Scalanytics: A Declarative Multi-core Platform for Scalable Composable Traffic Analytics

Harjot Gill, Dong Lin, Xianglong Han, Cam Nguyen, Tanveer Gill, Boon Thau Loo
University of Pennsylvania
{gillh,lindong,hanxiang,camng,tgill,boonloo}@cis.upenn.edu

ABSTRACT

This paper presents SCALANYTICS, a declarative platform that supports high-performance application layer analysis of network traffic. SCALANYTICS uses (1) stateful network packet processing techniques for extracting application-layer data from network packets, (2) a declarative rule-based language called ANALOG for compactly specifying analysis pipelines from reusable modules, and (3) a task-stealing architecture for processing network packets at high throughput within these pipelines, by leveraging multi-core processing capabilities in a load-balanced manner without the need for explicit performance profiling. We have developed a prototype of SCALANYTICS that enhances a declarative networking engine with support for ANALOG and various stateful components, integrated with a parallel task-stealing execution model. We evaluate our SCALANYTICS prototype on a wide range of pipelines for analyzing SMTP and SIP traffic, and for detecting malicious traffic flows. Our evaluation on a 16-core machine demonstrate that SCALANYTICS achieves up to 11.4× improvement in throughput compared with the best uniprocessor implementation. Moreover, SCALANYTICS outperforms the Bro intrusion detection system by an order of magnitude when used for analyzing SMTP traffic.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations—*Network monitoring*; D.3.2 [**Programming Languages**]: Language Classifications—*Concurrent, distributed, and parallel languages, Constraint and logic languages, Data-flow languages*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*

Keywords

Applications of parallel and distributed computing; Data-intensive computing; Programming languages and environments

1. INTRODUCTION

As networked applications become increasingly complex and heterogeneous, there is an increasing need for extensibility at the data plane, in order to carry out sophisticated operations such as traffic flow management, traffic filtering, and content-based networking. Central to several of these operations is the ability to perform content-based analysis of network traffic, in order to determine how traffic should be shaped, transformed, or filtered. These analysis tasks are often times compute and memory intensive, since multiple packets need to be buffered and correlated in order to extract application-layer semantics. If deployed at routers, such analysis can potentially slow down packet forwarding, which needs to occur at "line-speed".

In order to speed up packet processing, one promising direction is the use of multicore machines in software-based routers [9, 27]. While these solutions scale as the number of cores/machines increases, they suffer from three limitations: (1) the lack of programming tools to rapidly customize different forms of traffic analysis specific to individual applications, (2) require explicit performance profiling to determine how the packet processing workload should be partitioned across cores, or (3) are not capable of performing complex operations beyond stateless per-packet processing, such as basic IP routing and packet encryption. To address these limitations, we present SCALANYTICS (stands for *Scalable Analytics*), a software-based traffic analysis platform deployed at the network layer, that aims to provide high-performance analysis of packets, capable of extracting and assembling application-layer information from individual packets for analysis.

SCALANYTICS makes the following contributions:

- **Component-based stateful processing.** SCALANYTICS uses a component-based dataflow architecture [11] that allows for rapid assembly of packet processing functionalities from components into *dataflow pipelines* that perform tasks such as assembling packets into application-layer content, operations such as aggregating traffic statistics, or deep-packet inspections into specific application traffic.
- **Declarative configuration language.** SCALANYTICS uses a declarative rule-based language called ANALOG for compactly specifying pipelines that are assembled from existing packet processing components. SCALANYTICS extends prior declarative networking [12] languages with constructs for modularization and components, interoperability with legacy code, and runtime support for parallelism. To illustrate the flexibility of ANALOG, we provide four example pipelines: (1) analyzing SMTP message content using regular expression, (2) analyzing SMTP message content using a machine-learning based spam filter [19], (3) tracking VoIP (SIP [20] protocol) sessions, and (4) detecting Denial of Service (DoS) attack [13], using support vector machines [5] on actual datasets [8]. These pipelines require only 7, 7, 7, and 5 ANALOG rules respectively, and reuse some common components shared across these pipelines.

HPDC'13, June 17–21, 2013, New York, NY, USA.

- **High-performance fine-grained parallelization.** In order to execute these pipelines efficiently, SCALANYTICS provides fine-grained parallelism at the level of individual components within the pipelines. This is achieved through the use of a threading library based on the task-stealing model [18], that achieves both automatic load-balancing and high throughput processing. In addition to stateless per-packet processing (e.g. packet routing, IPSec etc.), SCALANYTICS can analyze the semantics of stateful protocols (e.g. SIP) with high degrees of parallelism. To ensure correct packet ordering in the presence of parallel processing, SCALANYTICS allows packets to be ordered based on its specified *context* attribute, defined in terms of application specific semantics (for instance, SIP session is uniquely identified by its call ID).

- **Implementation and evaluation.** We have developed a prototype of SCALANYTICS, by enhancing an open-source declarative networking engine [17] with the task-stealing model of parallelism and support for various stateful processing components. Using pipelines compactly expressed in ANALOG, we demonstrate the use of SCALANYTICS for performing high-throughput analysis of SMTP, SIP, and detection of malicious flows. In data intensive workloads such as SMTP and SIP, we achieve up to 11.4× speedup on a 16-core machine and a throughput of 1.89 Gbps for analyzing SMTP emails. SCALANYTICS incurs low per-packet latency ranging from 0.1 ms to 2.3 ms. Even under a heavy workload that saturates the system with significant queueing delays, SCALANYTICS's per-packet latency ranges from 0.5 ms to 6.0 ms. We further compared with the Bro Intrusion Detection System (IDS) [14], and observe an order of magnitude performance improvement for SCALANYTICS in analyzing SMTP traffic.

2. DATAFLOW ARCHITECTURE

Figure 1 shows the architecture of SCALANYTICS. The system is deployed at core routers, capturing IP packets for analysis as they arrive. In a typical deployment, SCALANYTICS is used as a non-intrusive network monitoring tool through a network tap. SCALANYTICS can also be deployed inline for making decisions on packet forwarding/filtering and packet transformation.

A network operator provides a ANALOG program that specifies a linear pipeline of components, where each component corresponds to a specific *stage* in the analysis pipelines. Each component can be implemented from scratch or as wrappers over existing libraries. Multiple ANALOG programs corresponding to different analysis pipelines can be installed at the same time. As input to each pipeline, SCALANYTICS accepts events that could either be external, e.g. packet capture from the network, or internal, e.g. local periodic events. The events are queued and scheduled by the *platform thread*, which generates a continuous stream of tuples from the incoming events and inserts them into the pipelines for execution. In this paper, we refer to events processed within the pipeline as *tuples*.

The SCALANYTICS utilizes a *token-based* scheduling mechanism, whereby each incoming tuple is assigned a token number by the *token-dispenser*, and then scheduled for running within the dataflow pipeline. Each pipeline has its own token dispenser. At any time, only a pre-specified number of tokens are allocated for each pipeline, hence limiting the number of in-flight tuples in the pipeline.

Once tuples are assigned a token number, they are then processed within the dataflow pipeline. The first stage in the pipeline is an input component. All components are executed using a *task-stealing* framework (Section 2.2). As input tuples traverse each component at every processing stage, output tuples are generated and buffered for processing in the next component. Based on the ordering semantics of each component, each buffered tuple ready for processing is designated a ready *task*, and enqueued into task queues. Each task queue is assigned to a *task stealing thread* running on a processing core, which dequeues the task from its assigned task queue for processing. In a multicore system, these threads can run in parallel, hence allowing multiple tuples to be processed in parallel within the pipelines. This enables concurrent processing within each component (for different incoming tuples), or processing stages within a pipeline to run in parallel.

In the event of overload due to high traffic load, incoming packets are dropped by the packet capture thread. However, once a packet is accepted into the event queue, SCALANYTICS ensures that this packet will be processed by the system.

At the final stage of the pipeline, output *action tuples* are generated and are used to perform specific actions, including (1) shipping the tuples into another pipeline (at the same or remote node) for further processing, (2) redirecting tuples to a router controller for making traffic management decisions (e.g. rate limit or block a particular flow), (3) materializing into tables as traffic statistics, or (4) raising alarms for the user.

2.1 Dataflow Pipeline

Figure 2 shows an example dataflow pipeline that illustrates the execution model of SCALANYTICS. Each dataflow pipeline consists of several components connected in a linear chain. Briefly, this pipeline receives packets from the network (`Packet Capture`), assembles packets into complete IP packets and TCP segments (`IP Assembler` and `TCP Assembler`, filters out packets related to emails (by recognizing the SMTP protocol from the packet payloads using the `Protocol Detector` component), assembles complete emails (`SMTP Processor`), before finally filtering out emails that match a given regular expression (`Regex Matcher`).

