Quantitative Network Monitoring with NetQRE

Yifei Yuan
University of Pennsylvania
yifeiy@cis.upenn.edu

Dong Lin
LinkedIn Inc.
dolin@linkedin.com

Ankit Mishra
University of Pennsylvania
mankit@seas.upenn.edu

Sajal Marwaha
University of Pennsylvania
sajalm@seas.upenn.edu

Rajeev Alur
University of Pennsylvania
alur@cis.upenn.edu

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

ABSTRACT

In network management today, dynamic updates are required for traffic engineering and for timely response to security threats. Decisions for such updates are based on monitoring network traffic to compute numerical quantities based on a variety of network and application-level performance metrics. Today's state-of-the-art tools lack programming abstractions that capture application or session-layer semantics, and thus require network operators to specify and reason about complex state machines and interactions across layers. To address this limitation, we present the design and implementation of NetQRE, a high-level declarative toolkit that aims to simplify the specification and implementation of such quantitative network policies. NetQRE integrates regular-expression-like pattern matching at flow-level as well as application-level payloads with aggregation operations such as sum and average counts. We describe a compiler for NetQRE that automatically generates an efficient implementation with low memory footprint. Our evaluation results demonstrate that NetQRE allows natural specification of a wide range of quantitative network tasks ranging from detecting security attacks to enforcing application-layer network management policies. NetQRE results in high performance that is comparable with optimized manually-written low-level code and is significantly more efficient than alternative solutions, and can provide timely enforcement of network policies that require quantitative network monitoring.

CCS CONCEPTS

• **Networks → Network monitoring**; **Programmable networks**;

KEYWORDS

NetQRE, network monitoring language, quantitative regular expression

ACM Reference format:
Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative Network Monitoring with NetQRE. In *Proceedings of SIGCOMM '17, Los Angeles, CA, USA, August 21-25, 2017,* 14 pages.
DOI: 10.1145/3098822.3098830

SIGCOMM '17, Los Angeles, CA, USA

DOI: 10.1145/3098822.3098830

1 INTRODUCTION

Network management today often requires dynamic updates in response to traffic engineering and security events. For example, in data centers, heavy hitters [8, 40] need to be detected in real-time and bandwidth limits may be imposed on them. Within an enterprise network, users can be rate-limited if they exceed quotas on their application usage. Network traffic anomalies that are detected require immediate mitigation strategies to block potential security attacks [22].

Decisions for such updates require *quantitative network monitoring* capabilities, which is a combination of monitoring a variety of network and application-layer performance metrics with known traffic patterns, and the real-time computation of quantitative aggregate values that are used as a basis for network configuration updates in order to meet performance and security goals.

Imperative languages provide low-level abstractions, which makes it cumbersome to perform complex quantitative analysis on packet streams. To illustrate this difficult, we consider a quantitative network monitoring task of detecting two well-known denial-of-service attacks: Slowloris [36] and the SSL renegotiation attack [6]. The former attack requires tracking TCP connections, and raising an anomaly when the number of bytes transferred is unusually lower than normal. The latter attack requires identifying traffic signature patterns for TLS renegotiations, and counting the number of such renegotiations across multiple HTTP sessions. Detecting both attacks require identifying HTTP or TCP traffic in packet streams, extracting out attack patterns, and monitoring quantitative values to identify possible anomalies. As new packets arrive, state has to be incrementally updated as TCP sessions are established and torn down. A more natural and intuitive approach for a programmer to detect these attacks is to provide a framework to express these monitoring tasks in a modular way, i.e., define the pattern of a TCP connection or HTTP request, recognizing data transfer packets for the connection or TLS renegotiations, and then aggregating traffic rates or renegotiation counts as connections are established.

Quantitative network monitoring is not only restricted to security use cases, but are also useful for enforcing network management policies based on a given application-level metric. For example, we consider a policy that aims to enforce a particular bandwidth quota for a given application (e.g. VoIP) on a per-user basis. This policy requires identifying VoIP traffic in packet streams, manually maintaining state across packets to detect the start of each VoIP session, extracting out user information, and monitoring aggregate VoIP usage per user even as their IP address is updated. Supporting such monitoring functionality requires identifying VoIP

session for each user from input packet streams, monitor usage for each identified VoIP session, and aggregating usage across all VoIP sessions grouped-by all users being tracked.

Today, there are several point-solutions to support certain aspects of quantitative network monitoring. However, they suffer from one or more of the following limitations. First, a large majority of network measurement tools focus on flow-level measurements [15, 18, 19, 39–41] that do not capture application-level or session-level semantics. This excludes a range of policies that are application-dependent, for example, tracking security signatures at the application level or rate-limiting users based on their VoIP call usage. Second, these tools tend to provide ad-hoc solutions that are difficult to generalize or customize. Programming frameworks for software-defined networks (SDN) [21, 30, 32], do not support integration with queries beyond basic flow-level counters, and none of these languages support quantitative monitoring at the application or session level. Finally, tools such as Bro [33] requires network operators to have significant programming expertise to implement state machines and reason about state transitions.

To address the above limitations, we present NetQRE, a practical tool aimed at simplifying the specification and implementation of quantitative network policies. Our proposal is based on the observation that traffic patterns such as a TCP connection and a application-level session can be specified using *regular expressions*, which are an abstraction that network operators who may not be well-versed in programming find natural to use. Regular expressions are widely used in systems management, for example, in the popular grep tool for searching for patterns in systems logs, application-level packet classification [1], signature-based attack detection [3, 34]. We make the following contributions.

• **NetQRE language**. We present the NetQRE declarative language that provides high-level abstractions and express a variety of quantitative policies that span multiple packets grouped into flows and application-level sessions. NetQRE is based on novel theoretical foundation of *parameterized quantitative regular expressions* (PQRE). PQRE extends prior work on QRE [9], which provides a formal foundation of combining traditional regular expressions with numerical computations. NetQRE integrates regular-expression-like pattern matching at flow-level and application-level payloads with aggregation operations such as sum and average counts. The language further supports *quantitative network policies* by allowing actions on packets to be generated as output of monitoring applications.

• **Compilation and efficient runtime system.** We developed a compiler that can automatically generate efficient NetQRE implementations with low memory footprint. As part of the compilation process, the compiler automatically infers the state that needs to be maintained for NetQRE programs, and optimizes the compiled imperative code. A NetQRE runtime then executes the generated code efficiently. In fact, this process also eases the burden on programmers having to handwrite optimized versions in low-level code.

• **NetQRE implementation and evaluation.** We have developed a NetQRE prototype which we evaluate over a range of quantitative network monitoring tasks. Our evaluation results demonstrate that NetQRE can express a wide range of quantitative network policies with no more than 18 lines of code, while specifying them in imperative languages requires at least 100 lines of code. The compiled implementations incur no more than 9% overhead in throughput compared with optimized manually-written low-level code, are able to handle more than 10Gbps of traffic using a single core (and is amenable to parallelization), and are significantly more efficient (e.g. 11×) compared to monitoring solutions based on Bro and OpenSketch [40].

2 OVERVIEW

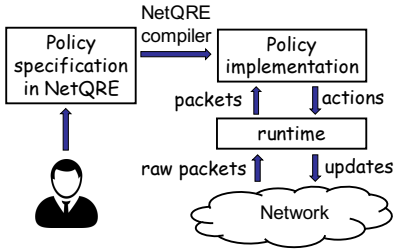


Figure 1: NetQRE Architecture.

Fig. 1 shows the architecture of NetQRE. To program a quantitative network monitoring task, a programmer simply specifies this monitoring task in a declarative fashion by viewing the input as a *stream of packets*. The NetQRE compiler automatically generates efficient low-level imperative code that implements the specification. Each incoming raw packet from the network is first parsed and preprocessed by the runtime into a form that can be referenced by the monitoring implementation. In our deployment, NetQRE relies on the underlying runtime to handle packet reordering and retries (if needed) due to losses in TCP connections, and also fragmentation and defragmentation of IP packets. The stream of processed packets are then sent to the appropriate NetQRE program for execution.

As the NetQRE compiled program executes, the incoming packets are processed in a streaming fashion. The output of a NetQRE program is the generation of analysis results or actions taken to reconfigure the network. The NetQRE tool can be deployed in different settings, for example, a tap on a SPAN port analyzing mirrored traffic, an inline solution, or running in the cloud as a virtualized middlebox. Our language design and compiler is agnostic to the deployment setting. While we focus on packet stream processing in this paper, the NetQRE language and compilation is agnostic to the input stream and can handle any item streams (e.g. raw packets, email messages, video frames).

2.1 Motivating Example

As a motivating example, we consider a network management policy where an enterprise monitors the usage of Voice-over-IP (VoIP) for each user, and alert the user whose usage is significantly higher than the average usage over all users. The actual language specification will be described in later sections. We provide a high-level intuition of programming language features necessary to support this policy. Note that our examples are not solely limited to VoIP. Section 4 provides more examples to showcase the wide-applicability of NetQRE, and our evaluation section has a detailed listing.

