Retina: Analyzing 100 GbE Traffic on Commodity Hardware

Gerry Wan
Stanford University

Fengchen Gong
Stanford University

Tom Barbette
UCLouvain

Zakir Durumeric
Stanford University

ABSTRACT

As network speeds have increased to over 100 Gbps, operators and researchers have lost the ability to easily ask complex questions of reassembled and parsed network traffic. In this paper, we introduce Retina, a software framework that lets users analyze over 100 Gbps of real-world traffic on a single server with no specialized hardware. Retina supports running arbitrary user-defined analysis functions on a wide variety of extensible data representations ranging from raw packets to parsed application-layer handshakes. We introduce a novel filtering mechanism and subscription interface to safely and efficiently process high-speed traffic. Under the hood, Retina implements an efficient data pipeline that strategically discards unneeded traffic and defers expensive processing operations to preserve computation for complex analyses. We present the framework architecture, evaluate its performance on production traffic, and explore several applications. Our experiments show that Retina is capable of running sophisticated analyses at over 100 Gbps on a single commodity server and can support 5–100× higher traffic rates than existing solutions, dramatically reducing the effort to complete investigations on real-world networks.

CCS CONCEPTS

• **Networks** → **Network monitoring**; **Network measurement**;

KEYWORDS

Traffic Analysis, Internet Measurement, 100 GbE.

ACM Reference Format:
Gerry Wan, Fengchen Gong, Tom Barbette, and Zakir Durumeric. 2022. Retina: Analyzing 100 GbE Traffic on Commodity Hardware. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3544216.3544227

1 INTRODUCTION

Network operators and researchers routinely need to investigate production network traffic. However, over the past few years, network speeds have grown to 100+ Gbps, outpacing the performance of traditional analysis tools. As a result, seemingly simple, yet increasingly important questions that require analyzing reassembled flows or parsed application-layer data have become extremely difficult to answer (e.g., "What is the packet loss of traffic from YouTube?" or "How much traffic is sent unencrypted and why?").

Many solutions have been proposed for analyzing high-speed network traffic, but few have seen real-world adoption. This is in part because a trade-off remains between highly-optimized packet processing frameworks and the expressiveness and ease-of-use needed to quickly answer operational questions. Fast packet processors often lack the flow-level [8, 37, 45, 47, 64, 96] and application-level semantics [55, 67, 87] needed for common analysis tasks. Other systems, like high-performance intrusion detection systems and flow monitors, require niche hardware setups [43, 100], and/or are limited to fixed analysis functions [80, 94, 99, 100].

As a result, operators and researchers continue to attempt to scale older systems like Zeek (formerly Bro) [68] and Snort [73]—tools that expose expressive interfaces and high-level network abstractions [19, 22, 29, 30, 33, 40, 50–52, 82–84, 101]—or build custom analysis solutions using low-level libraries like DPDK and PF_RING [16, 38, 39, 80, 85, 94]. Unfortunately, older platforms scale inefficiently to today's high-speed environments, and custom-built solutions are time consuming and error prone to develop. For example, prior work estimates that Zeek requires over 100 CPU cores to process 100 Gbps traffic, assuming perfect scaling [76].

In this paper, we present *Retina*, a software framework that supports 100+ Gbps traffic analysis on a single server with no specialized hardware. Retina is designed to enable safely and easily answering complex questions about entire networks or uplinks rather than continually performing deep inspection of all network traffic. Retina dramatically reduces the effort required to understand high-speed networks by allowing users to subscribe to packets, reassembled connections, or parsed application-layer sessions using a simple filter and callback interface. For example, Retina can log the server names and ciphersuites of all TLS handshakes with ".com" domains on over 160 Gbps of network traffic using 8 cores, a standard server NIC, and 10 lines of Rust code. Retina's design and performance is based on several insights:

(1) In contrast to fast packet processing platforms and intrusion detection systems, analysis questions typically focus on only a subset of packets and flows (e.g., [17, 22, 29, 30, 101]). By optimizing the analysis pipeline to discard out-of-scope traffic as early and as often as possible, we can dramatically reduce the computation spent reassembling, parsing, and processing network traffic, bringing 100+ Gbps visibility within range. Retina decomposes user-friendly filters into each processing step and uses static code generation to compile filters into performant native assembly.
(2) Many burdensome development tasks like load balancing, connection tracking, stream reassembly, and application-layer parsing can be automated. However, while techniques for optimizing these individual components are well-studied, naïvely gluing them together results in redundant processing and data

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands
© 2022 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-9420-8/22/08…$15.00
https://doi.org/10.1145/3544216.3544227

transfer inefficiencies. For example, there is no reason to continue to buffer, copy, and reassemble TCP flows that contain encrypted data if users are only analyzing cryptographic handshakes. Retina's processing pipeline is built on the principle of lazy data reconstruction, where expensive operations are deferred until the framework is confident that the operation is needed to achieve the desired analysis result.