Each SCALANYTICS pipeline can be specified as a ANALOG program (described in Section 3). SCALANYTICS compiles the program into a pipeline, which is installed into the runtime system. To support different forms of serial and parallel execution, SCALANYTICS has three types of components:

- **Serial.** Packets are processed in strict FIFO order. This is typically done for operations where total order is essential. For instance, in our example, `Packet Capture` is a serial component, since the initial stream of packets obtained from the network should be first processed in the order in which they arrive.

- **Parallel.** Incoming packets to a parallel component can be processed by multiple concurrent threads in a manner where ordering does not matter. For instance, once SMTP messages are assembled, regular expression matches on individual email messages can be parallelized, and the order in which these emails are processed is not essential. `IP Assembler` and `Regex Match` are parallel components.

- **Parallel context-ordered.** These components are processed in a partial order. A *context-key* is specified, in which all packets that have the same key should be processed in an order based on their arrival into the system. However, the ordering of packet processing for packets with different keys is not required. For instance, when assembling email messages from TCP segments, messages have to be assembled in partial order (based on TCP flows). Likewise, `TCP Assembler` processes incoming packets

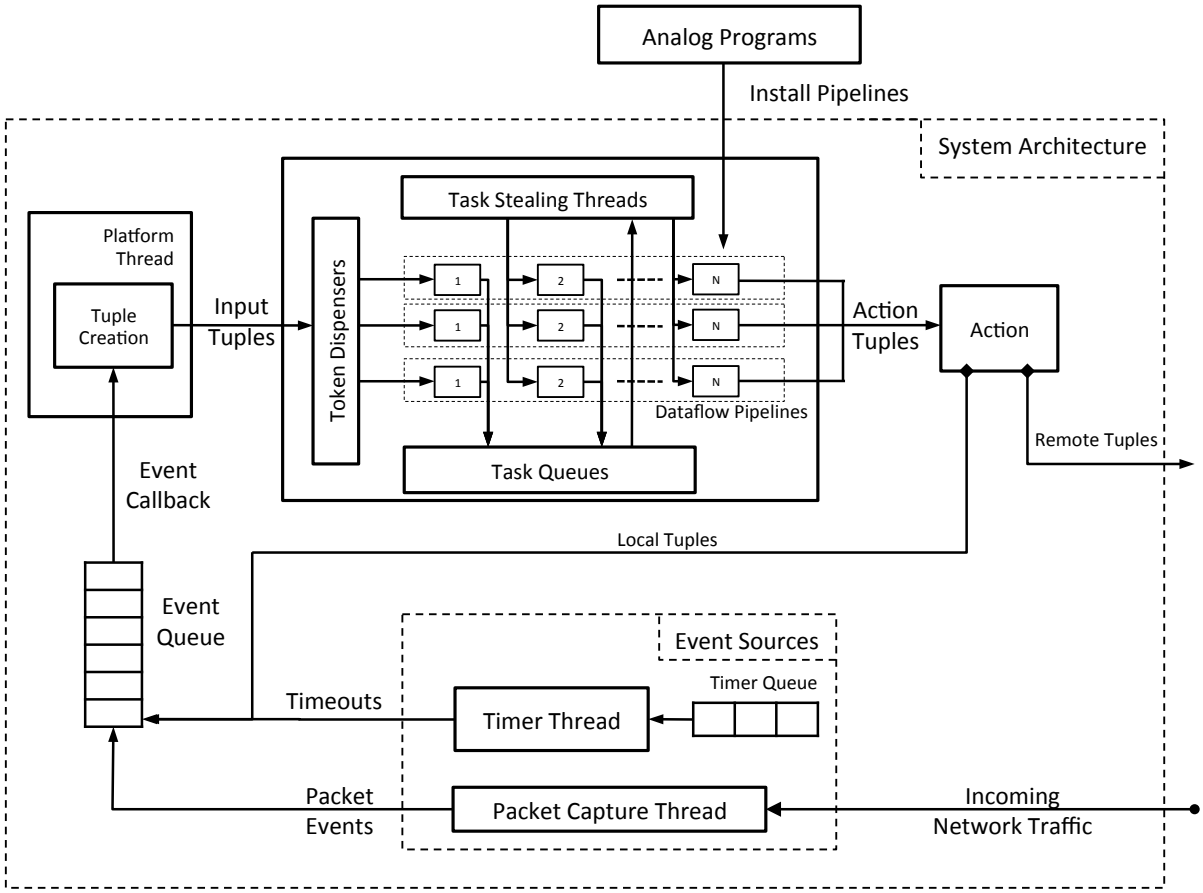

Figure 1: System architecture.

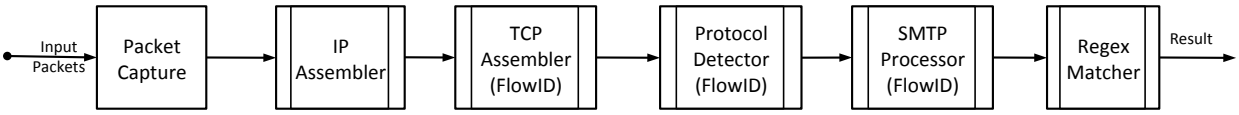

Figure 2: A SCALANYTICS pipeline for SMTP analysis. Parallel components have double lines, and context-ordered component additionally have their context-keys shown in (...).

belonging to the same flow in the order of their arrival, but the ordering of packets across flows is not enforced. `TCP Assembler`, `Protocol Detector`, and `SMTP Processor` are parallel context-ordered components. Note that flow ID is not the only form of context-key supported by SCALANYTICS. As we will later show in our examples, other forms of context-keys are easily supported, e.g. the call ID for a VoIP session.

SMTP pipeline details. Given the above component types, we describe the SMTP pipeline in greater detail. This SMTP pipeline assembles SMTP emails from IP packets, and then filters out emails whose content matches a specified regular expression. The pipeline receives a stream of incoming IP packets that are captured via the `Packet Capture` component. Since the incoming IP packets may be fragmented, they are sent to the `IP Assembler` component for assembling the packets. This component is *stateful*, since previous packet fragments must be kept in-memory at the component until assembled.

After IP assembly, the packets are filtered based on their protocol types (`TCP` or `UDP`). Further processing is done to the TCP stream to assemble TCP segments based on the TCP sequence numbers in the incoming packets. The `TCP Assembler` component is also stateful and *context ordered* based on flow ID. This means that the TCP messages with the same flow ID are processed serially and each unique flow is processed in parallel. The flow ID is extracted from a 4-tuple consisting of source and destination IP addresses and port numbers.

Each assembled TCP flow (consisting of TCP segments in the form of tuples) is then sent to the `Protocol Detector` component, which is stateful and context ordered by flow ID as its context key. The detection component monitors the flow and tries to detect its application layer protocol. For example, the detection component looks for the initial handshake of the SMTP protocol and marks the flow as SMTP if handshake is detected. All the subsequent TCP segments in the SMTP flow would bypass the detector and be marked as SMTP segments.

The detected SMTP segments are sent to the `SMTP Processor` component for message assembly, which is stateful with the flow ID as its context key. The SMTP component tracks the state machine of the active SMTP session. When the data portion of the email is detected, it assembles the complete email message, which may arrive in several TCP segments.

The assembled message is then sent to a parallel regular expression matcher component to look for patterns in the emails. Regex matcher is a stateless component. If there is a match, a positive result is generated. All of the above components are decoupled and run in parallel with respect to each other. This allows flexibility to insert compatible components in between any two components. e.g. an `IP filter` can be inserted between `IP assembler` and `TCP assembler` components to process packets which are destined to specific destination address and port number.

Note that in all of the above components, they can be extracted from existing libraries, rather than be implemented from scratch. We will describe in Section 4 more details on the extraction process. Once extracted into pipeline components, pipeline can be constructed using ANALOG specifications.

2.2 Parallel Execution

SCALANYTICS aims to minimize latency when the system is under light load and maximize the throughput when the system is under heavy load. To achieve this goal, SCALANYTICS uses a task stealing [18, 10] framework to enable adaptive fine-grained parallel processing of individual components within each pipeline.

Figure 3 shows an example SMTP pipeline similar to the one described in Section 2.1. Here, all emails that match the given regular expression are stored in a serial `Logger` component. For ease of exposition, we defer the discussion on context ordered components to Section 2.4, and focus on serial and parallel components here.

Token-based task scheduling. SCALANYTICS utilizes token-based task scheduling mechanism as follows. Each pipeline is assigned its own *token dispenser* component (that is situated just outside the pipeline), as shown in Figure 3. All incoming tuples are first assigned a monotonically increasing token number by its token dispenser component before entering pipeline. Tuples outputted from each component retain the token number of the tuple that triggers its generation. If multiple output tuples are generated from a single input tuple (possibly from multiple rules within one component), these tuples will be processed as a single batch with a common token number. Conversely, if one output tuple is generated from multiple input tuples (a common occurrence when doing application level data assembly), the tuple will be tagged with the token number of the final input tuple that generates this output. In addition, for each input tuple that comes before the final input tuple, a dummy tuple is generated with the corresponding token number of the input tuple. This ensures that the next component sees tokens in increasing order. Hence, token numbers do not disappear. These dummy tokens add minimal processing overhead since each subsequent component simply passes them along.