We use the Session Initiation Protocol (SIP) used in VoIP applications as our example. Typically, a call based on SIP consists of three phases, *init*, *call*, and *end*. In the *init* phase, the caller and the callee exchange messages (e.g. call ID, user name, the connection channel) in order to set up the call session. Once established, the *call* phase allows VoIP data to be transmitted between the caller and callee using the channel defined in the first phase. Finally, either side can end the call as shown in the *end* phase.

To analyze the SIP call in the midst of network traffic from all types of protocols, the protocol analyzer is required to (1) identify all SIP traffic traversing the network, (2) separate the SIP traffic into different sessions for different caller/callee pairs, (3) group packets within each session into phases (init, call, and end), and then perform a count of bytes only within the second (call) phase.

NetQRE offers a natural programming model to implement this policy. Conceptually, the input to a NetQRE program is modeled as a stream of packets. The programmer may assume that all received packets have been stored and presented as the input. Based on this input, NetQRE then provides programming abstractions for a programmer to implement various functions to process the stream. These functions can then be composed in a modular fashion to implement the quantitative policy. Using the VoIP example, the programmer specifies a *filter* function to identify SIP traffic of each user, an *iteration* function that splits the SIP traffic stream into a sequence of VoIP sessions, a *split* function that further breaks up each VoIP session into the three phases, and finally, an *aggregation* function that sums up the total bandwidth in the call phase.

These functions are specified by users in a declarative fashion, and composed together in a fashion that requires users to only think in terms of protocol patterns over multiple packets. Explicit state maintenance to track session state is handled by the NetQRE runtime. Note that this programming model is a conceptual one, and the actual execution is optimized by our compiler. For example, it will incur unacceptable storage and performance overhead if the runtime system has to log all incoming packets and present them as a program input. We have implemented a NetQRE compiler which compiles a NetQRE program into an optimized imperative program. The compiler automatically infers the states that needs to be maintained for the NetQRE program, and generates execution code that implements a streaming algorithm to update the states for each input packet, and evaluate the output in an incremental way at runtime.

2.2 Alternative Approaches

To implement this policy in a low-level language, a programmer often faces the challenges such as what state needs to be maintained and updated in order to track the progress of the SIP protocol. The programmer also needs to track each user's media traffic based on the correct SIP state and aggregate the average VoIP usage across all users. This makes the implementation of the policy highly coupled with the low-level implementation of the SIP protocol. While individual filter, split and aggregation functions can be implemented in a low-level language, composing them together to implement the correct functionality is challenging. Although there are tools today for VoIP monitoring (e.g. [17]), they lack *extensibility* to support other applications and policies. For example, if a new policy requires to monitor video usage instead of VoIP, one has to use other tools.

Recently proposed domain-specific languages and tools cannot address this problem either. For example, traffic measurement tools typically focus on per source-destination traffic measurement and cannot be used to monitor an application's usage. SDN programming frameworks only offer basic flow statistics based on flow-level counters on switches, and do not have the abstraction to address the above mentioned challenge. Event-based intrusion detection systems (IDS) such as Bro [33] generate high-level events at the session level and application level from network traffic. However, their focus is primarily on intrusion detection, and hence lack language primitives that make it natural to support a wide range of quantitative monitoring capabilities. For example, we observe that it is feasible to implement an analyzer that counts the number of VoIP calls using Bro, but significantly harder to further identify the packets corresponding to the call phase to calculate bandwidth utilization. Bro also requires the user to be an expert programmer knowledgeable with state machine models.

3 THE NETQRE LANGUAGE

In NetQRE, a packet is modeled as a sequence of bytes, and we use parsing functions to extract information from the packet. Common parsing functions include `srcip` which returns the source IP of the packet, `srcport` (source port), `syn` (SYN bit), `data` (bytes in payload), and `time` (the time stamp on receipt of the packet). The parsing functions can be customizable by the user, for example, to extract application-level headers.

Values, variables and functions in NetQRE are typed. NetQRE offers basic types, such as **`int`**, **`bool`**, **`string`**, as well as a set of domain-specific types, such as **`IP`** (IP addresses), **`Port`** (TCP and UDP ports), **`packet`** (all packets), and **`action`**. The **`action`** type consists of predefined functions that either generate alerts or send updates to switches. NetQRE also provides high-level types such as **`Conn`** (tuples of source IP-port and destination IP-port), which is used for TCP and UDP packets.

NetQRE offers a convenient way to write *stream functions* to process packet streams. A stream function takes as input a stream of packets, and produces as output values (e.g. monitoring results to an application), actions (e.g. alerts to the controller), or packets (e.g. packets filtered from an application). A stream function can be specified as below.

```
sfun type func_name(type var) = exp;
```

The stream function declaration includes the keyword **`sfun`**, returned type of the stream function, followed by the name of the function. Any other arguments are specified following the function name. The body of each stream function (right-hand-side to '=') consists of *expressions* which are used to specify the functionalities of the stream function.

Figure 2 shows a summary of the syntax of NetQRE expressions. Expressions in stream functions are based on the theoretical foundations of *quantitative regular expressions* (QRE) [9], a novel proposal that integrates regular expressions with numerical computations. In the rest of this section, we describe the features of NetQRE expressions in stages. We first introduce how to use an extension to regular expressions to detect patterns of the input stream. Second,

Predicate	*P*	::= **.** | [field = value] | [field = variable] | *P*&*P* | (*P* '||' *P*) | !*P*
Regular Exp.	*re*	::= *P* | *re re* | *re* * | *re* '||' *re*
Conditional	*cond*	::= *exp* ? *exp* | *exp* ? *exp* : *exp*
Aggregation	*agg*	::= aggop { *exp* | type variable }
	aggop	::= **sum** | **avg** | **max** | **min**
Split	*split*	::= **split**(*exp* , *exp*, aggop)
Iteration	*iter*	::= **iter**(*exp*, aggop)
Composition	*comp*	::= *exp* >> *exp*
Expression	*exp*	::= value | action | op *exp* | *exp* op *exp* | *re* | *cond* | *agg* | *split* | *iter* | *comp*

Figure 2: Syntax of NetQRE expressions.

we discuss how to associate values and actions for the stream. Finally, we describe a set of high-level operations that allow modular programming of stream functions.

3.1 Pattern Matching over Streams

The basic feature of NetQRE is to detect the patterns of the input packet stream. As the basic building block, NetQRE uses an extension of regular expressions, to which we refer as *parameterized symbolic regular expressions* (PSRE), for pattern matching over the input stream. Regular expressions (RE) are widely used for pattern matching over strings (sequences of bytes). Typically, a RE uses a fixed finite alphabet, and the atoms in a RE are symbols in the alphabet. In NetQRE, a PSRE generalizes a RE in the two ways.

First, the atoms in a PSRE are predicates over packets instead of single symbols, which allows a PSRE to handle very large and potentially infinite alphabet. This property makes PSRE an appealing fit for customized network filtering, since the space of packets is often very large and typically one is only interested in the values of some fields in a packet (e.g. the source and destination) instead of the whole packet itself.

Second, PSRE allows the use of *parameters* to represent unknown values in the predicate. With this generalization, a PSRE can detect a variety of patterns using different instantiation of the parameters. As a result, it allows the programmer to specify applications such as counting the number of distinct IP addresses appeared in the stream, which is hard, if not impossible, to be specified without parameters, because no concrete predicates can be defined without knowing those IP addresses at runtime.

Predicate. A basic packet predicate `[f = v]` checks whether the value in the field `f` of a packet is `v`. As an example, the predicate `[srcip=\`1.0.0.1']` matches all packets whose source IP address is `1.0.0.1`. As an example of the use of parameters, `[srcip=x]` defines a function from the domain of `x` (i.e. IP addresses in this case) to a concrete predicate over packets. We also use `.` to denote the predicate that matches all packets. Predicates can be composed using standard boolean combinations. NetQRE also provides handy macros for widely used predicates. For example, `is_tcp(c)` is a shorthand for the predicate that matches a TCP packet in the connection `c`.

PSRE. Like REs, the basic operations for a PSRE are concatenation, union and Kleene star. For example, `/[syn=1][syn=0]*/` matches a stream of packets where the first packet is a SYN packet followed by a sequence of non-SYN packets (including 0 non-SYN packets). Note that, syntactically, predicates are enclosed in square brackets and a PSRE is enclosed in a pair of slashes in NetQRE, and `*` is the Kleene star operator. To illustrate the use of parameters, consider the example to count distinct source IP addresses in the stream. The core PSRE to implement this example is the function `exist(x)` which checks whether a source IP `x` appeared in the stream. The function can be specified using the PSRE as below.

```
sfun bool exist(IP x) = /.*[srcip=x].*/;
```

We defer the discussion of the full expression for this example in §3.5.