(3) Prior works have developed domain-specific languages and custom APIs for network analysis [13, 45, 55, 63, 67, 68]. While beneficial for security and performance, these interfaces inherently limit the analysis that can be performed. Inspired by Tock [58] and NetBricks [66], Retina introduces a Rust-based subscription programming model that allows users to write arbitrary analysis code on a diverse set of data representations at multiple layers of the networking stack. By leveraging Rust's compile-time guarantees, Retina can safely run complex analyses with minimal run-time overhead.

We deploy Retina in a production network using a single commodity server and 100 GbE NIC, and we demonstrate the simplicity with which users can answer diverse, meaningful questions about real-world traffic at 100 Gbps with *zero* packet loss. We are releasing Retina under the Apache 2.0 license to enable operators and researchers to more easily ask questions of high-speed networks at https://github.com/stanford-esrg/retina.

2 DESIGN GOALS

We aim to build an analysis solution that network operators and researchers can deploy and use in practice. To ensure we meet real-world demands, we start by analyzing prior research studies that employ passive traffic analysis (e.g., [17, 22, 29, 30, 51, 83, 95, 101]), consider our own research questions, and collect deployment requirements from operators, which we use to develop the following goals and design constraints:

Enable Complex Analysis. Our platform must support arbitrarily complex processing of individual packets, reassembled flows, and parsed application-layer sessions. Domain-specific query interfaces [13, 45, 63, 66] and specialized monitoring tools [23, 94, 99, 100] are performant but do not accommodate many real-world needs in practice. For instance, fixed function network analyzers fall short when applied to unanticipated research questions, such as uncovering nuanced anomalies in cryptographic operations or understanding previously unknown vulnerabilities [30]. In other domains, users should be able to run custom machine learning models on raw traffic [49], or extract user-defined features to infer application performance [16]. Our framework should also enable easily focusing on specific subsets of traffic (e.g., connections to YouTube and Netflix [17], or all SMTP sessions).

100+ Gbps Performance. Many networks that operators and researchers seek to analyze operate at 100+ Gbps [15]. Countless empirical studies have relied on historical traces from sub-10 GbE traffic collectors like the CAIDA equinix-chicago and equinix-sanjose vantage points [10, 48, 54, 71, 95]. However, for sustained 100+ Gbps traffic, it is difficult and often infeasible to store all packets to disk for after-the-fact analysis [18], especially for longitudinal studies that require weeks to months of data. In addition, IXPs and other deployment locations are frequently space and power constrained.

```rust
#[filter("tls.sni matches '.*\\.com$'")]
fn main() -> Result<()> {
    let cfg = load_config();
    let callback = |hs: TlsHandshake| {
        log::info!("TLS handshake with {} using {}",
            hs.sni(), hs.cipher());
    };
    let mut runtime = Runtime::new(cfg, filter, callback)?;
    Ok(runtime.run());
}
```

Figure 1: Example Traffic Subscription—Retina users subscribe to traffic using a filter and Rust callback. Here, we show a subscription for parsed TLS handshakes to all domains ending in ".com".

Several ISPs that our team is working with have been able to provide only 1–2 rack units for analysis equipment. Our system must support analyzing 100+ Gbps links in real-time using a single 1RU server and no external appliances.

Readily Deployable. Despite scalability issues [76, 100], traditional monitoring platforms like Zeek and Snort are popular due to their expressiveness, relative ease-of-use, and simple deployment requirements. While prior works have improved packet processing performance by offloading CPU intensive tasks to specialized hardware like SmartNICs [35, 62, 79], FPGAs [35, 74, 100] and GPUs [43, 87], these devices require elaborate development cycles that preclude many operators and researchers [5]. Our framework must be readily deployable on commonly available hardware (e.g., "dumb" NICs), run in a standard software environment, and remain easy to use without the need to learn specialized skills or programming paradigms.

Security. Real-world network traffic can be unpredictable and malicious [26, 68]. Vulnerabilities in processing code can be remotely exploited to expose sensitive network communications. Unfortunately, due to performance considerations, most monitoring tools are written in low-level languages like C and C++, which have allowed memory safety errors escalate into vulnerabilities on production networks [24, 25, 27]. Our system needs to safely perform internal framework operations (e.g., packet parsing and stream reassembly) and enforce memory safety within user-defined analysis functions such that individual experiments do not place users at unnecessary risk.

While there has been much progress in each individual area, we find that existing systems fulfill only a subset of our requirements. Motivated by these observations and constraints, we set out to develop a new system capable of meeting *all* of the above objectives.