The token dispenser implements flow control, by allowing only a limited number of in-flight tokens in the pipeline. There is a memory vs. speedup tradeoff in the number of tokens allowed. Having more tokens potentially increases the degree of parallelism, since in-flight tuples can be processed by multiple cores simultaneously. However, this comes at the expense of requiring larger buffer sizes.

Once tuples are assigned tokens, they enter the pipeline for processing. These tuples are processed within components as *tasks*. Each task is hence associated with a component and a set of tuples for execution. Once a tuple is ready for processing within a component, it is inserted into one of the *task queues* for execution. Each task queue is assigned to a task stealing thread, which can process any task on its designated queue. Once a task is executed, any output tuples generated from the component are reinserted into the input buffer for the next component. When these tuples are again ready for execution (for instance, in a serial or parallel context-ordered component, they are next in line for processing based on token numbers), they are placed into a task queue for execution in the next component

Serial components process tuples in strict FIFO order based on token numbers. Since tuples can arrive out of order when a parallel component precedes a serial component, an input buffer is used internally at the beginning of each serial component, to ensure that tuples are processed in-order. This requires each component to maintain the token number for the last tuple processed (*lastToken*), and only allow the next tuple whose token number is one larger than *lastToken* to be processed. Once a tuple is ready for processing within a component (i.e. reaches the head of the input queue), it is then inserted into one of the *task queues* for execution. On the other hand, since *Parallel* components have no ordering constraints, they can process tuples in completely parallel fashion. As a result, no input buffer is necessary. Tuples arriving at the parallel component are directly inserted into task queue for execution and are never deferred.

2.3 Task Stealing

SCALANYTICS provides dynamic load balancing through the use of a task stealing architecture [18], also known as *work stealing* [10]. A task stealing framework achieves parallelism by using multiple cores at the same time. Each core on a multicore machine can run one or more task stealing thread with hyper-threading. Each task stealing thread is assigned a task queue, where tasks can be dequeued in any order by the thread for processing.

In SCALANYTICS, a task is denoted by the processing of a tuple (or group of tuples with the same token number) within a component. Whenever a tuple is ready for processing, it is inserted into one of the task queues. By default, the task is inserted into the task queue of the currently executing thread, in order to minimize CPU cache misses. A thread that has completed a task can then dequeue from its corresponding task queue. However, if a thread is idle and has no outstanding tasks, it can *steal* tasks designated for other threads that are currently busy, hence achieving dynamic load balancing. This is unlike other prior approaches used in scaling software-based routers, which either utilize a static partitioning approach [9], or require pre-creation of multiple component instances based on performance profiling [27].

In SCALANYTICS, task stealing is particularly useful when not all flows are arriving at the same rate. For instance, when the rate of SIP messages is higher than SMTP messages, SCALANYTICS's use of task stealing would naturally adapt to use more threads for processing SIP components. A task stealing thread (referred to as thread below) is responsible for executing tasks (in this case, tuples or sets of tuples designated for each component to be processed). Each thread attempts to steal tasks from another randomly chosen thread (the victim) when there are no pending tasks in its local task queue. An unsuccessful attempt to steal, in case of an empty victim task queue, leads the thread to back off for a predetermined amount of time and try again. A point to be noted is that components do not map to any particular thread. However, SCALANYTICS optimizes CPU cache locality, by using the same thread whenever possible to process the same tuple across components. This vastly reduces the cache-misses (associated with transfer/stealing of a task to another core) as most tuples would be recycled into being processed by the next component on the same core. A task is transferred only when

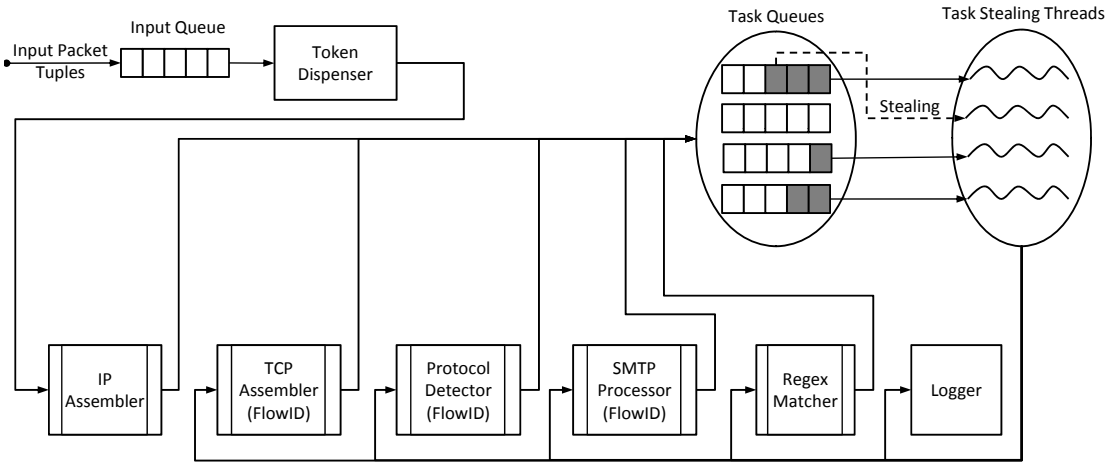

Figure 3: SCALANYTICS's pipeline architecture showing task stealing in action.

a thread is busy processing a component and some idle thread is able to steal that task.

2.4 Enforcing Context-Order

Figure 4 shows the internals of a parallel context-ordered component. Unlike an unordered parallel component, all tuples within this component have to be processed in a partially ordered fashion based on user-specified context key(s). The context keys can be specified when defining each component in ANALOG (Section 3).

An *Input Buffer* is required here, in order to process event tuples in FIFO order based on their token numbers. A key difference here compared with the serial component is the use of *context keys*, where tuples from the *Input Buffer* are further divided into sub-buffers, one for each key. To ensure correct ordering within each sub-buffer, a *Context Designator* is used to classify incoming tuples based on their context keys, and then insert them into their respective sub-buffers. Note that a tuple may have multiple context keys, in which case, it will be placed in multiple buffers, and dequeued for processing when it reaches the front in all its buffers. Once a tuple is ready for processing, it will be inserted into the task queue, at which point, its corresponding ANALOG execution rules becomes eligible for execution by one of the task stealing threads.

In order to ensure that outgoing tuples (corresponding to the same context key) are processed in order of their token numbers, at any point in time, only the topmost outgoing tuple from each sub-buffer is inserted into task queues. Upon executing this tuple, the next tuple from the same sub-buffer is inserted into the task queues for execution.

3. ANALOG LANGUAGE

Each pipeline component in SCALANYTICS is specified using the ANALOG language. We extend this language based on prior work on declarative networking [12]. Declarative networking enables specification of networking protocols as a set of queries in a high-level language, primarily based on Datalog [16]. A Datalog program is a set of *rules*, which are of the form `q :- p1, p2, ...., pn`. Here, `q` is the *head* of the rule and `p1, p2, ...., pn` is a set of *literals* that constitute the *body* of the rule. Literals are either *predicates* (e.g. relations or streaming tuples) with *attributes* or boolean expressions that involve function symbols (e.g. arithmetic) applied to attributes. A Datalog program is said to be recursive if a *cycle* exists through any predicate, i.e. predicate appearing in a rule's body appears in the head of the same rule. Body predicates are evaluated in a left-to-right order. Like prior declarative networking languages, we adopt the use of a location specifier attribute `@` that is used to denote the location (physical network address) of each tuple. Though not an emphasis in this paper, location specifiers will make distributed extensions natural to realize in future, hence further improving the scalability of our system.

3.1 Components

Our ANALOG language extends traditional Datalog by allowing the specification of a processing pipeline as a linear chain of components. These components are executed in parallel to each other by the run-time, thus exhibiting pipelined parallelism. A component is essentially a set of rules, and its syntax is as follows:

```
component comp-name(type, context_keys(pA1(k1, ..., kn), ...,
                                        pNn(k1, ...,kn))) {
 rA qA :- pA1, pA2, ..., pAn.
 rB qB :- pB1, pB2, ..., pBn.
 ..
}
```

The component specification is labeled by the keyword `component`, followed by the component name, `comp-name`. In the component specification, `type` can be `serial`, `parallel` or `parallel_context_ordered`. `context_keys(...)` specifies the *context-keys* for parallel context ordered components.

To ensure deterministic behavior, we constrain the ANALOG language specification to only allow pipelines that can be represented as a linear chain of components. This technique is logically similar to *stratification* restrictions (e.g. on the use of negation operator) in Datalog [15], where the next component is not executed until the previous component has completed its execution.