3.2 Conditional Expressions

Given the pattern matching capability for the input stream, it is natural to use conditional expressions to incorporate PSRE with other values and actions in order to assign cost/generate actions to the stream.

In NetQRE, a conditional expression has the form exp? exp_1, where, exp is an expression that returns boolean values such as a PSRE defined above, and exp_1 is an expression in NetQRE. A stream evaluated to `true` by exp will be applied to exp_1, otherwise this expression is not defined and returns `undef`.

For example, the expression `/./?1` returns `1` if the stream matches the PSRE `/./` (i.e. the stream consists of a single packet), otherwise it returns `undef`; the expression `/.*/?size(last)` returns the size of the last packet in the stream. The keyword `last` denotes the last received packet in the input stream. As another example, `(count>k)?alert` returns an action `alert` when the total number of packets in the stream is larger than `k`. Here, `count` is a stream function that counts the number of packets in the stream, and `alert` is a pre-defined action to generate an alert event, for example, to a controller node. These actions are a basis for implementating quantitative network policies using NetQRE.

A conditional expression can also be specified as exp? exp_1 : exp_2, which applies exp_2 to the stream if exp is not satisfied. To ensure the consistency, exp_1 and exp_2 should return the same type for the stream. As an example, the expression `/[srcip=\`1.0.0.1']/?1:0` returns `1` if the stream contains a single packet with source IP `1.0.0.1`, and it returns `0` if the stream contains multiple packets, or the only packet in the stream does not have the source IP.

3.3 Stream Split

In many network applications, the input stream consists of multiple phases. For example, a VoIP call splits into three phases as shown in the motivating example. It is convenient to handle different phases of the stream separately, and combine the results of each phase in a modular way. Following the operators defined in Quantitative Regular Expressions [9], NetQRE uses the **`split`** operator to split the stream and compose two stream functions.

A stream split expression has the form **`split`**`(f,g,aggop)`, where `f` and `g` are two expressions, and `aggop` is an aggregation operator such as **`sum`**, **`avg`** (average), **`max`**, and **`min`**.

For an input stream ρ, a **split** function splits ρ into two substreams ρ_1 and ρ_2, such that `f` is defined on the first substream ρ_1 and `g` is defined on ρ_2. `f` and `g` are applied respectively to the two substreams, and the returned value is aggregated using `aggop`, as the return value of **split**(`f,g,aggop`). The **split** operation is a natural quantitative generalization of the concatenation operation in regular expression, and Fig. 3 illustrates how a **split** expression works.

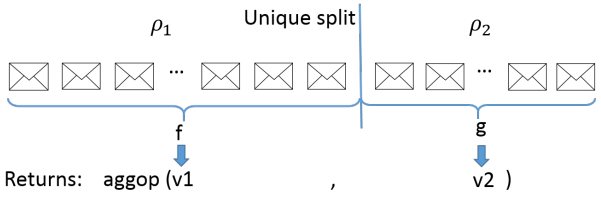


Figure 3: Illustration of `split`.

Note that, in order to return a unique value for a **split** expression, it is required that the splitting is unambiguous. That is, for all input streams and all values of parameters in the expression, there is at most one way to split stream for `f` and `g`. The property of unambiguity can be checked efficiently at compile time [9]. When no unambiguous splitting is possible, the **split** expression returns `undef` for the input stream.

As an example of **split**, consider the example of counting the number of packets since the last SYN packet in the stream. Naturally, one can split the stream into two substreams separated by the last SYN packet, and count the number of packets in the second substream, as shown in the following expression.

```
split(any?0, last_syn?count, sum)
```

Here, `any` is the PSRE `/.*/` that matches any packet streams, and `last_syn` is the PSRE `/[syn=1][syn=0]*/` that matches a stream that starts with a SYN packet followed by non-SYN packets. Putting together, the **split** expression splits the input stream before the last appearance of a SYN packet. Note that the first substream is assigned the value 0, while the second substream is applied to the `count` function (defined later in §3.4) to count the number of packets.

3.4 Stream Iteration

A network application often requires to iterate over the input stream. For example, counting the number of packets in the stream requires to iterate over all single packets.

Similar to the **split** expression, NetQRE offers the **iter** operator to iterate over the stream. An **iter** expression is of the form **iter**(`f,aggop`), where `f` is an expression in NetQRE and `aggop` is an aggregation operator. The **iter** expression splits the input stream into multiple substreams, such that `f` is defined on each substream. It then iterates through all substreams and evaluates `f` on each substream, and finally aggregates the return value on each substream using the aggregation operator. The **iter** operator is a quantitative generalization of the Kleene star operation, and Figure 4 illustrates this process. Again, the splitting is required to be unambiguous for all input streams to `f`.

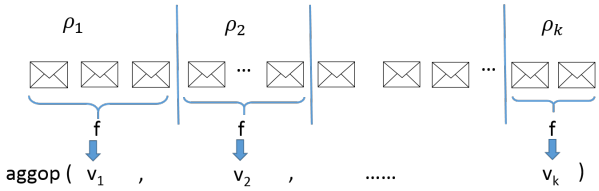


Figure 4: Illustration of `iter`.

As an example, the function `count` that counts the number of packets in the stream can be specified below.

```
sfun int count = iter(/./?1, sum)
```

This function splits the stream into single packets, and the inner expression returns `1` for each single packet. As a result, the whole **iter** expression counts how many packets in the stream.

3.5 Aggregation over Parameters

Aggregation is a common feature that many network monitoring applications share. For example, computing the average flow size needs to aggregate the average size across multiple flows. NetQRE offers high-level aggregation expressions to aggregate functions over parameters.

Aggregation expressions are of the form `aggop{f | T x}`, where `x` is a parameter with its type `T`, and `f` is an expression where the parameter `x` appears in it. Intuitively, the aggregation expression goes through all possible values of `x`, and evaluate `f` using the substituted value for `x`, and finally aggregates all valid return values using the aggregation operator `aggop`. Revisiting the example of counting distinct source IP addresses in a stream, we can use the expression **sum**`{exist(x)?1:0 | IP x}`.

3.6 Stream Composition

Typically the input stream consists of packets from multiple sources and destinations. Oftentimes, the programmer only wants to handle a stream from a particular source, for example. NetQRE offers the stream composition operator >> that allows to preprocess the stream before applying another stream function.

Stream composition expressions are of the form `f >> g` where `f` and `g` are two stream functions. The stream composition allows the processing of a stream using the first stream function `f` repeatedly on every prefix of the stream, and the returned outputs yield a new stream that is then piped as the input stream to a second stream function `g`. For example, if the stream contains two packets (P_1, P_2), `f` is first applied to P_1 as a single-packet stream, and then (P_1, P_2). The output of `f` on the two substreams is then piped to `g`.

Stream composition is useful when the stream of packets need to be preprocessed in order to filter related packets for further processing. For example, counting the number of all packets in a TCP connection `c` can be specified using `filter_tcp(c)>> count`, which first filters all TCP packets in the connection `c` using the function `filter_tcp`, and then counts the filtered stream using `count`. The function `filter_tcp` is defined as follows:

```
sfun packet filter_tcp(Conn c) =
  /.*[is_tcp(c)]/?last;
```

Filter functions are convenient and used extensively in our use cases. As a short-land, NetQRE uses `filter(p)` for the filter function that filters packets satisfying the predicate `p`. For example, the above filter function can be abbreviated as `filter(is_tcp(c))`.

Stream composition can also be used to filter packets according to timestamps. NetQRE builds in two filter functions based on timestamps. The `recent(t)` function filters the stream in the recent `t` seconds, and the `every(t)` function periodically filters the stream in every `t` seconds. For example, `recent(5)>>count` counts the number of packets in the recent 5 seconds. Since the two build-in filter functions are not in the core of NetQRE, we only allow the use of time-based filtering outside the high-level NetQRE operations described above.

4 USE CASES

We provide use cases, ranging from flow-level traffic measurements, to complex examples involving application-level content analysis and dynamic updates. Since the high-level constructs in NetQRE language is based on regular expressions, we can only support queries for regular traffic patterns. Nevertheless, we note that this covers a wide range of useful queries, as we will show in this section. More examples can be found in our technical report.

4.1 Flow-level Traffic Measurements

We highlight two measurement tasks that have been proposed recently as a means to do flow scheduling and attack detection.