3 RETINA ANALYSIS FRAMEWORK

We introduce Retina, a software framework that enables operators and researchers to ask complex questions of high-speed traffic by *subscribing* to filtered, reassembled, and parsed network data. For example, Retina lets users subscribe to all TLS handshakes with domains ending in ".com" and log the server names and ciphersuites in under 10 lines of Rust code (Figure 1). In this scenario, the framework automatically handles packet capture, load balancing, connection tracking, TCP reassembly, TLS handshake parsing, and flow filtering. Retina depends on only commonly available hardware primitives and is implemented in Rust, which enables the framework to isolate and secure user-defined analysis functions.

3.1 System Overview

We observe that most analysis questions require fully processing only a subset of Internet tra c. By deferring computationally expensive operations and discarding extraneous tra c at each processing step, we can dramatically reduce unnecessary computation such that it is possible to answer questions on high-speed links without restricting our analysis language or requiring specialized hardware.

Retina users subscribe to network tra c by specifying a lter and a callback, which Retina compiles into a work-conserving processing pipeline that (1) eagerly discards out-of-scope tra c, and (2) lazily reconstructs relevant network data (deferring expensive operations until the framework determines they are necessary). As shown in Figure 2, Retina decomposes subscription lters into multiple layers, each of which hierarchically reduces the tra c sent to subsequent processing stages. As packets arrive, Retina reconstructs larger and more relevant segments of data up through each layer while explicitly deferring data transfers, reassembly, and application-layer parsing. This approach e ciently retains desired tra c, while minimizing the work spent processing data that will eventually be ltered out. Retina abstracts away the lter decomposition and data reconstruction steps, allowing users to focus on core analysis logic without worrying about hardware speci cs or low-level optimizations. We detail lter decomposition (Figure 2, left) in Section 4 and data processing (Figure 2, right) in Section 5.

3.2 Subscription Programming Model

Users interact with Retina by specifying a tra c lter and a Rust callback function that receives network data in one of several extensible representations. Filters and data types registered with the callback are independent, enabling users to subscribe to desired tra c while choosing the data representation most suitable for their analysis (e.g., raw IP packets associated with TLS handshakes with net ix.com).

3.2.1 Accessible Filters. Filters let users easily focus on tra c of interest, while simultaneously allowing Retina to quickly discard out-of-scope tra c to save computation. Our lter language is designed to be expressive, semantically simple, and familiar to users. While not identical, the syntax is inspired by Wireshark Display Filters [90] and the Camus subscription language [56]. Filters are de ned as a logical expression of constraints on attributes of the input data. Each constraint is either a binary predicate that compares the value of an entity's attribute with a constant, or a unary predicate that matches against the entity itself. We show the lter syntax in Table 1, along with several examples.

3.2.2 Expressive Callbacks. Rather than introducing a domain-speci c analysis language that potentially restricts functionality, Retina presents an expressive callback interface in Rust that allows users to safely write arbitrary analysis code in a standard software development environment. Callbacks are registered with a subscribable type that provides access to network data at one of three abstraction levels corresponding to layers of the OSI model:

- **Raw Packets (L2 3):** Raw Ethernet frames or IP packets are provided in the order received on the network.

Figure 2: Framework Architecture Retina e ciently processes tra c on high-speed networks by decomposing lters into multiple layers that hierarchically reduce tra c sent to subsequent processing stages. At runtime, data is reconstructed lazily to minimize computation on tra c that will be ltered out.

Protocols	h	generic
Fields	f	generic
RHS values	r	int|string|ipv4|ipv6|int_range
Predicates	p	h|h.f = r|h.f < r|h.f in r|h.f matches r|...
Expressions	e	p|e$_1$ and e$_2$|e$_1$ or e$_2$|(e)|...
Examples		ipv4.ttl > 64 ipv4 and (tls or ssh) ipv6.addr in 3::b/125 and tcp http.user_agent matches ' Firefox '

Table 1: Filter Syntax Retina employs a Wireshark-inspired lter syntax that helps users control the type of tra c processed. In contrast to prior systems, identi ers are are not hard-coded into the framework, but rather are exposed by extensible protocol modules.

Reassembled Connections (L4): Data from connections are reassembled in sequence and delivered as fully reconstructed byte-streams or connection records.

Application-Layer Sessions (L5–7): Reassembled streams are further parsed into protocol messages and presented to the user as parsed application-layer data.

Our data model contrasts those used by many middlebox development frameworks and intrusion detection systems, which expose event-based interfaces that allow users to manipulate flows or respond to pre-defined network behaviors (e.g., TCP state transition, triple duplicate ACK, etc.) [36, 55, 73]. We observe that this model complicates many research and analysis tasks by requiring the user to reason about complex network interactions and protocol-specific dynamics that are better abstracted away by the framework. Retina shifts the responsibility of managing protocol events and other complexities from the user to the framework itself.