Within each component, a set of rules is executed to a *fixpoint*, upon the arrival of an input tuple that triggers the execution of the component. Typically, a component is triggered for execution only if the incoming tuple matches the body predicates in one of the rules. The execution of one of these rules may generate new tuples that will trigger other rules within a component. A fixpoint is reached when no new facts are derived locally, at which point, all output tuples are batched, tagged with the token number of the initial input tuple, and sent to the next component for further processing. If a newly derived or an input tuple does not invoke any

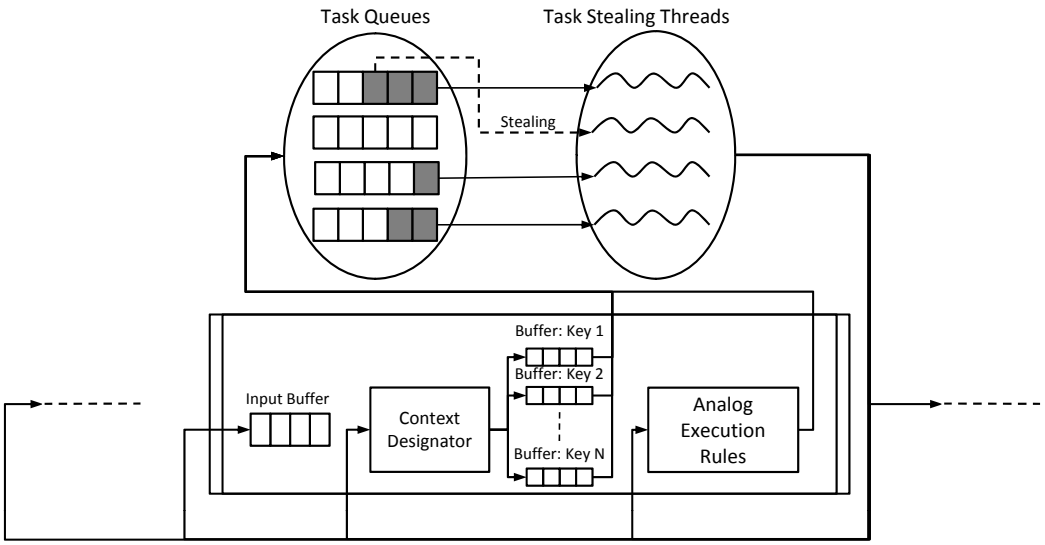

Figure 4: Parallel context-ordered component.

rule within a component, it is simply used as an output tuple of the component without any rule executions.

3.2 Example Pipeline in ANALOG

To better illustrate ANALOG language, we provide an example program of three components that corresponds to the pipeline in Figure 2.The compete ANALOG program is shown in Appendix A. In our example, ANALOG is used as a wrapper for gluing together various components. Within each component, user-defined functions are used to invoke external libraries implementing various processing capabilities (Section 4). These components are connected into a linear chain, since the output of one component is used as input to the next component.

```
component tcp_protocol_detector(parallel_context_order,
          context_keys(tcp_stream(5))) {
det1 detected_tcp_protocol(@N, PKT, SRC_IP, DST_IP,
                           FLOW_ID, TCP_MESSAGE, PROTOCOL) :-
        tcp_stream(@N, PKT, SRC_IP,
                   DST_IP, FLOW_ID, TCP_MESSAGE),
        PROTOCOL = f_processTCPDetector(TCP_MESSAGE, FLOW_ID).

det2 smtp_segment(@N, PKT, SRC_IP, DST_IP,
                  FLOW_ID, TCP_MESSAGE) :-
        detected_tcp_protocol(@N, PKT, SRC_IP, DST_IP,
                              FLOW_ID, TCP_MESSAGE, PROTOCOL),
        PROTOCOL = "SMTP".
}

component smtp_processor(parallel_context_order,
          context_keys(smtp_segment(5))) {
smtp1 smtp_mail(@N, PKT, SRC_IP, DST_IP, FLOW_ID, MAIL) :-
        smtp_segment(@N, PKT, SRC_IP, DST_IP,
                     FLOW_ID, TCP_MESSAGE),
        MAIL = f_processSMTPSegment(TCP_MESSAGE, FLOW_ID),
        MAIL != "".
}

component regex_match(parallel) {
regex1 positive_match(@N, PKT, SRC_IP, DST_IP,
                      FLOW_ID, MAIL, MATCH) :-
        smtp_mail(@N, PKT, SRC_IP, DST_IP, FLOW_ID, MAIL),
        MATCH = f_match(MAIL, "hello"),
        MATCH = "true".
}
```

`tcp_protocol_detector` and `smtp_processor` are parallel context-ordered components, since they are both declared to be of type `parallel_context_order`. The `context_keys` parameter is only applicable to parallel context-ordered components. Both components are parameterized by `context_keys(tcp_stream(5))` and `context_keys(smtp_segment(5))` , which means that they use the 5^{th} attribute (`FLOW_ID`)[1] of their respective `tcp_stream` and `smtp_segment` input tuples as their context key.

Hence, tuples carrying packets (`PKT`) with different `FLOW_ID` are allowed to be assembled and processed in parallel. Component `regex_match` on the other hand is a `parallel` component. The `tcp_protocol_detector` component is triggered upon the arrival of an input `tcp_stream` tuple. Each input tuple carries the assembled TCP payload as its `TCP_MESSAGE` attribute as well as the `FLOW_ID` attribute. Upon its arrival, rule `det1` is triggered, which then results in the invocation of a protocol detection module implemented by the `f_processTCPDetector` function call (which returns a `PROTOCOL` type, e.g. "SMTP" or "SIP").

The firing of `det1` results in the generation of a `detected_tcp_protocol` tuple, which is then used to execute rule `det2`. The rule `det2` checks if the `PROTOCOL` is "SMTP" and generates `smtp_segment` tuple if true. Since the resulting `smtp_segment` tuple does not occur in any rule body, a fixpoint is reached, and this tuple is added to the list of output tuples to be sent to the next component. When there are no tuples pending evaluation in any rules within the component, a local fixpoint is reached, and all outgoing tuples are batched and sent to the next component in the pipeline.

The output tuple `tcp_segment` is then used as input to the next component, which in this case, is the `smtp_processor` component (rule `smtp1`). The `smtp_processor` component generates a `smtp_mail` tuple if a mail is successfully assembled. This is then used as input to the `regex_match` component, which generates a `positive_match` tuple if `MAIL` matches the given regular expression string.

[1]The `FLOW_ID` attribute is a concatenation of the source and destination IP addresses and port numbers, and used to uniquely identify a TCP or SMTP session.

4. IMPLEMENTATION AND USE CASES

We have developed a prototype of SCALANYTICS using the open-source RapidNet declarative networking engine [17], which provides support for Click-like dataflows [11] and a Datalog compiler. We have enhanced the Datalog compiler to support the ANALOG language, and implemented several dataflow pipelines in SCALANYTICS from ANALOG specifications.

We enhanced RapidNet's runtime engine to support the new architecture described in Section 2. The new execution model is designed and implemented to enable local fixed-point computation (i.e. rules are executed till no new facts are derived) within a component before the derived tuples are batched together and sent to the next component in pipeline. The runtime engine is enhanced through task stealing framework (i.e. the task queues and task stealing threads) described in Section 2.2 using Intel's Threading Building Blocks (TBB) library [18]. Using TBB as a basis, we are able to implement SCALANYTICS's serial and parallel components. To enable parallel context-ordered components, we implemented a new feature called *context-filter*. Briefly, the context-filter allows us to create sub-buffers (Figure 4) keyed using context key. This ensures strict ordering of tuples that share the same context key but allows parallel processing of tuples across multiple contexts.

In the rest of this section, we briefly present four example dataflow pipelines that highlight different uses of SCALANYTICS. In all cases, these pipelines are assembled from reusable components that are implemented from existing legacy code.

4.1 SMTP Analysis Pipeline

In our first use case, we implemented the SMTP analysis pipeline as shown in Figure 2. The detailed specification of the pipeline in ANALOG is in Appendix A. The entire pipeline requires only 7 rules in 6 components (including the initial `Packet Capture`). The `IP Assembler` and `TCP Assembler` components are adapted from libnids [26] source code, which in itself is derived from the Linux kernel's TCP/IP stack code.

For the purpose of this project, libnids code is unusable directly as it is designed for single threaded use. The libnid's IP and TCP stacks are decoupled from each other and made thread-safe by adding locks to their internal tables. The locks are added to protect the lookup and insertion of stateful structures in hash tables (e.g. flow table). The context-filter extensions to TBB guarantees safe access to these structures based on TCP flow ID, and thus no fine-grained locks are added. Also, we have made modifications to hashing scheme used by TCP stack so as to map client to server and server to client packets to the same hash bucket. This change allows us to use a single, unique flow ID for identifying TCP packets flowing in either direction. Our changes enable libnids's IP stack to run in completely parallel and the TCP stack to run in parallel context-ordered mode. It takes us just a couple of days to make these changes to the code.