Heavy hitter. Heavy hitters [19] are flows that consume bandwidth larger than a threshold `T`. A key step to identify a heavy hitter is to count the size of a flow. In this example, we consider a flow as a source-destination pair. A natural way to specify this functionality is to first filter all packets from a source `x` to a destination `y`, and then count the size of the filtered stream.

```
sfun int hh(IP x, IP y) =
  filter(srcip=x, dstip=y) >> count_size;
```

`hh` first filters packets based on the source and destinations IP, and second, the filtered stream is piped into `count_size` which counts the size of all packets in the stream. Due to space limits, we do not show the definition of `count_size` which is similar to that of `count`.

Second, to alert a heavy hitter in real time to the controller, we can use the following program.

```
sfun action alert_hh =
  (hh(last.srcip, last.dstip)>T) ?
  alert(last.srcip, last.dstip);
```

The function `alert_hh` checks for every newly received packet (i.e. `last` in the stream) that whether its flow reaches the threshold `T`, and issues an `alert` correspondingly. By further applying the time-based filtering `every(5)>>alert_hh`, one can detect heavy hitters in every 5 seconds.

Super spreader. A super spreader [40] is the host that contacts more than k distinct destinations during a time interval. Like the use case of heavy hitter detection, a key step is to count how many distinct destinations a source `x` contacted. The following NetQRE program does this counting.

```
sfun int ss(IP x) =
  sum{ exist_pair(x, y)?1:0 | IP y};
```

The function `exist_pair(x,y)` checks whether the source-destination pair `(x,y)` appeared in the stream. The `ss` function uses an aggregation function **`sum`** to aggregate all possible destinations, and thus gives the total number of distinct destination addresses `x` contacted.

Similar to heavy hitter detection, we can use stream composition to filter the input stream based on time as well as traffic types. For brevity of the paper, we omit the discussion of these functionalities.

4.2 TCP State Monitoring

We next showcase applications that rely on monitoring the states within a TCP flow.

SYN flood attacks. In this example, NetQRE is used to detect SYN flood attacks by counting the number of incomplete TCP handshakes in a time interval. We consider an incomplete TCP handshake as a packet trace consisting of a SYN packet and a SYNACK packet, with corresponding sequence number and acknowledge number, but does not have a further ACK packet to complete the handshake. As the key step, the following program counts the number of incomplete handshakes, assuming that the input stream consists of TCP packets between the same source and destination.

```
sfun int bad_tcp_pat(int x, int y) =
 concat(syn(x), synack(y,x+1), no_ack(y+1));
sfun int incomplete_handshake_num =
  sum{bad_tcp_pat(x,y)?1 | int x, int y};
```

The function `bad_tcp_pat` specifies the pattern of an incomplete handshake: there is a SYN packet with a sequence number `x` in the stream, followed by a SYNACK packet with acknowledge number `x+1` and sequence number `y`, and then followed by packets that do not include an ACK with acknowledge number `y+1` to complete the handshake. The **`sum`** aggregation in `incomplete_handshake_num` sums up the number of appearances of such patterns for all `x` and `y`. Using the stream composition of filtering functions based on time and packet type, the complete function can be specified as follows:

```
sfun int syn_flood(Conn c) =
  recent(5) >>
  filter_tcp(c) >>
  incomplete_handshake_num>T?block(c.srcip);
```

Completed flows. Our next example counts the number of legitimate flows that are completed, delineated by a SYN at the beginning, and ending with a FIN. Note that the last **`iter`** uses the regular expression which matches a stream ending with a SYN and a FIN packets, and no SYN-FIN pairs appear before. Therefore, the **`iter`** expression splits the entire (filtered) stream into sessions where each session contains exactly one complete flow.

```
sfun int count_flow(Conn c) =
   filter_tcp(c) >>
   filter(flag=SYN || flag=FIN) >>
   iter(/[fin=1]*[syn=1]*[syn=1][fin=1]/?1, sum);
```

Slowloris attacks. We describe a detection of the aforementioned Slowloris attack in NetQRE, based on computing the average rate of packet transferring in all legitimate TCP connections, shown below.

```
sfun double conn_rate = iter(tcp_pattern?rate, sum);
sfun double avg_rate =
  sum{filter_tcp(c) >> conn_rate | Conn c}
  / sum{count_flow(c) | Conn c};
```

First, to compute the rate of a legitimate TCP connection, we can simply specify the pattern of a legitimate TCP connection using RE, and compute its transferring rate (the `rate` function can be specified easily and thus not shown here). Then we can iterate all TCP connections given a 5-tuple, and compute the aggregated transferring rate, as shown in the function `conn_rate`.

Second, we use **`sum`** operation to aggregate transferring rate across traffic on all 5-tuples. Dividing the aggregated rate by the number of flows computed using the function defined above gives us the average transferring rate over all connections, as shown in `avg_rate`.

4.3 Application-level Monitoring

Our final example revisits the Voice-over-IP (VoIP) usage usecase in §2.

Let us first focus on the function to monitor the usage of a VoIP call based on a SIP connection `sip_conn` and media connection `m_conn`. As introduced in §2, a VoIP call consists of three phases; and a modular programming way is to handle the three phases using three stream functions, and then compose them using the **`split`** operator. The function is shown below.

```
sfun int usage_per_call(Conn sip_conn, Conn m_conn,
     string user, string id) =
  split(init(id,user,m_conn)?0,
        call(m_conn)?count_size,
        end(id)?0, sum);
```

Here, `init`, `call` and `end` are simply the PSREs that capture the patterns of each phase. Note that the `init` and the `end` phase return a value `0`, and only the usage of the call phase is counted, as required.

Next, a programmer can view the traffic using the SIP connection `sip_conn` and the media connection `m_conn` as a sequence of calls. Using the **`iter`** operator, the programmer can easily aggregate the usage of all calls in the traffic, as shown below.

```
sfun int usage_per_conn(Conn sip_conn, Conn m_conn,
     string user, string id) =
  filter_sip(sip_conn, m_conn, user, id) >>
  iter(split(any?0, usage_per_call(sip_conn, m_conn,
       user, id), sum),sum);
```

Note that we first filter all traffic that belongs to `sip_conn` or `m_conn`, in order to get the stream of interest. This filtering may result in non-VoIP traffic between two VoIP calls, and thus we simply compose the PSRE `any` with `usage_per_call` inside the **`iter`** expression, to handle any traffic between two calls.

Now we can aggregate the usage for each user across multiple connections and calls using the aggregation operation, and further compute the average usage over all users, as shown below.

```
sfun int usage_per_user(string user) =
  sum{usage_per_conn(sip_conn, m_conn, user, id) | Conn
       sip_conn, Conn m_conn, string id};

sfun int average_usage =
  avg{usage_per_user(u) | string u};
```

Suppose we need to implement a quantitative network policy that sends an alert to the controller if a user's usage is larger than five times of average usage, we can simply specify the policy as below.

```
sfun action alert_high_usage(string user) =
  (usage_per_user(user) > 5*average_usage) ? alert(user
       );
```

5 COMPILATION ALGORITHM

We next describe how to compile a NetQRE program into an efficient executable that can run with low memory footprint. Typically the stream of packets is very large, thus it is not feasible to store the entire stream of packets. Therefore, the compiled program should evaluate *incrementally* on each single incoming packet without storing the history of the stream. To achieve this goal, we have to address the following challenges. First, the compiled program needs to maintain parameters in NetQRE in a succinct way. Second, in order to implement the semantics of **`split`** and **`iter`**, these operations have to be performed in a single online pass over streaming packets. Lastly, the compiled program needs to handle aggregation expressions over parameters. In the following subsections, we describe the compilation of selected NetQRE expressions.

5.1 Compilation of PSRE

First, we describe the algorithm for compiling a PSRE, as the basic building block for stream functions. Similar to traditional RE, a PSRE can be translated to an equivalent finite state machine, which we refer to as *parameterized symbolic automaton* (PSA). For example, consider the following PSRE `/.*[srcip=x].*/`. Intuitively this PSRE checks whether there exists a packet with source IP `x` in the stream. Its corresponding PSA is shown in Fig. 5.

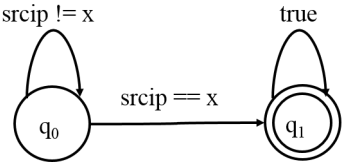


Figure 5: An Example PSA.

However, PSA cannot be directly updated due to its uninstantiated parameters. To illustrate this challenge, consider a naive solution which is to instantiate the parameters using all possible values, and maintain all the instantiated state machine, namely symbolic automata (SA), for each instantiation of the parameters. When reading an input packet, we update all the instantiated SA in the standard way. However, it is not hard to note that this approach is not feasible due to the large space of parameters. As an example, the PSA in Fig. 5 with a single IP parameter has up to 2^{32} instantiated symbolic automata. Whereas at runtime, the function only needs to store the source addresses that appeared in the stream, which can be significantly less than 2^{32}.