For example, we provide the L7 session abstraction needed to make analyzing TLS handshakes as simple as writing a function that takes a parsed TLS handshake as an argument (Figure 1). Fine-grained L3 analyses (e.g., network classification with packet binaries [49]) can also be performed by subscribing to raw packets while filtering for connections to specific domains. We note that even if Retina does not directly implement a particular data abstraction, subscribing to raw packets is equivalent to using a high-speed packet analyzer with more advanced filtering capabilities than `tcpdump`. Users can still choose to run arbitrary analyses using raw packets, or alternatively, extend the framework with new a protocol or subscription module.

3.3 Framework Extensibility

Retina's filter language and subscription data model are extensible. Unlike existing filtering techniques [61, 90] that have a fixed set of allowable primitives hard-wired into the framework, Retina maps filter predicates to identifiers exposed by a set of extensible protocol modules. Each module defines how to parse and reconstruct incoming traffic and exposes a set of protocol-specific fields that the framework can use for filtering. Users can also create new subscribable types (e.g., SSH handshake transcript, fully reconstructed byte-stream, etc.) by implementing a new subscription module or by modifying an existing one. We detail how modules are defined in Appendix A.

4 FILTER DECOMPOSITION

Retina's traffic filters are not merely a convenience they dramatically increase performance by efficiently discarding unneeded traffic and connection state as early and as often as possible. Retina applies filters by decomposing a user-specified filter expression into four components: (1) a NIC-compatible hardware packet filter, (2) a more expressive software packet filter, (3) a connection filter, and (4) an application-layer session filter. Each filter hierarchically matches on incoming packets, connections, and sessions, allowing Retina to drop out-of-scope traffic at every processing step. Hardware filters are limited in complexity, but winnow down traffic at zero CPU cost, while the software filters work in tandem to reduce computation in subsequent stages of the data pipeline. At compile time, Retina transforms each software sub-filter into a fixed sequence of conditionals that is statically verified by the Rust compiler for memory safety and correctness before being inlined at its respective processing layer. This technique bakes the filter logic into the application binary as if it were hard-coded by a developer, avoiding the overhead of interpreting filters at runtime.

4.1 Generating Multi-Layer Filters

We represent filter expressions as a predicate trie, of which input data must match at least one root-to-leaf path to satisfy the filter. This intermediate representation is similar to the control flow graph model used in BPF [61] and NNStat [14], but is modified to facilitate multi-layer filter decomposition and static code generation. For instance, all nodes are restricted to a single parent to eliminate ambiguity at compile time when generating the sub-filters. This structure also maps each node directly to a conditional statement in Rust (rather than compiling to a limited instruction set [61]), which increases flexibility to support new protocols and filterable fields.

To build the predicate trie, Retina first transforms the filter expression into disjunctive normal form, creating a set of patterns that each consist of a conjunction of atomic predicates. Using metadata from protocol modules that dictate how headers are encapsulated, Retina expands and reorders each pattern such that packet headers and application-layer protocols are parsed in sequence (due to the potential for variable length headers). During trie construction, we tag predicate nodes to indicate whether they apply to packet, connection, or session predicates, and whether they terminate a sub-filter (i.e., is a leaf node in the sub-tree). Once the predicate trie has been constructed, we group nodes into their respective sub-filters and perform an optimization pass to eliminate redundant branches to speed up matching. We show an example in Figure 3.

Hardware Packet Filter. Retina automatically installs an on-NIC hardware filter to decrease load on the CPU. For narrow filters, this can dramatically reduce the volume of traffic reaching software with no external equipment. Most commodity NICs are capable of some degree of flow filtering, but vary in terms of supported protocols, operands, and complexity. As a result, manually programming hardware rules requires significant effort from the user to understand limitations of different NICs and vendor specific quirks [57]. Our framework abstracts this process away by expanding each filter predicate into a hardware flow rule and dynamically validating its compatibility with the user's device. Retina caches each validated predicate and recombines them into a set of hardware rules that are at least as broad as the subscription filter. We show the hardware filter in Figure 3 for a NIC that does not support the `>=` operand in the predicate `tcp.port >= 100`. The resulting hardware packet filter permits all TCP packets, but relies on the software packet filter to implement the remaining filter logic.

Software Packet Filter. To generate the software packet filter, we leverage *procedural macros*, a metaprogramming feature of Rust that allows us to modify the syntax tree of the application source code at compile time. As shown in Figure 3, each unary predicate is converted to an `if let` statement[1] to parse packet headers, and each binary predicate maps to an `if` statement to match on fields.