The `SMTP Processor` component is adapted from the SMTP analysis code from Bro IDS [14, 1]. It outputs each assembled email on observing end of DATA for each SMTP session. The assembled email will be used by `Regex Matcher` to search for the given regular expression.

All in all, it takes us a few days to port existing legacy implementations into components usable by our SMTP analysis pipeline. Note that this is a one-time effort, as these components are reusable for other pipelines.

4.2 SMTP Spam Detector Pipeline

In our second use case, we improved upon the basic SMTP analysis pipeline shown above. Rather than do a generic regular expression match on the email body, we instead added a spam detector module that allows us to filter out unwanted emails. Figure 5 shows the entire pipeline. We reuse earlier components for assembling SMTP messages from incoming traffic. The assembled emails are then classified into spam or regular emails, using a `Naïve Bayes Text Classifier` component. This pipeline requires only 7 rules in 6 components.

Naïve Bayes classification is a well-known machine learning algorithm for performing efficient text classification [19]. The classifier is first trained offline, in our case, using pre-existing emails that have previously been identified as spams. Our `Naïve Bayes Text Classifier` is a generic implementation of the naïve Bayes classification algorithm tailored for text input. The component first tokenizes each incoming text document into individual words. Each input word is then tagged with a number indicating its probability of being a keyword for spam data. This probability is derived from the offline training phase. The classifier then computes a final value from all the input word probabilities, to make a final decision on whether the input text is classified as spam.

Note that this component is implemented in a generic fashion, and can be reused for any other forms of text classification that uses the naïve Bayes classification technique. It is also embarrassingly parallel, meaning that one can implement the classifier as a parallel component within SCALANYTICS.

4.3 SIP Analysis Pipeline

Our third use case is a pipeline that performs VoIP traffic analysis. Figure 6 shows a pipeline that implements an interception of VoIP traffic by monitoring setup and tear-down of VoIP calls based on Session Initiation Protocol. This pipeline reuses the `IP Assembler` component used in our previous example. The entire pipeline requires 7 rules in 5 components. The detailed ANALOG pipeline specification is provided in Appendix B.

In the pipeline, SIP protocol processing is composed of `SIP Parser` and `SIP Transaction Processor` components. The `SIP Parser` component parses the header of each SIP call to extract out relevant information from the call, such as the CallID. This information is then used by the `SIP Transaction Processor` component to track the state of the SIP call. This is a stateful component, since it needs to maintain the state machine corresponding to the SIP session. These two components are extracted from existing implementations of the GNU oSIP library [2], which provides interface for controlling SIP based sessions.

Once the setup or tear-down of a SIP session is detected, this information can be used by other components for further processing. For example, we can add a filter (either by CallID or user) to identify VoIP data streams of interests, and then record the VoIP data stream of these selected calls using a RTP stream interceptor component.

4.4 DoS Attack Detection Pipeline

Our last use case implements a Denial of Service (DoS) attack detection [13] using support vector machines (SVM) [5], a popular algorithm used in machine learning for data classification. Figure 7 shows the pipeline, which reuses earlier components for packet capture, IP and TCP assembly. The `Feature Extractor` component extracts out features for each assembled TCP flow, which are then used by the `SVM Classifier` to detect potentially malicious flows that exhibit DoS behavior. The complete ANALOG specification (in Appendix C) contains 5 rules in 4 components.

The `SVM Classifier` is implemented using libsvm [5], a publicly available SVM implementation. It is written as a component that can be used for other types of traffic analysis beyond our use case. The classifier is first trained using existing traces, a subset of which are tagged as malicious. The classifier depends on fea-

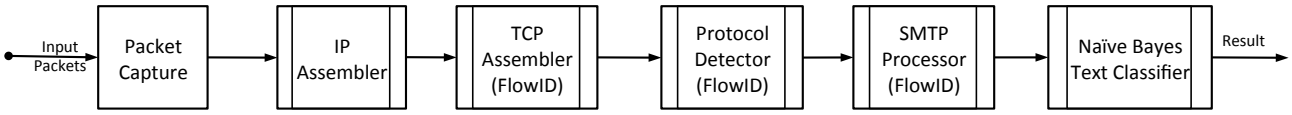

Figure 5: SMTP Spam Detector pipeline

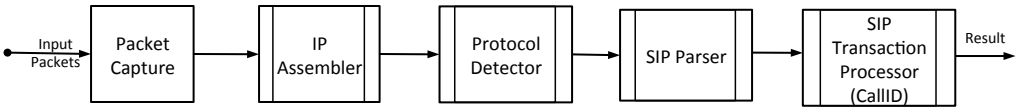

Figure 6: SIP call traffic interception pipeline. Tasks queues and task stealing threads are omitted for brevity.

tures extracted from each TCP flow for determining whether the flow is malicious. This is achieved using the `Feature Extractor` component that maintains features useful for separating malicious from normal TCP flows, such as connection duration, bytes/packets transferred, number of connections in the recent 1 minute to this same port number or IP address. Note that the last feature requires maintaining a global ordered list of TCP connections that is updated by all threads, which potentially limits the scalability of `Feature Extractor` component. We will revisit this issue in 5.2

5. EVALUATION

We perform an experimental evaluation to study SCALANYTICS's performance characteristics in extracting fine-grained parallelism. We also validate its capability to correctly implement the packet processing functionality specified in the ANALOG programs.

Our experiments are carried out on a HP ProLiant BL420c machine, that has dual-socket processors. Each processor is an Intel Xeon E5-2450 with 8 cores. In total, the machine has 16 cores, and with hyper-threading support on each core. Each core has a 2 MB L2 cache, and each processor includes a 20 MB L3 cache. The total memory size is 24 GB.

5.1 Experimental Setup

Given 16 cores and support for hyper-threading, all experiments are carried out with up to 32 threads. All experimental results are averaged across 5 runs. Our experiments consist of benchmarking the 4 pipelines described in Section 4.

- **SMTP analysis (Section 4.1)**. As input to this pipeline, we use a packet trace that contains transmission traffic of 50,000 emails with an average size of 150 KB. In this experiment, we assemble and match emails with a simple regular expression, and validate that SCALANYTICS is able to filter out all such emails. This experiment allows us to explore system's throughput when the workload is composed of large packets. (SMTP)

- **SMTP Spam Detector (Section 4.2)**. In this experiment, the input data contains of SMTP traffic traces generated by replaying emails from the *SpamAssassin* dataset [23]. This dataset includes 2155 (3160) emails tagged as spam (non-spam). Each email averages 6.7KB in size. By comparing directly to the original tagged dataset, we observe that our spam detector achieves high accuracy with only 0.3% false positive rate and 4.7% false negative rate in classifying SpamAssassin dataset. (SPAM)

- **SIP analysis (Section 4.3)**. As input, we use a trace file that contains 100,000 SIP sessions, which is generated using SIPp traffic generator [21]. Our analysis pipeline is able to correctly recognize the setup and tear-down of all SIP sessions, decode and record related information, including duration and URI of each session. (SIP)

- **DoS attack detection (Section 4.4)**. Our final experiment involves the DoS attack traffic. The SVM model used in the pipeline is trained offline using DARPA intrusion detection dataset [8], which contains traces that have been tagged with one of 155 different DoS attacks. We use 5 weeks of traffic traces consisting of 825K TCP connections for training purposes. To validate the model and pipeline, we instantiate the model into the `SVM Classifier` component, and inject 2 weeks of traffic traces into the pipeline. The 2 weeks of traces contain 810K connections, of which 371K are tagged as malicious. Our DoS detection pipeline achieves 95.85% accuracy on 371 thousand attack connections and 99.75% accuracy on 439 thousand non-attack connections. Note the false-positive rate is only 0.25%, which makes the system very useful for deployment in practice. (DOS)

For each pipeline, we execute two load scenarios. Under the *heavy* load scenario, we replay a PCAP file to generate the maximum amount of traffic that can be handled by our pipelines. This saturates our pipelines, maximizes the queueing delays between components, and allows us to explore the limits of our system in terms of its throughput and speedup. Under the *light* load scenario, we rate limit our input traces at 5 Mbps. This allows us to measure response time when the system is not experiencing heavy queueing delays. The response time is hence dominated by just the processing overhead of each component.

In all our experiments, we set the maximum number of in-flight tokens for each pipeline to 1000. We observe that the throughput does not increase significantly beyond 1000 tokens. In addition, the system is able to maintain a reasonably small resident memory footprint.

5.2 Speedup and Throughput

Figure 8 shows the speedup, and Figures 9-12 show the throughput achieved for all pipelines, as the number of threads increases from 1 to 32. Here, we subject all pipelines to the heavy load scenario. The speedup is then compared with the best uniprocessor implementation of each dataflow pipeline, without the added overhead of TBBs. The error bars in Figures 9-12 show the standard deviation obtained across 5 experimental runs.

