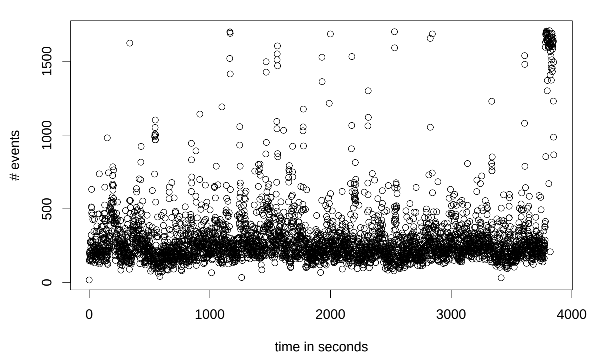
The global nature of network attacks makes them difficult to counter with only a strictly local vantage point. We envision VAST to act as a platform to facilitate op-



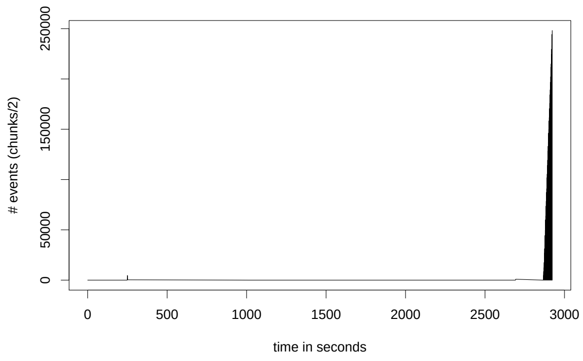
(a) Event processing rate (empty handler).



(b) Bro queue size (empty handler).



(c) Event processing rate.



(d) Bro queue size.

Figure 10: Event processing rate and Bro queue size.

erationally viable, cross-institutional information sharing of attack details. Sharing details in today's operational practices is highly inefficient, cumbersome and requires often human intervention.We can automate significant elements of cross-organizational security analyses via event-oriented analysis scripts that reduce human involvement.

9 Conclusion

Analysis of network activity is fragmented across space and time. There are several fields of application that would significantly benefit from a unified view. We presented the architecture of VAST, an intelligent distributed database that processes network activity logs in a comprehensive and coherent fashion. It accumulates data from disparate sources and provides a central vantage point to facilitate global analysis. We implemented a first prototype that supports archival of events from multiple sources.

The design of VAST is based on Bro's rich-typed event model which provides a generic abstraction for activity. Due to its component-based design, the system can be scaled across multiple machines to distribute the work load. Further, we carefully implemented a concurrent storage engine on top of FastBit to make full use of many-core machines.

Our initial performance analysis showed that the performance of the event layer needs further improvement to handle event rates of operational networks.

Acknowledgements

We would like to thank Robin Sommer for the numerous discussions about the system architecture. Christian Kreibich has also been a great help with his immediate response to the Broccoli issues we raised. Finally, we highly appreciate the invaluable feedback of Vern Paxson on this project.

References

[1] M. Allman, C. Kreibich, V. Paxson, R. Sommer, and N. Weaver. Principles for developing comprehensive network visibility. In *Proceedings of the Workshop on Hot Topics in Security (HotSec)*, July 2008.

[2] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. In *Record of the 1970 ACM SIGFIDET Workshop on Data Description and Access*, pages 107–141, Rice University, Houston, Texas, USA, Nov. 1970. ACM.

[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Asilomar, CA, January 2003.

[4] S. Chandrasekaran and M. J. Franklin. Remembrance of Streams Past: Overload-Sensitive Management of Archived Streams. In *VLDB*, pages 348–359, 2004.

[5] B.-C. Chen, V. Yegneswaran, P. Barford, and R. Ramakrishnan. Toward a query language for network attack data. In *ICDEW '06: Proceedings of the 22nd International Conference on Data Engineering Workshops (ICDEW'06)*, page 28, Washington, DC, USA, 2006. IEEE Computer Society.

[6] E. Cooke, R. Mortier, A. Donnelly, P. Barham, and R. Isaacs. Reclaiming network-wide visibility using ubiquitous endsystem monitors. In *ATEC '06: Proceedings of the annual conference on USENIX '06 Annual Technical Conference*, pages 32–32, Berkeley, CA, USA, 2006. USENIX Association.

[7] P. J. Desnoyers and P. Shenoy. Hyperion: High Volume Stream Archival for Retrospective Querying. In *Proceedings of the 2007 USENIX Annual Technical Conference*, Santa Clara, CA, June 2007.

[8] L. Golab and M. T. Özsu. Issues in data stream management. *SIGMOD Rec.*, 32(2):5–14, 2003.

[9] Google perftools. `http://code.google.com/p/google-perftools/`.

[10] G. Iannaccone, C. Diot, and D. McAuley. The CoMo white paper. Technical Report IRC-TR-04-017, Intel Research, Sept. 2004.

[11] S. Kornexl, V. Paxson, H. Dreger, A. Feldmann, and R. Sommer. Building a time machine for efficient recording and retrieval of high-volume network traffic. In *IMC'05: Proceedings of the Internet Measurement Conference 2005 on Internet Measurement Conference*, pages 23–23, Berkeley, CA, USA, 2005. USENIX Association.

[12] C. Kreibich and R. Sommer. Policy-controlled event management for distributed intrusion detection. In *ICDCSW '05: Proceedings of the Fourth International Workshop on Distributed Event-Based Systems (DEBS) (ICDCSW'05)*, pages 385–391, Washington, DC, USA, 2005. IEEE Computer Society.

[13] libpcap. `http://www.tcpdump.org`.

[14] G. Maier, R. Sommer, H. Dreger, A. Feldmann, V. Paxson, and F. Schneider. Enriching network security analysis with time travel. In *SIGCOMM '08: Proceedings of the 2008 conference on Applications, technologies, architectures, and protocols for computer communications*, New York, NY, USA, August 2008. ACM Press.

[15] S. McCanne and V. Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proc. Winter USENIX Conference*, 1993.

[16] E. O'Neil, P. O'Neil, and K. Wu. Bitmap index design choices and their performance implications. In *IDEAS '07: Proceedings of the 11th International Database Engineering and Applications Symposium*, pages 72–84, Washington, DC, USA, 2007. IEEE Computer Society.

[17] P. E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, London, UK, 1989. Springer-Verlag.

[18] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Computer Networks*, 31(23–24):2435–2463, 1999.

[19] F. Reiss, K. Stockinger, K. Wu, A. Shoshani, and J. M. Hellerstein. Enabling real-time querying of live and historical stream data. In *SSDBM '07: Proceedings of the 19th International Conference on Scientific and Statistical Database Management*, page 28, Washington, DC, USA, 2007. IEEE Computer Society.

[20] M. Siekkinen, E. W. Biersack, G. Urvoy-Keller, V. Gol, and T. Plagemann. Intrabase: Integrated

traffic analysis based on a database management system. In *E2EMON '05: Proceedings of the End-to-End Monitoring Tecques and Services on 2005. Workshop*, pages 32–46, Washington, DC, USA, 2005. IEEE Computer Society.

[21] M. Stonebraker, U. Çetintemel, and S. Zdonik. The 8 requirements of real-time stream processing. *SIGMOD Rec.*, 34(4):42–47, 2005.

[22] K. Wu. Fastbit: an efficient indexing technology for accelerating data-intensive science. *Journal of Physics: Conference Series*, 16:556–560, 2005.

[23] K. Wu, E. Otoo, and A. Shoshani. On the performance of bitmap indices for high cardinality attributes. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 24–35. VLDB Endowment, 2004.

[24] K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, 2006.