[1] In Rust, `if let` matches an expression to a pattern and gives access to the matched value inside the body of the conditional, whereas a normal `if` statement simply runs when the condition is true.

Figure 3: Filter Decomposition and Code Generation We show how the filter (ipv4 and tcp.port >= 100 and tls.sni ~' netflix ') or http is decomposed and executed. Retina parses the specified filter expression into a predicate trie representation, and generates (1) a set of NIC-compatible flow rules, (2) a more expressive packet filter, (3) a connection filter, and (4) an application-layer session filter. Flow rules are installed on the NIC, while the software filter functions are compiled and inlined at their respective processing layers. Native code generation can result in up to high[3]er throughputs over runtime interpretation, depending on the filter and traffic (Appendix B). Note: we truncate full path names and other expansions for readability.

Retina uses the relevant protocol modules (Section 3.3) to generate the actual parsing code and calls the appropriate accessor methods defined in each module to evaluate the predicate. If a packet filter pattern is non-terminating (i.e., there are subsequent connection or session-level predicates that must be applied downstream), we tag any matching packets with the ID of the last matched node in the trie to prevent redundant trie traversal in ensuing filters. The generated filter function is verified by the Rust compiler to be memory safe and syntactically correct, then inlined in the processing code immediately after packet capture.

Connection Filter. The connection filter evaluates predicates on connections (e.g., L7 protocol), and serves to eliminate excess connection state and stop unnecessary stream reassembly and protocol parsing. This filter is applied as soon as enough data has been observed to identify the L7 protocol but before full L7 parsing occurs. Like the packet filter, the connection filter maps predicate nodes to a sequence of conditionals according to the predicate trie, and returns a filter result tagged with the deepest matched predicate node ID. In Figure 3, we show the generated connection filter code that will discard all non-HTTP and non-TLS connections, as well as TLS connections not destined for port 100 or greater.

Application-Layer Session Filter. The session filter evaluates predicates on application-layer elements (e.g., TLS version), working in conjunction with the connection filter to reduce memory and CPU cycles by discarding L7 data (or entire connections) that the user is not interested in. Retina applies this filter when a session is fully parsed by branching from the deepest predicate node ID matched in the pattern so far. If the connection had already matched a terminal predicate (e.g., nodes 3 and 9 in Figure 3), the session filter simply returns a successful match. For performance, Retina automatically declares lazily evaluated static variables for executing regular expressions (e.g., tls.sni ~' netflix '). This ensures that all regular expressions in the filter are compiled only once rather than every time the filter is applied.

We emphasize that Retina filters are generated at compile time so that Retina does not need to consider how to apply the filter against input traffic at run time. This enables Retina to safely and efficiently discard out-of-scope traffic, dramatically reducing computational burden at each processing step.

5 DATA RECONSTRUCTION

In this section, we describe the design of Retina's runtime processing pipeline, depicted on the right-hand side of Figure 2. At a high level, Retina receives raw packets from the network and builds increasingly larger segments of data up through each filtering layer before executing the callback with the user's subscription data. Because Retina is subscription-aware, it is able to lazily reconstruct network traffic. That is, we avoid redundant copying, reassembling, and parsing of data destined for the user until we are sure that it fulfills the desired subscription. This approach minimizes any wasted computation on traffic that will be discarded by later filters.

5.1 Stateless Packet Processing

Retina is designed to support modern multi-core architectures and leverages commodity NIC hardware for initial packet filtering and load balancing at zero CPU cost. Ingress packets are processed by the hardware filter before being distributed among cores via symmetric Receive Side Scaling (RSS) [53]. RSS is a well-known technique that load balances traffic with per-connection consistency, avoiding cross-core data sharing and enabling near-linear scaling by increasing core counts. RSS is implemented by most commodity NICs [7] and works by hashing packet headers and dispatching them to receive queues based on a redirection table lookup. While symmetric RSS does not necessarily achieve perfect load balancing, we find that on real-world traffic, the number of flows tends to be well distributed among cores. More advanced load balancing techniques (e.g., [7, 55]) can yield further scaling improvements but are orthogonal to our work.

Since modern operating system kernels are unable to sustain 100 Gbps ingress traffic rates [8], we use kernel-bypass (specifically, DPDK [37]) to deliver raw packets from the NIC directly to user-space memory. Retina assigns one CPU core per receive queue, and each core polls its associated descriptor ring for packets. Packets

are immediately filtered again in software to efficiently discard those that are unable to be filtered in hardware (e.g., due to unsupported header fields or operands). If the user is subscribed to packets with no need for connection or session filtering, the callback is directly invoked at this stage to bypass further processing. Otherwise, packets are forwarded to the Connection Tracker, which continues processing them into larger and more relevant segments for further analysis.