We make the following observations. First, as the number of threads increases (and hence more cores are utilized), the maximum speedup ranges from 9.0×-11.4× for *SMTP*, *SPAM*, and *SIP*. We observe that linear speedup is not achievable, given that our pipelines have serial as well as parallel context-ordered components, and hence are not embarrassingly parallel. Nevertheless, SCALANYTICS provides a significant speedup over the uniprocessor performance.

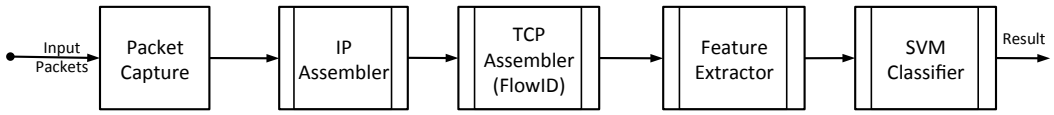

Figure 7: DoS attack detection pipeline

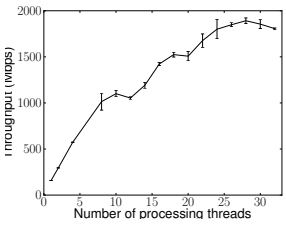

Figure 9: SMTP Analysis Throughput (Mbps).

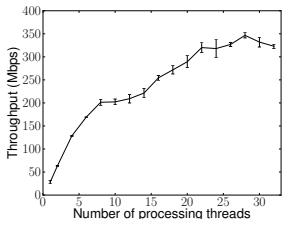

Figure 10: Spam Filter Throughput (Mbps).

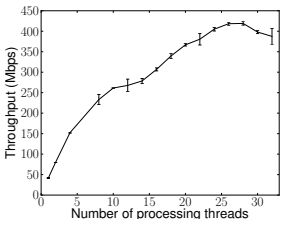

Figure 11: SIP Analysis Throughput (Mbps).

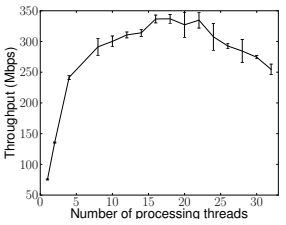

Figure 12: DoS Attack Detection Throughput (Mbps).

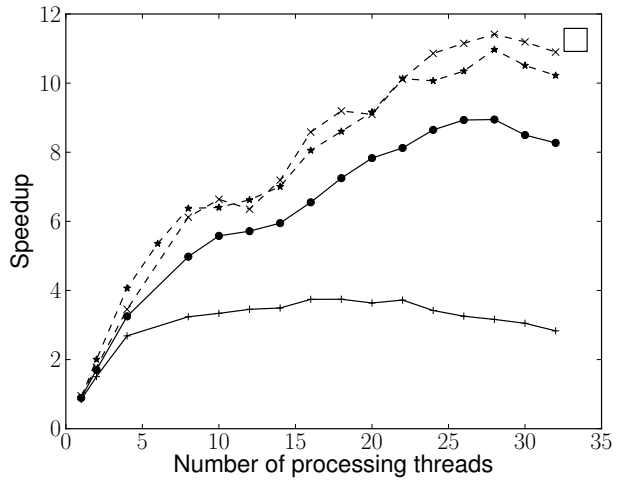

Figure 8: Speedup

The maximum speedup (3.8×) obtained by *DoS* pales in comparison with that of other pipelines. This is because our `Feature Extractor` component in the *DoS* pipeline maintains a global list of all TCP connections within a specified time window. This list is continuously updated for each incoming packet, hence slowing down the entire pipeline.

Table 1 shows the performance statistics for each pipeline execution, under the setting where throughput is maximized for a given number of threads. We observe that the throughput of *SMTP* is as high as 1.89 Gbps (1474 emails per second). The resident memory footprint across all pipelines is below 310 MB, well within the physical memory capacity of the machine. We further observe that dynamic load balancing via task stealing is actively used by our system. On average, between 15.4% and 35.5% of tasks are stolen by idle threads.

To explore the limits of our system, we further experiment with larger number of threads beyond 16 (number of cores). Our Intel processor supports *hyper-threading*, in which two simultaneous execution threads can be maintained by each core sharing the same execution resources. This allows two threads to essentially context-switch "for free" without involving any heavy-weight kernel operations. In principle, this allows us to parallelize up to 32 threads. We observe that hyper-threading is indeed useful at increasing system throughput in all of our experiments, in particularly the SMTP pipelines that are data-intensive. Pipelines such as *SMTP* benefit more from hyper-threading, given that there is a larger likelihood of L2 cache misses due to the larger payloads and emails assembled. For example, at 28 threads, using the Intel VTune [3] profiler, we observe that on average, the *SMTP* workload incurs 2×-2.5× the number of L2 cache misses as compared to the other pipelines that are processing smaller payloads.

When a L2 cache miss occurs, hyper-threading allows another thread to execute a new task, while the previously executing thread yields the processor to perform out-of-core memory operations. However, beyond 28 threads, we observe that the use of hyper-threading actually backfires due to spin lock contention on the TBB queues by threads running on the same core. Note that such TBB locking contention is not an issue when the number of threads is the same as the number of cores, given that each TBB task stealer owns its own processing core.

Bro comparison. As a basis for comparison, we compare our SMTP analysis throughput against that of the Bro IDS [14]. This allows us to sanity check that SCALANYTICS indeed can outperform an existing Bro deployment on a single core. We use Bro 2.0 [1], and its existing protocol decoder and SMTP module. We configure Bro to use its existing regular expression pattern search operation over the emails. The pattern search operation is similar to our SMTP pipeline. We observe that for the SMTP analysis pipeline, Bro achieves a throughput of 140 Mbps (109 emails/s). Compared to our highest throughput numbers (*SMTP*) in Table 1, SCALANYTICS shows a 13.5× improvements in throughput over Bro. We note that while one can naively scale Bro by running multiple independent instances on a single machine, this approach does not easily support stateful analysis for any of our four applications, since these Bro instances would not be coordinated.

5.3 Latency

Table 2 shows the average per-tuple latency for all pipelines. In addition to the heavy load, we also contrast each pipeline execution to a light load scenario, as described in our experimental setup (Section 5.1). In the table, *Latency* shows the per-tuple latency (in ms), averaged across all tuples that traverse each pipeline.

We further broken down this latency number into the follow-

Pipeline	Threads	Speedup	Throughput	Packets/s	Memory	% TBB steal
SMTP	28	11.4	1892 Mbps (1474 emails/s)	6.2×10^4	310MB	15.4%
SPAM	28	11.0	347 Mbps (4842 emails/s)	1.2×10^5	255MB	20.2%
SIP	28	9.0	419 Mbps (19200 sessions/s)	1.2×10^5	204MB	24.8%
DoS	18	3.8	337 Mbps (1632 flows/s)	1.7×10^6	195MB	35.5%

Table 1: Summary of performance statistics for each pipeline execution with the number of threads where throughput is maximized.

Pipeline	Workload	Latency (ms)	Queueing time (ms)	Processing time (ms)
SMTP	Heavy	6.036	5.227	0.809
	Light	2.338	1.398	0.940
SPAM	Heavy	5.000	3.320	1.680
	Light	1.960	0.275	1.685
SIP	Heavy	0.451	0.246	0.205
	Light	0.236	0.022	0.214
DoS	Heavy	0.520	0.425	0.095
	Light	0.114	0.018	0.096

Table 2: Average per-tuple latency (ms) for each pipeline under the heavy and light load scenarios.

ing components: (1) *Queueing* corresponds to the time each tuple spends in the input buffer of each component; and (2) *Processing* is the actual time spent by each tuple within each component execution.

We observe that under the heavy load scenario, when SCALANYTICS utilizes all cores at or close to 100% for maximum throughput, each tuple incurs on average 0.5 to 6 ms latency across all pipelines. As expected, queueing time dominates the overall latency. Under the light load scenario where queueing time is not a factor, per-tuple latency is modest, requiring only 0.1 to 2.3 ms. Each component typically requires only a fraction of a millisecond to complete execution, demonstrating the overall efficiency of SCALANYTICS. We further observe that for pipelines that are less data intensive (such as SIP and DoS), the per-tuple latency is below 0.5 ms for both heavy and light load scenario.

.