To address this challenge, we propose a new algorithm that updates PSA on-demand. As the high-level idea, suppose we instantiated a PSA to a set of SA using all possible instantiations. Notice that even though there is a large space of instantiated SA, however, many of them will keep the same states at runtime given a stream of packets. As an example, consider feeding a first packet with the source IP `10.0.0.1` to all SA instantiated from the PSA in Figure 5. It is easy to see that the only SA that will transition to state q_1 is the one where `x` is instantiated by `10.0.0.1`; and all other SA will stay at q_0.

Therefore, the compiled program at runtime maintains *guarded states* for the PSA, where a guarded state is pair (s, ϕ). Here, ϕ is

a predicate of the parameters in the PSA (which we will refer as a guard), and s is a state in PSA. The pair (s, ϕ) means that when reading the input stream, all instantiated SA will be at state s, if they are instantiated by the values satisfying ϕ. For example, initially there is only one guarded state maintained for the PSA in Figure 5, which is $(q_0, \texttt{true})$, meaning that all instantiated SA is at the initial state q_0 of the PSA.

To update the PSA, the key step is to update each guarded state dynamically when reading packets from the stream. The updating algorithm for a guarded state is shown in Algorithm 1. This algorithm iterates through all transitions from the state s, and for the predicate (with parameters) P on the transition, it instantiates P using the current packet it receives in the stream. This step simply replaces the function name in P by the return value of it on the packet. The obtained predicate T is a predicate over the parameters in the PSA. Finally, the algorithm emits a new pair (t, ϕ') if ϕ' is not `false`. Intuitively, this step accounts the fact that the instantiated SA under ϕ' will transition to the next state t.

Algorithm 1 update_psa((s, ϕ), packet)

1: **for all** transitions originated from s **do**
2: let t be the destinate state, and P be the parameterized predicate
3: let T be the guard instantiated from P using the packet
4: let $\phi' = \phi \wedge T$
5: emit (t, ϕ') if $\phi \neq$ `false`
6: **end for**

Using Algorithm 1, the overall updating algorithm is straightforward. Initially, the compiled program only contains a guarded state $(q_0, \texttt{true})$, where q_0 is the initial state of the PSA. Every time reading a new packet from the stream, we call Algorithm 1 on every guarded state, and the new guarded states consists of all the ones emitted from Algorithm 1. To check the state given concrete values for parameters, we simply go through all maintained guarded states, and return the state if the guard is satisfied.

Example. Using the example in Figure 5, we illustrate the execution of Algorithm 1 given the pair $(q_0, \texttt{true})$ and the packet with source IP `10.0.0.1`. Consider the transition from q_0 to q_1. By substituting `srcip` in the predicate with the source IP of the packet, we get the guard T which is `x=10.0.0.1`. Since ϕ is true now, ϕ' is simply `x=10.0.0.1`. At the end, the algorithm will emit the pair $(q_1,$ `x=10.0.0.1`$)$. Similarly, for the other transition the algorithm emit the pair $(q_0,$ `x!=10.0.0.1`$)$. Therefore, there are two new guarded states, namely, $(q_1,$ `x=10.0.0.1`$)$ and $(q_0,$ `x!=10.0.0.1`$)$. The updated guarded states account for the fact that `10.0.0.1` has appeared and thus the corresponding instantiated SA transitioned to state q_1.

5.2 Compilation of `split`

We next discuss how to compile a **`split`** expression which is of the form **`split`**`(f, g, aggop)`. First, we highlight the key insight from [9] to compile **`split`** without parameters, and then describe how to generalize this idea to compile **`split`** with parameters.

Without parameters. The challenge of compiling **`split`** is to split the stream dynamically at runtime without revisiting the entire history of the stream. For example, in the **`split`** example in §3.3, we need to split the stream at the last appearance of the SYN packet. A natural way to implement the semantics of **`split`** is to maintain all cases to split the stream. For example, Fig. 6 shows two cases to split the stream consisting of a non-SYN and a SYN packet at runtime for the example in §3.3. Moreover, the following two observations ensure that the compiled code only uses a constant space. First, each case can be represented using a triple (s_f, s_g, F), where F is a flag indicating whether the stream is split, and s_f(s_g, resp.) is the state of f(g, resp.) on evaluating the prefix(suffix, resp.) of the stream. Second, the number of maintained (distinct) cases is bounded by a constant only related to g at any time point.

Figure 6: Example run of `split`.

With parameters. Similar to PSRE compilation, in the general scenario, we need to maintain guarded cases. That is, we maintain a set of (T, ϕ) pairs, where T is a triple (s_f, s_g, F) corresponding to a case as defined above, and ϕ is a guard. It means that if the parameters are instantiated by values satisfying ϕ, there is a case of splitting the stream, which can be represented as T.

Algorithm 2 update_split((T, ϕ), packet)

1: suppose $T = (s_f, s_g, F)$
2: **if** F is `false` **then**
3: **for all** emitted (s'_f, ϕ') from update_f((s_f, ϕ), packet) **do**
4: **if** `f` is defined on s'_f **then**
5: emit $((s'_f, s^0_g, \texttt{true}), \phi')$ {s^0_g is the initial state of `g`}
6: **end if**
7: emit $((s'_f, s_g, \texttt{false}), \phi')$
8: **end for**
9: **else**
10: **for all** emitted (s'_g, ϕ') from update_g((s_g, ϕ), packet) **do**
11: emit $((s_f, s'_g, \texttt{true}), \phi')$
12: **end for**
13: **end if**

Algorithm 2 shows the algorithm to update a pair (T, ϕ). The algorithm feeds the input packet to `f` or `g` corresponding to whether the stream has been split as indicated by F. For example, when F is `false`, namely the stream has not been split yet in this case, the algorithm need to update `f` on the packet (line 2-9). In this case, for each emitted guarded state (s'_f, ϕ') from the update of `f`, a corresponding guarded case is emitted (line 7). Moreover, if `f` is defined on s'_f, we may split the stream at the current position and thus emit the guarded case $((s'_f, s^0_g, \texttt{true}), \phi')$ (line 4-6). Here, s^0_g is the initial state of `g`, and the flag F is set to `true` to indicate that the stream is split. For the else branch from line 10 to 12 in the algorithm, we need to update (s_g,ϕ) using `g`'s updating procedure, which may emit a set of guarded states of `g`. Correspondingly, the guard needs to be refined.

Given concrete values for parameters, evaluating a **split** expression is as follows. First, we check if there exist a guarded case ((s_f, s_g, F),ϕ) such that ϕ is satisfied and both f and g are defined on the states in the case. Then we evaluate the two expressions based on their states in the case, and finally aggregate the evaluated values using aggop. If no such guarded cases exist, the **split** expression is not defined.

5.3 Compilation of iter

We first review the key ideas of evaluating an **iter** expression when there are no parameters [9], and then describe how to handle parameters.

Consider an **iter** expression **iter**(f, aggop) without any parameters. Similar to **split** expressions, the **iter** expression needs to maintain all possible cases of splitting the input stream. Though the number of such cases is only determined by f (thus not relevant to the input stream), how to succinctly represent each case is a challenge. Specifically, unlike a **split** expression which only splits the stream into a prefix and a suffix, an **iter** expression needs to split the stream into multiple substreams such that each one can be applied to the expression f. Therefore, naively maintaining all the states of f for each substream may use a large space as the stream grows. Fortunately, our choice of the aggregation operators allows us to summarizes the application of f to all substreams but the last one using the aggregated return value. For example, if aggop is **sum**, we only need to remember the aggregated sum on these substreams. Thus, we can represent a case as a pair (v, s_f), where v is the aggregated value and s_f is the state of f on evaluating the last substream.

In the general scenario with parameters, we simply maintain a guarded case as $((v, s_f), \phi)$. The update of a guarded case $((v, s_f), \phi)$ is shown in Algorithm 3. The evaluation of f given guarded cases is similar to that of a **split** expression.

Algorithm 3 update_iter((T, ϕ), packet)

1: suppose $T = (v, s_f)$
2: **for all** emitted (s'_f, ϕ') from update_f((s_f, ϕ), packet) **do**
3: **if** f is defined on s'_f **then**
4: let v' be the evaluated value of f on s'_f
5: emit $((v'', s^0_f), \phi')$, where v''=aggop (v, v') and s^0_f is the initial state of f
6: **end if**
7: emit $((v, s'_f), \phi')$
8: **end for**

5.4 Compilation of Aggregation

Now we describe the aggregation function aggop{f | T x}. Following the definition of the aggregation function, an aggregation function maintains guarded states (s_f,ϕ). To update the aggregation expression, we just need to update each guarded state using f's updating procedure. To illustrate the evaluation procedure, suppose the aggregation operator is **sum**. Given values for the parameters, we need to iterate through all guarded states (s_f,ϕ). If ϕ is satisfied, we evaluate f on s_f, say the return value is v; and count the number of values of x which satisfies ϕ, say it is k; we sum up $v * k$ into the aggregated sum. For other aggregation operators, the evaluation process works similarly.