5.2 Stateful Connection Processing

Retina determines if stateful processing is needed based on the subscription data type and the result of the packet filter. Connection and application-level traffic subscriptions, as well as non-terminating packet filter matches (i.e., the packet only partially matched a filter pattern), require stateful processing to reassemble connections or parse application-layer data. This is critical for a broad range of analysis tasks, ranging from identifying patterns across packet boundaries to investigating anomalies in cryptographic handshakes.

Connection Tracking. Retina uses per-core hash tables to manage connection state, an approach that has been shown to scale independently of the offered traffic load [41]. Each core tracks only connections received from symmetric RSS, allowing tables to be maintained independently and without cross-core synchronization.

Connections often fail to terminate gracefully on real-world networks [27], in part due to large-scale SYN scanning (e.g., ZMap scans [31]). Around 65% of TCP connections observed on our network consist of a single unanswered SYN, causing new connections to arrive at a far higher rate than that of connection establishment or termination. To prevent memory exhaustion from inactive connections, we build upon a timer wheel [86] mechanism adapted to accommodate modern network behaviors. Recent work as shown that timer wheel based flow deletion scales better than alternative techniques without adding complexity to hash table insertions [41].

Retina employs hierarchical timer wheels based on empirical observations: a short connection establishment timeout to expire unanswered SYNs, and a longer inactivity timeout to remove established inactive connections. Unanswered SYNs are treated as proper connections and can be analyzed in the same manner as other connections. Naturally, there is a trade-off between timeout length and analysis depth, as the framework may prematurely remove connections that have long intervals between packets. Our default values of 5 seconds and 5 minutes, respectively, are chosen conservatively based on the 99th percentile inactivity intervals measured on our network (1 second from SYN to SYN-ACK, 163 seconds between consecutive packets), though all timeouts are configurable to accommodate different network environments and subscription types.

Light-Weight Stream Reassembly. Traditional approaches to TCP reassembly involve allocating data buffers to hold packet payloads as they arrive from the network [46, 67, 92]. While this design provides a convenient stream abstraction for applications that require access to in-order bytes, it is wasteful in situations where fully reconstructed byte-streams are not needed for every connection, or only partially needed for some connections. For example, if a user subscribes to TLS byte-streams with domains ending in `.com`, it is wasteful to allocate stream buffers and copy bytes over until the session filter can verify that the server name indeed ends in `.com`. Furthermore, prior work has shown that the vast majority of TCP packets arrive in sequence order [26, 100], demonstrating that large reassembled data buffers are unnecessary to support most real-world networks. Indeed, our measurements of a large university network show that 94% of flows with at least two packets arrive completely in order, while the median number of packet arrivals it takes to fill a hole in a TCP byte-stream is 1.

Since Retina's behavior is derived from the subscription, we can tightly interlace the processing logic to conserve work for traffic that actually pertains to the user. Rather than reconstructing byte-streams by copying payloads into a separate receive buffer, Retina only reorders packets as they arrive. We track the next expected sequence number in each flow, and immediately send packets that match the expected sequence downstream for further processing. Out-of-order packets are stored by reference in a configurable-length ring-buffer, which is flushed when the next expected segment arrives. By default, we use 500 packets as the maximum out-of-order capacity, which can be adjusted based on available memory and expected packet loss on the network. This fast, common-case approach to stream-reassembly avoids unnecessary computation and memory on streams that do not fulfill the user's subscription, allowing most packets to simply pass through the stream reassembler.

Application-layer Parsing. Retina parses connections according to a subscription-specific state machine derived from both the filter and the subscribed data type, an approach that prevents wasteful computation from parsing or reassembling data no longer needed for the subscription. For example, if a user is interested in analyzing raw packets associated with HTTP connections, we can stop reordering flows after identifying the protocol as it is sufficient to simply track the connection and deliver each packet. If a user is subscribed to TLS handshakes, we can even stop processing traffic mid-connection as there is no reason to continue tracking the encrypted TCP connection after the initial handshake.

All connections transition through four possible states (Figure 4), which indicate whether Retina should **P**robe the connection for protocol messages, **P**arse the application-layer protocol, **T**rack the connection without parsing, or **D**elete the entire connection from the state table. The connection filter and session filter are choice pseudostates that split transitions according to the output of the filter, and determine when connections can stop being parsed, reassembled, or tracked altogether. Retina automatically derives the state transitions according to parsing behavior defined in protocol modules, as well as output behavior defined in subscription modules. This design avoids wasting memory and CPU cycles on connections that no longer fulfill the subscription, enabling Retina to better serve connections that do require additional processing.