5.4 Summary of Results

Overall, our SCALANYTICS system is able to implement the SMTP, SPAM, SIP and DoS detection pipelines efficiently. Moreover, SCALANYTICS is able to leverage up to 16 processing cores to achieve speedup ranging from 9.0× to 11.4×. Large emails can be processed within 6 ms even under heavy load, at a rate of 1474 emails per second (or 1.89 Gbps) on an inexpensive commodity multi-core machine. Our SMTP analysis throughput outperforms the Bro IDS by up to 13.5×. Spam filter assembles and classifies incoming emails at a rate of 4842 emails per second (or 347 Mbps). SIP traffic is similarly identified and analyzed at a rate of 19,200 SIP calls per second. In the DoS dataflow pipeline, 1632 TCP flows are analyzed per second, and our SVM component exhibits high accuracy on the DARPA dataset. In all our experiments, under the light load scenarios, the per-tuple latency within the pipeline is less than 2.3 ms, and the resident memory footprint of SCALANYTICS is modest (reaching 310 MB for *SMTP*).

6. RELATED WORK

The component-based framework that SCALANYTICS uses is similar to that of the approach taken by Click [11]. We have enhanced Click's dataflow framework to incorporate support for fine-grained parallelism. Our ANALOG language has its root in declarative networking [12], which aims to provide a compact domain specific language for implementing network routing protocols. We have extended the original declarative networking language with constructs for specifying and gluing components, as well as invoking existing libraries through the use of user-defined functions. Moreover, our task-stealing framework can be used for executing Datalog programs in parallel, hence exploiting multicore processing capabilities.

Multi-core software router platforms such as RouteBricks [9] similarly leverage multiple processing cores within a cluster environment to scale up packet processing. However, these systems are typically geared towards stateless processing, such as packet forwarding, packet classification, IPSec etc. at the network layer, and are ill-suited for sophisticated workloads such as SPAM and SIP analysis. *Pipeline-parallelism* approaches taken by [27, 6] provide fine-grained parallelism when processing a packet. However, these approaches require run-time system profiling and periodic load-balancing across cores, which is expensive and also susceptible to inaccuracy. Through our use of a task-stealing approach, we are able to achieve dynamic load-balancing and fine-grained parallelism without requiring explicit runtime profiling.

Prior work on multi-core Snort [7] are unable to support stateful processing. Hence, these systems would not be able to support any of our use cases. Gnort [24] uses GPUs for parallelizing packet analysis. However, their approach is only suited for embarrassingly parallel applications, e.g. pattern matching, and is not suited for processing stateful flows. Moreover, the restricted execution model of GPUs (which requires all cores to run the same instructions under the SIMD architecture) limits the flexibility of their system in allowing different cores to perform different processing tasks simultaneously. A recent work[25] uses a hybrid CPU/GPU approach, where GPUs are again limited to embarrassingly parallel pattern matching, while stateful operations are serialized on CPUs on a per-flow basis. This approach lacks the flexibility provided by SCALANYTICS, where pipelines involving a combination of serial, embarrassingly parallel, and parallel context-ordered components can be easily specified. The underlying system automatically load-balances and parallelizes the pipelines, while respecting the semantics of different component types.

Sommer *et al.* [22] proposed a parallelization strategy for Bro. However, the evaluation of this system appears preliminary: most of the results are presented in simulation, and the actual evaluation is smaller in scale compared to our paper (both in scale and use cases). In terms of the execution model, Sommer et al. does a static partitioning of flows across threads, an approach that may result in load imbalance across cores. Scalanytics avoids such load imbalance issues through the use of the task stealing framework, where each task represents an execution instance of a tuple within a component

A key contribution of SCALANYTICS is to provide a programming platform that allows us to assemble component modules out of existing packet analysis platforms for parallel execution, as we have demonstrated in Section 4. We note that intrusion detection systems (IDS) such as Bro and Snort achieves only a subset of the functionality provided by Scalanytics. SCALANYTICS enables traffic analytics to be customized and parallelized, and has applications beyond IDS, for instance, allowing machine-learning based classification of application-layer content at the network layer, performing spam email detection, measuring VoIP usage statistics so as to customize traffic shaping policies, etc.

7. CONCLUSION

This paper presents the design and implementation of SCALANYTICS, a scalable packet processing platform that supports (1) stateful application-layer traffic analytics, (2) high degrees of configuration through reusable components in a dataflow framework and a high-level declarative configuration language, and (3) parallelism and automatic load-balancing through the use of a task-stealing framework integrated into a declarative networking engine. We have developed a prototype of SCALANYTICS, and our evaluations demonstrate its scalability as well as functionality. As ongoing work, we are enhancing our language and runtime system to enable directed acyclic graphs beyond linear pipelines. We are integrating our system with existing software-defined networking platforms such as OpenFlow [4], so that the analysis output from SCALANYTICS can be used to actuate flows in the network. We are also working towards a distributed implementation that allows pipelines to be constructed across components running on different machines.

Acknowledgements. This project is supported in part by NSF grants IIS-0812270, CAREER CNS-0845552, DARPA SAFER award N66001-C-4020, and a AFOSR Young Investigator Award FA9550-12-1-0327.

8. REFERENCES

[1] Bro Intrusion Detection System. `http://bro-ids.org`.

[2] GNU oSIP library. `http://www.gnu.org/software/osip`.

[3] Intel VTune Amplifier XE 2013. `http://software.intel.com/en-us/intel-vtune-amplifier-xe/`.

[4] OpenFlow. `http://www.openflow.org/`.

[5] CHANG, C.-C., AND LIN, C.-J. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology 2* (2011), 27:1–27:27. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[6] CHEN, B., AND MORRIS, R. Flexible control of parallelism in a multiprocessor pc router. In *USENIX ATC* (2001).

[7] CHEN, X., WU, Y., XU, L., XUE, Y., AND LI, J. Para-snort: A multi-thread snort on multi-core ia platform. *Proceedings of Parallel and Distributed Computing and Systems (PDCS)* (2009).

[8] DARPA INTRUSION DETECTION DATASET. `http://www.ll.mit.edu/mission/communications/ist/corpora/ideval/data/index.html`.

[9] DOBRESCU, M., EGI, N., ARGYRAKI, K., CHUN, B.-G., FALL, K., ET AL. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *SOSP* (2009).

[10] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The Implementation of the Cilk-5 Multithreaded Language. In *PLDI* (1998).

[11] KOHLER, E., MORRIS, R., CHEN, B., JANNOTTI, J., AND KAASHOEK, M. F. The Click Modular Router. *ACM Trans. Comput. Syst.* (Aug. 2000).

[12] LOO, B. T., CONDIE, T., GAROFALAKIS, M., GAY, D. E., HELLERSTEIN, J. M., MANIATIS, P., RAMAKRISHNAN, R., ROSCOE, T., AND STOICA, I. Declarative Networking. *CACM* (2009).

[13] MUKKAMALA, S., AND SUNG, A. H. Detecting Denial of Service Attacks Using Support Vector Machines. In *IEEE International Conference on Fuzzy Systems (IEEE FUZZ)* (2003).

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

[15] PRZYMUSINSKI, T. C. *On the declarative semantics of deductive databases and logic programs*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988, pp. 193–216.

[16] RAMAKRISHNAN, R., AND ULLMAN, J. D. A survey of research on deductive database systems. *Journal of Logic Programming 23* (1993), 125–149.

[17] RAPIDNET: A DECLARATIVE TOOLKIT FOR RAPID NETWORK SIMULATION AND EXPERIMENTATION. `http://netdb.cis.upenn.edu/rapidnet/`.

[18] REINDERS, J. Intel Thread Building Blocks. In *OReilly Associates* (2007).

[19] SAHAMI, M., DUMAIS, S., HECKERMAN, D., AND HORVITZ, E. A bayesian approach to filtering junk e-mail. In *Learning for Text Categorization: Papers from the 1998 workshop* (1998), vol. 62, Madison, Wisconsin: AAAI Technical Report WS-98-05, pp. 98–105.

[20] SIP: SESSION INITIATION PROTOCOL. `http://www.ietf.org/rfc/rfc3261.txt`.

[21] SIPP OPEN SOURCE TEST TOOL / TRAFFIC GENERATOR FOR THE SIP PROTOCOL. `http://sipp.sourceforge.net/`.

[22] SOMMER, R., PAXSON, V., AND WEAVER, N. An architecture for exploiting multi-core processors to parallelize network intrusion prevention. *Concurrency and Computation: Practice and Experience* (2009).

[23] SPAMASSASSIN DATASET. `http://spamassassin.apache.org/publiccorpus`.

[24] VASILIADIS, G., ANTONATOS, S., POLYCHRONAKIS, M., MARKATOS, E. P., AND IOANNIDIS, S. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *RAID* (2008).

[25] VASILIADIS, G., POLYCHRONAKIS, M., AND IOANNIDIS, S. Midea: a multi-parallel intrusion detection architecture. In *Proceedings of the 18th ACM conference on Computer and communications security* (New York, NY, USA, 2011), CCS '11, ACM, pp. 297–308.

[26] WOJTCZUK, R. `http://libnids.sourceforge.net/`.

[27] WOLF, T., WENG, N., AND TAI, C.-H. Runtime support for multicore packet processing systems. *Network, IEEE 21*, 4 (2007), 29 –37.