5.5 Compilation of Stream Composition

Lastly, we discuss stream composition. Recall that stream composition allows to process the input stream using a function f, and then apply the second function g on the processed stream. Following its definition, each time a new packet arrives, we need to first process it with f, and the returned value (e.g. a filtered packet) is then fed into g.

Therefore, the stream composition expression f>>g needs to maintain guarded states $((s_f, s_g), \phi)$. Algorithm 4 shows the algorithm to update a guarded state following the intuition described above. The evaluation of the expression is similar to that of previous discussed expressions.

Algorithm 4 update_comp((S, ϕ), packet)

1: suppose $S = (s_f, s_g)$
2: **for all** emitted (s'_f, ϕ') from update_f((s_f, ϕ),packet) **do**
3: **if** f is defined on s'_f **then**
4: let ret be the returned packet of f on s'_f
5: emit $((s'_f, s'_g), \phi'')$, for all emitted (s'_g,ϕ'') from update_g((s_g, ϕ'), ret)
6: **else**
7: emit $((s'_f, s_g), \phi')$
8: **end if**
9: **end for**

6 IMPLEMENTATION

We have implemented a prototype of the NetQRE system in C++, which consists of two main components, namely the compiler for NetQRE, and the NetQRE runtime.

NetQRE Compiler. The NetQRE compiler implements the compilation algorithm described in §5. The compiler first generates a C++ program from an input NetQRE program, which is then compiled by the gcc compiler into executable. Our compiled program uses a tree data structure to represent predicates over parameters. The choice of tree is driven by its simplicity, ease of encoding, and lookup performance.

In addition to the basic compilation algorithm, we include additional optimizations to the compiler. First, we use the standard minimization algorithm to minimize the state machine for a regular expression. Second, for aggregation expressions with **sum** and **avg**, we use an incremental updating algorithm to update the expression: we maintain the running sum as the state of the aggregation expression, and we update the sum incrementally when the state of the inner expression is updated.

NetQRE Parallelization. The NetQRE compiler can optionally compile the NetQRE specification into parallelized implementations based on the instantiation of parameters in the NetQRE specification. For the example described in § 5.1, a packet with source IP address s can be processed by the $hash(s)$-th thread, which handles that instantiation of the parameter x.

NetQRE Runtime. The NetQRE runtime includes a packet capture agent implemented using the pcap library [28]. Each packet that arrives at the runtime is parsed, and then processed by the compiled NetQRE program as shown in Figure 1. Currently, our runtime supports actions that include sending alert events to a controller, and directly installing a rule on a switch. The NetQRE runtime is not specifically optimized for fast packet capture and processing, and as future work, we plan to explore the use of DPDK [24]. Our current implementation analyzes every packet, though orthogonal sampling techniques can be also explored in future.

7 EVALUATION

We evaluate our NetQRE prototype centered around answering three key questions. First, can the NetQRE language express a wide range of quantitative network monitoring applications in a concise and intuitive manner? Second, is the generated code efficient in terms of throughput and memory footprint? Third, can NetQRE be used in a real-time monitoring setting that provides rapid mitigation based on output of quantitative network monitoring applications? Unless otherwise specified, all our experiments are carried out on a cluster of commodity servers, each of which has 10-core 2.6GHz Xeon E5-2660V3. Each core has a 256KB L2 cache, and shared 25 MB L3 cache. The memory size is 64 GB. The OS is Ubuntu 14.04, and the kernel version is 4.2.0.

7.1 Expressiveness

We have implemented a set of quantitative network monitoring applications using NetQRE. The examples are drawn from a literature survey on quantitative network monitoring applications [13, 18, 35, 40, 41]. Table 1 summarizes the example applications in NetQRE that we use, as well as and the lines of code of each NetQRE program.

	LoC
Heavy Hitter (§4.1)	6
Super Spreader (§4.1)	2
Entropy Estimation [40]	6
Flow size dist. [18]	8
Traffic change detection [35]	10
Count traffic [40]	2
Completed flows (§4.2)	6
SYN flood detection (§4.2)	9
Slowloris detection (§4.2)	12
Lifetime of connection	8
Newly opened connection recently	11
# duplicated ACKs	5
# VoIP call	7
VoIP usage (§4.3)	18
Key word counting in emails	11
DNS tunnel detection [12]	4
DNS amplification [20]	4

Table 1: Example monitoring applications NetQRE supports.

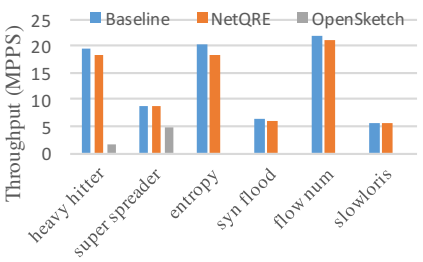
We make the following observations. First, NetQRE is able to express a large variety of quantitative network monitoring applications, ranging from flow-level traffic measurement to application-level quantitative monitoring. We validate the correctness of our applications by running them and comparing their output with hand-crafted implementations. Second, programming in NetQRE is concise. All example programs can be specified within 18 lines of code in NetQRE. This count includes commonly used filter functions as well as regular expressions, which can be built into a library for reference. As an interesting comparison, we encoded the VoIP call use case in Bro [33], a well-known intrusion detection system. Bro required 51 lines of code, as compared to only 7 in NetQRE. However, Bro is unable to easily support the VoIP usage case written in NetQRE. In particular, Bro cannot use patterns to separate packets into different VoIP sessions for the purposes of counting bandwidth consumption per session. This is not surprising as Bro's primary use is that of an intrusion detection system. We validated the correctness of our implementation on actual SIP traffic by comparing Bro's output against NetQRE's.

7.2 Performance

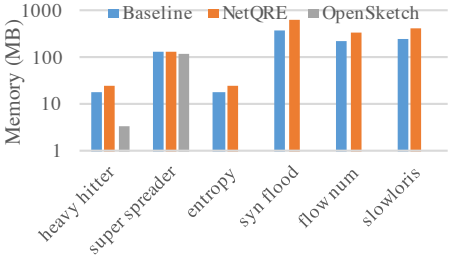
Our next set of experiments evaluate the performance of NetQRE's compiled implementations. NetQRE runs on a single machine based on the setup described earlier. We use a single core to measure NetQRE's performance. As a point of comparison, we compare each NetQRE program against an equivalent carefully hand-crafted C++ implementation. We note that all our hand-crafted implementations require at least 100 lines of code, and often times, require users to explicit manage internal state across packets – a programming task that NetQRE abstracts away.

Fig. 7 shows the performance of NetQRE implementations (NetQRE in the figure) compared with manually optimized implementations (Baseline in the figure) that we spent days optimizing. Our examples include heavy hitter, super spreader, entropy, SYN flood detection, completed flows count, and Slowloris use cases presented in §4. As workload, we use a one-minute CAIDA traffic trace [7] captured on a high-speed Internet backbone link, which contains 37 million packets (i.e., about 620k packets/sec). We replay the traffic trace to the implementations. Since the traffic trace does not contain payloads, we report the throughput in million packets per second (MPPS).

We make the following observations. First, the NetQRE implementations are able to achieve line-rate processing speed without being the bottleneck. In particular, for three out of six applications, the throughput of NetQRE implementations is about 20MPPS using a single thread. The average packet size of our input traffic trace is 888B. For a 10Gbps network, 1.4M packets of size 888B can be forwarded per second, which is well below the throughput achieved by NetQRE. The throughput can be further improved by parallelization (presented later in this section). Second, the compiled NetQRE implementation incurs negligible overhead compared with the manually optimized low-level implementation. The difference between the throughput of the compiled NetQRE implementation and that of the manual implementation is within 9% across all use cases we studied. Third, NetQRE incurs slightly increase (up to 60%) in terms



(a) Throughput



(b) Memory

Figure 7: Performance comparison.

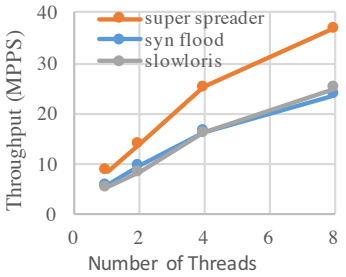


Figure 8: Throughput with parallelization.

of memory footprint (Fig. 7b) compared with the manually optimized implementation. However, the overall memory footprint of NetQRE remains low. We have also run the experiments on another CAIDA traffic [2] and the comparison result is similar.