To illustrate this, we show the state diagrams for two example subscriptions in Figure 4: one for raw packets in HTTP connections, and the other for transcripts of TLS handshakes with `.com` domains. In the first example (Figure 4a), Retina buffers incoming packets while probing for HTTP messages. Connections that match the HTTP connection filter are checked against the session filter (which in this case defaults to `True` since the connection filter is terminal, recall from Section 4.1). On a filter match, Retina runs the callback on any buffered packets and transitions the connection to

(a) Packets in HTTP connections (b) TLS handshakes with domains ending in .com

Figure 4: Subscription-Speci c Connection State Diagrams Retina automatically derives state transitions to e ciently parse connections according to the lter and subscription type. Each state dictates the parsing behavior for the connection, and dashed lines indicate opportunities to discard connections before they naturally terminate.

Track, allowing it to bypass parsing on remaining packets in the connection. By comparison, in Figure 4b, Retina probes for TLS protocol messages and internally manages TLS state. Connections that fail to match the TLS connection lter (which can happen as early as the Client Hello) are immediately dropped, and likewise for those that do not match the session lter (tls.sni ~'.*\\.com$'). On a full lter match, Retina runs the callback on the parsed handshake transcript and removes the connection before it naturally terminates. We emphasize that these state transitions are automatically derived by the framework to halt redundant parsing and remove connection state where possible, allowing Retina to accommodate large amounts of tra c while remaining exible over di erent subscription types.

5.3 Callback Execution

Callbacks are closures that de ne the user-level processing logic to be performed against subscribed network data. Retina transfers ownership[2] of the data from the core framework to the callback, providing the user with exibility to write arbitrary analysis functions in a general purpose programming language. By building on Rust, Retina provides memory safety guarantees that prevent user-de ned code from leaking memory or crashing the framework. Additionally, it allows users to easily take advantage of Rust's extensive ecosystem of third-party libraries [21], all within a uni ed processing environment. This signi cantly reduces development overhead and allows Retina to run user code alongside the data collection infrastructure without context switching or placing serialized data in an external queue.

Callbacks are implemented inline rather than in a separate thread, which enables e cient execution without cross-core communication. As with any real-time system, however, callbacks cannot perform arbitrarily long computation. To some extent, expensive operations can be absorbed by packet receive queues without stalling the processing pipeline. Retina does provide logs and real-time monitoring of packet loss, throughput, and memory usage that can be used as feedback to adjust the lter or improve callback e ciency if needed. For instance, if an application is writing data to disk on each callback but is not able to keep up with the ingress tra c rate (i.e., incurs sustained non-zero packet loss), the user may consider using a bu ered writer, increasing the number of cores, or even narrowing the lter. We evaluate Retina's performance with callbacks of varying complexity in Section 6.1, but leave support for alternative execution models to future work.

[2]Ownership is a unique feature of Rust that enables the compiler to ensure memory safety at compile time.

6 PERFORMANCE EVALUATION

In this section, we evaluate Retina's performance, and demonstrate its ability to process over 100 Gbps of real-world tra c for a variety of analysis applications on a single commodity server. We show that Retina outperforms popular network monitoring tools by sustaining 5–100 times higher tra c rates with zero packet loss, and can be used for long-term analysis of high volume networks without exhausting memory.

Hardware Setup. We perform our evaluations on a dual Xeon Gold 6248R 3GHz CPU (24 cores), with 384 GB of memory and two 100 GbE Mellanox ConnectX-5 NICs. The ConnectX-5 is a standard 100 GbE "dumb" NIC frequently used in prior work [7, 34, 56, 57, 72], but we note that Retina supports other DPDK-compatible NICs as well (e.g., Intel E810). Both NICs receive a 100 GbE link carrying real-time tra c from a large university network, with packets duplicated across the two links such that we receive double the regular tra c. We use this setup to stress test Retina beyond 100 Gbps, as the live rate on a single link rarely exceeds 75 Gbps at peak times during the day. Despite using a dual-socket server, we restrict ourselves to using cores from only one CPU. We keep hyper-threading enabled but only use one thread per physical core.

Monitoring Environment. Retina targets high-volume, real-world networks. As such, we run our evaluations on live tra c from a large university campus network, unless otherwise stated. Due to storage limitations, it is infeasible to capture a large enough packet trace to properly evaluate Retina against realistic workloads for more than a few seconds at 100 Gbps. Open-source tra c generators like DPDK Pktgen [89] and TRex [78] are unable to synthetically generate a realistic distribution of ows with proper payloads at line rate either. While using real tra c exactly matches our intended use-case, it is inherently inconsistent and di cult to control for experiments. We combat this by running experiments temporally close to each other and running multiple trials where possible. We summarize several features of our network tra c in Appendix C.