APPENDIX

A. SMTP EXAMPLE IN ANALOG

```
/* Assemble IP Fragments
  Takes as an input raw IP fragments and emits
  assembled IP payload. */
component ip_assembler(parallel) {
ip1  ip_pkt(@N,PKT, IP_PKT, SRC_IP2,
             DST_IP2, TRANSPORT_PROTOCOL) :-
    input(@N, PKT, SRC_IP, DST_IP),
    IP_PKT = f_processIpFrag(PKT),
    TRANSPORT_PROTOCOL = f_getTransportProtocol(IP_PKT),
    SRC_IP2 = f_getSrcIPAddr(IP_PKT, TRANSPORT_PROTOCOL),
    DST_IP2 = f_getDstIPAddr(IP_PKT, TRANSPORT_PROTOCOL),
    P = f_traffic(PKT, "input").
ip2  tcp_pkt(@N, PKT, IP_PKT, SRC_IP, DST_IP, FLOW_ID) :-
    ip_pkt(@N, PKT, IP_PKT, SRC_IP,
    DST_IP, TRANSPORT_PROTOCOL),
    TRANSPORT_PROTOCOL == "TCP",
    FLOW_ID = f_getTCPFlowId(SRC_IP, DST_IP).  }

/* Assemble TCP payload.
  Takes IP payload as input and emits assembled TCP messages. */
component tcp_process(parallel_context_order,
                      context_keys(tcp_pkt(6))) {
tcp1 tcp_stream(@N, PKT, SRC_IP,
                DST_IP, FLOW_ID, TCP_MESSAGE) :-
    tcp_pkt(@N, PKT, IP_PKT, SRC_IP, DST_IP, FLOW_ID),
    TCP_MESSAGE = f_processTCP(IP_PKT).  }

/* Detect TCP user protocol, e.g. SMTP. */
component tcp_protocol_detector(parallel_context_order,
                    context_keys(tcp_stream(5))) {
det1 detected_tcp_protocol(@N, PKT, SRC_IP, DST_IP,
    FLOW_ID, TCP_MESSAGE, PROTOCOL) :-
    tcp_stream(@N, PKT, SRC_IP, DST_IP, FLOW_ID, TCP_MESSAGE),
    PROTOCOL = f_processTCPDetector(TCP_MESSAGE, FLOW_ID).
det2 smtp_segment(@N, PKT, SRC_IP, DST_IP, FLOW_ID, TCP_MESSAGE) :-
    detected_tcp_protocol(@N, PKT, SRC_IP, DST_IP,
    FLOW_ID, TCP_MESSAGE, PROTOCOL),
    PROTOCOL == "SMTP".  }

/* Process SMTP message.
  Assemble e-mails from multiple SMTP segments. */
component smtp_processor(parallel_context_order,
                         context_keys(smtp_segment(5))) {
smtp1 smtp_mail(@N, PKT, SRC_IP, DST_IP, FLOW_ID, MAIL) :-
    smtp_segment(@N, PKT, SRC_IP, DST_IP, FLOW_ID, TCP_MESSAGE),
    MAIL = f_processSMTPSegment(TCP_MESSAGE, FLOW_ID),
    MAIL != "".  }

/* Match payload with a regular expression */
component regex_match(parallel) {
regex1 positive_match(@N, PKT, SRC_IP,
    DST_IP, FLOW_ID, MAIL, MATCH) :-
    smtp_mail(@N, PKT, SRC_IP, DST_IP, FLOW_ID, MAIL),
    MATCH = f_match(MAIL, "hello"),
    MATCH == "true".  }
```

B. SIP EXAMPLE IN ANALOG

```
/* Assemble IP Fragments
  Takes as an input raw IP fragments and emits
  assembled IP payload. */
component ip_assembler(parallel) {
ip1  ip_pkt(@N, PKT, IP_PKT, SRC_IP2,
         DST_IP2, TRANSPORT_PROTOCOL) :-
        input(@N, PKT, SRC_IP, DST_IP),
        IP_PKT = f_processIpFrag(PKT),
        TRANSPORT_PROTOCOL =
            f_getTransportProtocol(IP_PKT),
        SRC_IP2 = f_getSrcIPAddr(IP_PKT,
            TRANSPORT_PROTOCOL),
        DST_IP2 = f_getDstIPAddr(IP_PKT,
            TRANSPORT_PROTOCOL),
        P = f_traffic(PKT, "input").
ip2 udp_pkt(@N, PKT, IP_PKT, SRC_IP, DST_IP, FLOW_ID) :-
        ip_pkt(@N, PKT, IP_PKT, SRC_IP,
                DST_IP, TRANSPORT_PROTOCOL),
        TRANSPORT_PROTOCOL == "UDP",
        FLOW_ID = f_getUDPFlowId(SRC_IP, DST_IP). }

/* Detect UDP user protocol, e.g. SIP */
component udp_protocol_detector(parallel) {
ud1 detected_udp_protocol(@N, PKT, IP_PKT, SRC_IP,
        DST_IP, FLOW_ID, PROTOCOL) :-
        udp_pkt(@N, PKT, IP_PKT, SRC_IP,
            DST_IP, FLOW_ID),
        PROTOCOL = f_processUDPDetector(IP_PKT).
ud2 sip_segment(@N, PKT, IP_PKT, SRC_IP,
        DST_IP, FLOW_ID) :-
        detected_udp_protocol(@N, PKT, IP_PKT, SRC_IP,
            DST_IP, FLOW_ID, PROTOCOL),
        PROTOCOL == "SIP". }

/* Parse the SIP payload and emit SIP event */
component sip_parse(parallel) {
s1 sip_event(@N, PKT, EVENT, FLOW_ID) :-
        sip_segment(@N, PKT, IP_PKT, SRC_IP,
            DST_IP, FLOW_ID),
        EVENT = f_sipEventParse(IP_PKT).
s2 sip_event_callid(@N, PKT, EVENT, CALLID) :-
        sip_event(@N, PKT, EVENT, FLOW_ID),
        CALLID = f_sipcallid(EVENT).
}

/* Process the parsed SIP message */
component sip_process(parallel_context_order,
              context_keys(sip_event_callid(4))) {
sip1 sip_session_event(@N, PKT, SESSION_EVENT, CALLID):-
        sip_event_callid(@N, PKT, EVENT, CALLID),
        SESSION_EVENT = f_sipEventProcess(EVENT). }
```

C. DOS EXAMPLE IN ANALOG

```
/* Assemble IP Fragments
  Takes as an input raw IP fragments and emits
  assembled IP payload. */
component ip_assemble(parallel) {
ip1  ip_pkt(@N, PKT, IP_PKT, SRC_IP2, DST_IP2, TRANSPORT_PROTOCOL) :-
    input(@N, PKT, SRC_IP, DST_IP), IP_PKT := f_ProcessIpFrag(PKT),
    TRANSPORT_PROTOCOL := f_GetTransportProtocol(IP_PKT),
    SRC_IP2 := f_GetSrcIPAddr(IP_PKT, TRANSPORT_PROTOCOL),
    DST_IP2 := f_GetDstIPAddr(IP_PKT, TRANSPORT_PROTOCOL),
    P := f_Traffic(PKT, "input").
ip2 tcp_pkt(@N, PKT, IP_PKT, SRC_IP, DST_IP, FLOW_ID) :-
    ip_pkt(@N, PKT, IP_PKT, SRC_IP, DST_IP, TRANSPORT_PROTOCOL),
    TRANSPORT_PROTOCOL == "TCP",
    FLOW_ID := f_GetTCPFlowId(SRC_IP, DST_IP). }

/* Assemble TCP payload.
  Takes IP payload as input and emits assembled TCP messages. */
component tcp_process(serial_context_order,
                      context_keys(tcp_pkt(6))) {
tcp1 tcp_stream(@N, PKT, SRC_IP, DST_IP, FLOW_ID, TCP_MESSAGE) :-
    tcp_pkt(@N, PKT, IP_PKT, SRC_IP, DST_IP, FLOW_ID),
    TCP_MESSAGE := f_ProcessTCP(IP_PKT). }

/* Detect malicious TCP flows
  Extract features and classify TCP flow using support vector machine*/
component tcp_ddos_detect(parallel) {
ddos1 tcp_stream_feature(@N, SRC_IP, DST_IP, FEATURE) :-
    tcp_stream(@N, PKT, SRC_IP, DST_IP, FLOW_ID, TCP_MESSAGE),
    FEATURE := f_GetTCPFeature(TCP_MESSAGE).
ddos2 tcp_stream_attack(@N, SRC_IPSTR, DST_IPSTR) :-
    tcp_stream_feature(@N, SRC_IP, DST_IP, FEATURE),
    ISATTACK := f_DetectAttack(FEATURE),
    ISATTACK == "true", SRC_IPSTR := f_IPtoSTR(SRC_IP),
    DST_IPSTR := f_IPtoSTR(DST_IP). }
```