Comparison with OpenSketch. In addition to comparisons with our manually optimized code, we also compare with OpenSketch [40], a popular traffic measurement tool based on sketching techniques. Given OpenSketch's limitation in handling stateful monitoring tasks, we only compare the performance of NetQRE implementations on the heavy hitter and super spreader examples with that of OpenSketch (in software). We use the same CAIDA traffic trace as before, and follow the default setting of the reference code of OpenSketch [4] for the two examples. The NetQRE compiled implementations significantly outperform OpenSketch implementations: The throughput of NetQRE compiled code is 11× and 1.8× as that of OpenSketch (Fig. 7a). As OpenSketch focuses on optimizing memory usage, it is not surprising that OpenSketch uses less memory than NetQRE implementations. Nevertheless, we observe that NetQRE uses only 11% more memory on the super spreader example than that of OpenSketch (Fig. 7b). Note that we use an open-source version of OpenSketch software that runs on a similar x86 machine as NetQRE, in order to have an "apples-to-apples" comparison on similar hardware. NetQRE may also exploit similar optimizations performed by OpenSketch in order to get higher performance on a NetFPGA. Exploiting a hardware platform such as NetFPGA in order to further improve NetQRE performance is an interesting avenue for future work.

Comparison with Bro. We further compare with Bro. Given the limitations of Bro in handling the complete VoIP use cases, we simplify the use case to simply counting the number of VoIP calls (as opposed to bandwidth consumption) per user. NetQRE compiled implementation can finish counting within 1 second, while Bro takes about 23 seconds for a SIP traffic trace containing 4338 VoIP calls. Both programs output the same results for this use case. We believe there are at least two reasons for Bro's slower performance. First, Bro is designed for intrusion detection, and not aimed at and optimized for monitoring tasks. Second, unlike NetQRE which compiles high-level code to efficient low-level code, Bro uses an interpreter to execute the script written in Bro's language, which could result in considerable overhead.

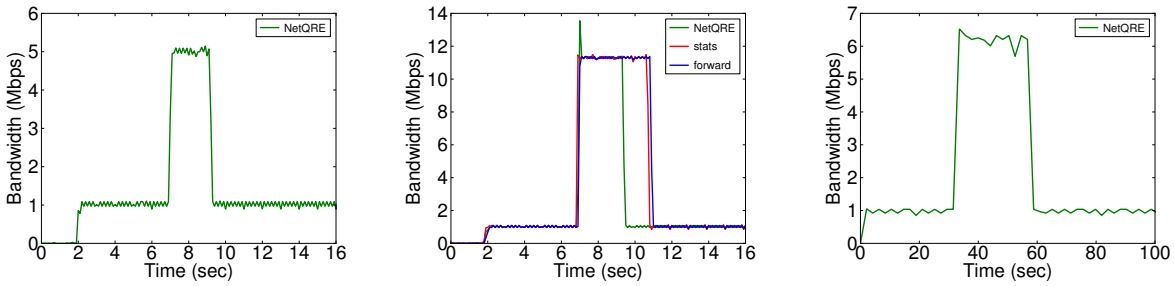
Parallelization speedup. NetQRE compiler automatically compiles parallelized implementations as described in §6. In the considered examples, parallelization involves sending flows of packets to different NetQRE instances running on different threads. Fig. 8 plots the throughput of the NetQRE implementations with different numbers of threads for the super spreader, syn flood and slowloris detection examples. In all examples, the NetQRE parallelized code is able to achieve at least 3.9× speedup with 8 threads. Note that linear speedup is not achieved as the traffic (in our finite trace) is not uniformly distributed across all 8 threads. This is an artifact of the trace data itself. When we include the overhead of the software load balancer that dispatches packet flows to different threads, the NetQRE implementations achieve at least 2.6× speedup for all examples. This overhead can be mitigated with a hardware load-balancer. Overall, we observe that NetQRE's throughput is more than sufficient to handle line-rate traffic, and the NetQRE runtime itself is not the bottleneck.

7.3 End-to-end Validation

In the final set of experiments, we validate the end-to-end use of NetQRE for enforcing quantitative network policies which applies actions to quantitative network monitoring programs. We set up a network of two clients C_1 and C_2, one server S, and one SDN switch that mirrors traffic to a NetQRE runtime running the SYN flood detector presented in §4.2. In addition, a NetQRE controller based on POX [29] controls the switch. The network is emulated using Mininet [25] with link bandwidth set to 100Mbps.

In the experiment, C_1 sends normal traffic to S using iperf at rate 1Mbps, and at the 7th second, C_2 starts SYN flood attack generated using our generator to the same server. The attack is detected by NetQRE, which generates an alert event to the controller, which subsequently installs a forwarding rule on the switch to block traffic from C_2. To illustrate the attack detection and blocking, Figure 9a (left figure) shows the bandwidth utilization (Mbps) on the server. We observe that NetQRE successfully blocks C_2's attacking traffic in real-time.

As a second experiment, using a similar setup, our NetQRE heavy hitter program (§4.1) running at a switch-side NetQRE runtime monitors the traffic over a sliding window of 5 seconds, and issues an alert to the controller when detecting a heavy hitter, which further blocks the traffic. As a point of comparison, we compare our approach against two alternatives: (1) sending all packets to the controller which runs an equivalent heavy hitter detection program, (2) monitoring the flow counters on the switch to detect heavy hitters, as considered in other languages [30] and systems [31]. In the second alternative, the controller reads the counter every 1



(a) SYN flood. (b) Heavy hitter. (c) VoIP.

Figure 9: Bandwidth utilization (Mbps) at the server.

second, and compute the bandwidth usage for each flow over a similar 5 seconds window.

Figure 9b (middle figure) plots the bandwidth utilization of the server. The above alternative solutions are labeled as *forward* and *stats* respectively. Compared with these two approaches, our proposed approach can detect the heavy hitter and further respond to it in a more timely fashion. Moreover, compared with the *forward* approach, we send significantly fewer traffic to the controller, and is a more scalable solution.

In our final experiment, we validate the VoIP use case first presented in our introduction. We use a synthetic VoIP traffic trace generated using SIPp [5] replay a VoIP call (with accompanying H.264 MPEG video) made from a single user via client C_2. We replay the traffic at 5Mbps, and run iperf on C_1 as the background traffic. Our NetQRE program enforces a policy that blocks a call after the user making the call exceeds a SIP bandwidth usage of 18.75MB (around 30 seconds). We configure the NetQRE program to send an alert to the NetQRE controller which then blocks the traffic. Figure 9c (right figure) plots the bandwidth utilization of the server. Again, this verifies that NetQRE can be used to intercept a SIP session, monitor usage on a per-user basis, and react to high usage in a timely fashion.

7.4 Summary of Evaluation

Revisiting the questions at the beginning of our evaluation, we observe that (1) NetQRE can express a wide range of quantitative monitoring applications with a few lines of code, (2) achieves line-rate throughput performance which is comparable to that of carefully hand-crafted optimized low-level code while significantly outperforms other measurement and IDS tools, and (3) can be used in an SDN setting to monitor network traffic and update switches in real-time in response to specified NetQRE policies.

8 DISCUSSION

In this section, we discuss some of NetQRE's limitations and also some future work.

Expressiveness. As a language focusing on monitoring policy, NetQRE cannot specify general network policies, such as service chaining policies and traffic engineering policies. As an interesting future work, it might be possible to extend NetQRE's actions in support of more complex policies.

Given NetQRE's basis on regular expressions, NetQRE is not suited for monitoring tasks that can not be specified using regular traffic patterns. Though this is a fundamental limitation of NetQRE's expressiveness, we believe that the high-level constructs in NetQRE provides a natural programming abstraction to write tasks with hierarchical regular structures, and NetQRE can specify a wide range of monitoring tasks as shown in §4.

Handling packet loss/reordering/retransmission. For the performance consideration, NetQRE does not buffer packets in the stream, but processes packets in a streaming fashion. Thus, if packets in the stream are lost, reordered, or retransmitted, the query specified in NetQRE might not be evaluated correctly, since the internal state may transition to a wrong one. Current design of NetQRE relies on the runtime to handle packet loss, reordering and retransmission.

Network-wide monitoring. For the similar reasons discussed above, current deployment model of NetQRE is based on a centralize fashion, in order to receive all (ordered) packet in the stream of interest. For example, a flow counting task has to be deployed at a place where all packets in a flow can traverse. This is not a fundamental limitation, as one can easily deploy NetQRE in a distributed fashion provided that the query is insensitive to packet reordering in the stream. We leave it as future work to study the decomposition of a centralized NetQRE program into distributed queries.

9 RELATED WORK

Quantitative regular expressions. NetQRE is inspired by quantitative regular expressions (QRE) [9], which provides a theoretical foundation for regular programming of data streams. NetQRE extends QRE in the following ways. First, NetQRE introduces parameters in support of queries handling unknown values (e.g. counting distinct sources). Second, NetQRE proposes aggregation operators to compute aggregated values across parameters. It is shown in §4 that both extensions are essential to specify a wide range of interesting network monitoring policies.