Ethical Considerations. As part of our evaluation, we measured whether Retina was capable of processing high-speed tra c on our campus network. This analysis was approved by our university's privacy and security o ce. We did not investigate human behavior, surface or investigate any individual ows or IP addresses, or store any tra c or individual records to disk. We restricted all analysis to aggregate network statistics directly output by Retina. The tap setup only saw a copy of tra c to prevent impact on network users

(a) Raw Packets (b) TCP Connection Records (c) TLS Handshakes

Figure 5: Zero Packet Loss Processing Throughput We use CPU cycles per callback as a proxy for callback complexity. For all three subscription types, Retina is able to support more than 160 Gbps ingress rates on a single multi-core CPU. We also report the approximate packets-per-second using the average packet size on our network (895 B). We emphasize that variations above 100 Gbps are due to the ingress tra c rate at the time of the experiment (which we are unable to control), and should therefore be interpreted as at least 100 Gbps .

in the event of a system failure. The server used was deployed in partnership with our campus networking and security teams using the same security procedures as other production network equipment in order to ensure that we did not increase the attack surface of users on campus.

6.1 End-to-End Throughput

In this section, we evaluate Retina's end-to-end processing throughput. Since we accommodate a wide variety of use cases, there is signi cant variation that can occur depending on the lter, the data representation, and the complexity of the callback. We rst evaluate Retina over three subscription types: raw unordered packets, TCP connection records, and parsed TLS handshakes, each of which represents one of the three supported data abstraction levels (Section 3.2.2). To approximate callback complexity, we busy loop for a set number of CPU cycles within the callback function.

Setup. Since we cannot control the ingress tra c rate on the observed network, we adjust the rate of tra c that reaches the processing cores by modifying the NIC's RSS redirection table to direct random four-tuples to a separate sink core that drops all received packets. This sink core is not central to the Retina framework, but allows us to measure the e ective ingress rate at the CPU without sacri cing ow consistency. This technique can optionally be used to sample tra c for extremely heavy workloads, but none of our applications have required us to do so (Section 7). Unfortunately, ow sampling cannot be enabled with hardware ow rules, so we disable hardware ltering to give a lower bound on the maximum throughputs measured in this experiment.

For each trial, we start at the full ingress tra c rate and slowly increase the percentage of ows dropped by the NIC until we observe zero packet loss for ve minutes (enough to reach steady-state given our default inactivity timeouts). We repeat trials until either (1) the maximum zero-loss processing throughput is lower than the total ingress tra c rate, or (2) the throughput saturates the ingress link at above 100 Gbps. The dynamic nature of the input tra c makes it hard to nd a stable comparison point between con gurations that saturate the ingress rate, so we choose 100 Gbps as the target threshold. As such, any variations above this threshold should be viewed as an artifact of the tra c rate at the time of the experiment rather than as a function of the number of cores or the complexity of the callback.

Results. Figure 5 reports the maximum throughputs sustained with no packet drops. As a baseline, Retina is able to capture all packets on the network and run an empty callback with just two cores at over 162 Gbps. Unsurprisingly, the maximum throughput decreases as more time is spent analyzing each packet, with per-packet workloads that exceed 100K cycles incurring non-zero loss. Using just 8 cores, Retina is able to reassemble and process all TCP connection records from at least 127 Gbps of input tra c, and with 16 cores for more complex (100K+ cycles) processing. For context, logging connection records to a shared le takes around 12K cycles on our platform. Naturally, there are far fewer connections than individual packets, enabling higher maximum tra c rates for large per-callback workloads. When subscribed to TCP connection records, Retina proactively drops all non-TCP packets (30% of all packets on our network, Appendix C) and bypasses redundant payload processing.

Analyzing all TLS handshakes can be achieved using just 8 cores with ingress tra c rates exceeding 160 Gbps, even for heavy per-handshake workloads. Although subscriptions for TLS handshakes require application-layer parsing, Retina's ltering system automatically discards non-TLS connections as soon as they are detected and drops remaining packets in TLS streams that arrive after the handshake. By design, no CPU cycles are wasted reassembling or parsing packets that will never ful ll the subscription.

6.2 Comparison with Optimized IDSes

We compare Retina's performance to Zeek [68], Snort [73], and Suricata [36], popular intrusion detection and monitoring platforms that are frequently employed to ask research and operational questions on networks. These are open-source tools that many users are familiar with, and are some of the few existing platforms that can be adapted to various analysis tasks with relative ease. While not direct replacements (Retina is not an IDS), they provide similar support for stream reassembly and application-layer parsing, unlike purely packet-oriented frameworks like FastClick [8], BESS [46], VPP [6],