StreamQRE [27] is another recently proposed extension to QRE. NetQRE is different from StreamQRE in several aspects. First, NetQRE is specialized to networking applications, while StreamQRE focuses on database application. Second, StreamQRE allows relations as

types and supports relational operations such as join and key-based partitioning. These operations cannot be evaluated efficiently in a streaming fashion in general. NetQRE instead uses parameters and aggregation over parameters with a focus on efficient evaluation.

Intrusion detection systems and protocol analyzers. There are a number of key differences between NetQRE and intrusion detection systems (IDS) and protocol analyzers. First, systems such as Snort [34] allow regular expression matches on single packets, but is not perform regular expression matches that span multiple packets, let alone track sessions across packets. While Bro [33] supports aspects of NetQRE, the scripting language requires complex procedural programming where one has to maintain states and data structures, as opposed to NetQRE's declarative programming approach with high-level constructs over packet streams. As an IDS, Bro does not lend itself naturally to handle some quantitative monitoring use cases e.g. the VoIP usage example. While HILTI [37] uses Bro and offers abstract machine models for traffic analysis, it requires the operator to implement low-level state machines.

Domain specific languages in networking. There are several recent proposal on domain specific languages on networks, which includes Frenetic [21], Pyretic [30], NetKAT [10], Flowlog [32], Maple [38], Network Datalog [26]. To the best of our knowledge, none of these systems support monitoring queries beyond basic flow-level counters provided by OpenFlow.

Recently proposed SNAP [11] and Sonata [23] offer abstractions for traffic monitoring over packet streams, but in a more low-level procedural or SQL-like way. NetQRE offers a complementary abstraction on quantitative queries. It may be interesting to incorporate NetQRE with these languages in support for complex quantitative queries.

Streaming database languages. Database-style query languages [14, 16] provide SQL-like language support for running continuous queries over data streams. While there are constructs to do aggregation over sliding windows, they are designed with simple relational queries over packet headers or packet counts, and cannot handle the complex queries based on traffic patterns that NetQRE supports.

10 CONCLUSION

This paper presents NetQRE, a high-level declarative language and practical toolkit for quantitative network monitoring. NetQRE offers an intuitive programming model using regular expressions over streams of packets, and can naturally express flow-level as well as application-level quantitative policies. We developed a compilation algorithm that can compile NetQRE programs into efficient imperative programs. Our evaluation results demonstrate the expressiveness of NetQRE in support a wide range of quantitative network monitoring applications, its ability to match carefully optimized manually written low level code, and outperform equivalent implements in Bro and OpenSketch. A proof-of-concept scenario with an SDN controller and switches demonstrate NetQRE's ability to support timely and efficient enforcement of quantitative network policies.

ACKNOWLEDGEMENT

We would like to thank our shepherd Arjun Guha for his helpful comments. We also want to thank the anonymous reviewers for their insightful feedbacks on this work. This research was partially supported by NSF Expeditions in Computing award CCF 1138996, NSF CNS-1513679 and NSF CNS-1218066.

REFERENCES

[1] Application Layer Packet Classifier for Linux. *http://www.mcafee.com/us/products/network-security-platform.aspx*.
[2] CAIDA Traffic Trace. *https://data.caida.org/datasets/security/ddos-20070804/*.
[3] McAfee Network Security Platform. *http://l7-filter.sourceforge.net/*.
[4] OpenSketch reference code. *https://github.com/USC-NSL/opensketch*.
[5] SIPp. *http://sipp.sourceforge.net/*.
[6] SSL renegotiation DoS. https://www.ietf.org/mail-archive/web/tls/current/msg07553.html.
[7] Anonymized 2015 Internet Traces. *https://data.caida.org/datasets/passive-2015/*, 2015.
[8] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, volume 10, pages 19–19, 2010.
[9] Rajeev Alur, Dana Fisman, and Mukund Raghothaman. Regular Programming for Quantitative Properties of Data Streams. In *25th European Symposium on Programming*. ESOP, 2016.
[10] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. NetKAT: Semantic foundations for networks. In *Proceedings of the 41st annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 113–126. ACM, 2014.
[11] Mina Tahmasbi Arashloo, Yaron Koral, Michael Greenberg, Jennifer Rexford, and David Walker. Snap: Stateful network-wide abstractions for packet processing. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, pages 29–43, New York, NY, USA, 2016. ACM.
[12] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A declarative language for streaming network traffic analysis. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 19–19, Berkeley, CA, USA, 2012. USENIX Association.
[13] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly Detection: A Survey. *ACM computing surveys (CSUR)*, 41(3):15, 2009.
[14] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous Dataflow Processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.
[15] Benoit Claise. Cisco systems NetFlow services export version 9. 2004.
[16] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 647–651, New York, NY, USA, 2003. ACM.
[17] Luca Deri. Open source VoIP traffic monitoring. In *Proceedings of SANE*, volume 2006, 2006.
[18] Nick Duffield, Carsten Lund, and Mikkel Thorup. Estimating Flow Distributions from Sampled Flow Statistics. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 325–336. ACM, 2003.
[19] Cristian Estan and George Varghese. *New Directions in Traffic Measurement and Accounting*, volume 32. ACM, 2002.
[20] Seyed K. Fayaz, Yoshiaki Tobioka, Vyas Sekar, and Michael Bailey. Bohatei: Flexible and Elastic DDoS Defense. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 817–832, Washington, D.C., August 2015. USENIX Association.
[21] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ACM SIGPLAN Notices*, volume 46, pages 279–291. ACM, 2011.
[22] Pedro Garcia-Teodoro, J Diaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based Network Intrusion Detection: Techniques, Systems and Challenges. *computers & security*, 28(1):18–28, 2009.
[23] Arpit Gupta, Rüdiger Birkner, Marco Canini, Nick Feamster, Chris Mac-Stoker, and Walter Willinger. Network Monitoring As a Streaming Analytics Problem. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, HotNets '16, pages 106–112. ACM, 2016.
[24] DPDK Intel. Data Plane Development Kit. *http://dpdk.org*.
[25] Bob Lantz, Brandon Heller, and Nick McKeown. A Network in a Laptop: Rapid Prototyping for Software-defined Networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6. ACM, 2010.
[26] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking. *CACM*, 2009.
[27] Konstantinos Mamouras, Mukund Raghothaman, Rajeev Alur, Zachary G. Ives, and Sanjeev Khanna. StreamQRE: Modular Specification and Efficient Evaluation

of Quantitative Queries over Streaming Data. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2017.

[28] Steve McCanne, Craig Leres, and Van Jacobson. Libpcap. *http://www.tcpdump.org*, 1989.

[29] J Mccauley. POX: A Python-based Openflow Controller, 2014.

[30] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. Composing Software Defined Networks. In *NSDI*, pages 1–13, 2013.

[31] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. DREAM: dynamic resource allocation for software-defined measurement. In *Proceedings of the 2014 ACM conference on SIGCOMM*, pages 419–430. ACM, 2014.

[32] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. *NSDI, Apr*, 2014.

[33] Vern Paxson. Bro: A System for Detecting Network Intruders in Real-time. *Comput. Netw.*, 31(23-24):2435–2463, December 1999.

[34] Martin Roesch et al. Snort: Lightweight Intrusion Detection for Networks. In *LISA*, volume 99, pages 229–238, 1999.

[35] Vyas Sekar, Michael K Reiter, and Hui Zhang. Revisiting the case for a minimalist approach for network flow monitoring. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 328–341. ACM, 2010.

[36] David Senecal. Slow DoS on the rise. *https://blogs.akamai.com/2013/09/slow-dos-on-the-rise.html*.

[37] Robin Sommer, Matthias Vallentin, Lorenzo De Carli, and Vern Paxson. HILTI: An Abstract Execution Environment for Deep, Stateful Network Traffic Analysis. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, IMC '14, pages 461–474, New York, NY, USA, 2014. ACM.

[38] Andreas Voellmy, Junchang Wang, Y Richard Yang, Bryan Ford, and Paul Hudak. Maple: Simplifying SDN Programming using Algorithmic Policies. In *Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM*, pages 87–98. ACM, 2013.

[39] Mea Wang, Baochun Li, and Zongpeng Li. sFlow: Towards resource-efficient and agile service federation in service overlay networks. In *Distributed Computing Systems, 2004. Proceedings. 24th International Conference on*, pages 628–635. IEEE, 2004.

[40] Minlan Yu, Lavanya Jose, and Rui Miao. Software Defined Traffic Measurement with OpenSketch. In *NSDI*, volume 13, pages 29–42, 2013.

[41] Lihua Yuan, Chen-Nee Chuah, and Prasant Mohapatra. ProgME: Towards Programmable Network Measurement. *IEEE/ACM Transactions on Networking (TON)*, 19(1):115–128, 2011.