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

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

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 traffic. By deferring computationally expensive operations and discarding extraneous traffic 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 traffic by specifying a *filter* and a *callback*, which Retina compiles into a work-conserving processing pipeline that (1) eagerly discards out-of-scope traffic, 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 filters into multiple layers, each of which hierarchically reduces the traffic 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 efficiently retains desired traffic, while minimizing the work spent processing data that will eventually be filtered out. Retina abstracts away the filter decomposition and data reconstruction steps, allowing users to focus on core analysis logic without worrying about hardware specifics or low-level optimizations. We detail filter 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 traffic filter 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 traffic while choosing the data representation most suitable for their analysis (e.g., raw IP packets associated with TLS handshakes with "netflix.com").

3.2.1 Accessible Filters. Filters let users easily focus on traffic of interest, while simultaneously allowing Retina to quickly discard out-of-scope traffic to save computation. Our filter 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 defined 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 filter syntax in Table 1, along with several examples.

3.2.2 Expressive Callbacks. Rather than introducing a domain-specific 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.

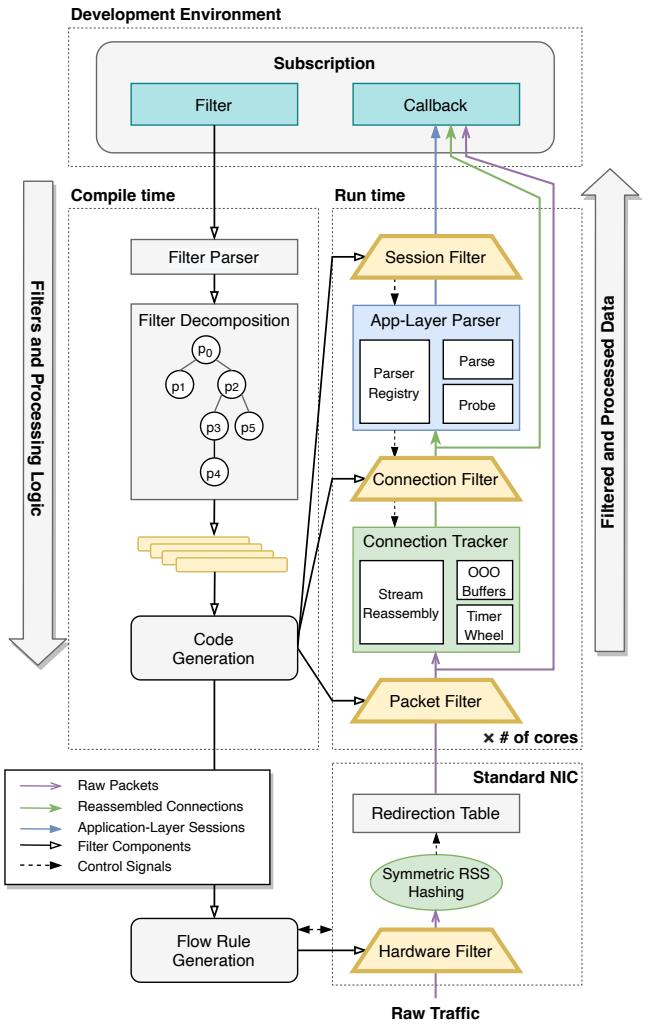


Figure 2: Framework Architecture—Retina efficiently processes traffic on high-speed networks by decomposing filters into multiple layers that hierarchically reduce traffic sent to subsequent processing stages. At runtime, data is reconstructed lazily to minimize computation on traffic that will be filtered 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	$p \mid e_1$ `and` $e_2 \mid e_1$ `or` $e_2 \mid$ `(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 filter syntax that helps users control the type of traffic processed. In contrast to prior systems, identifiers 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.

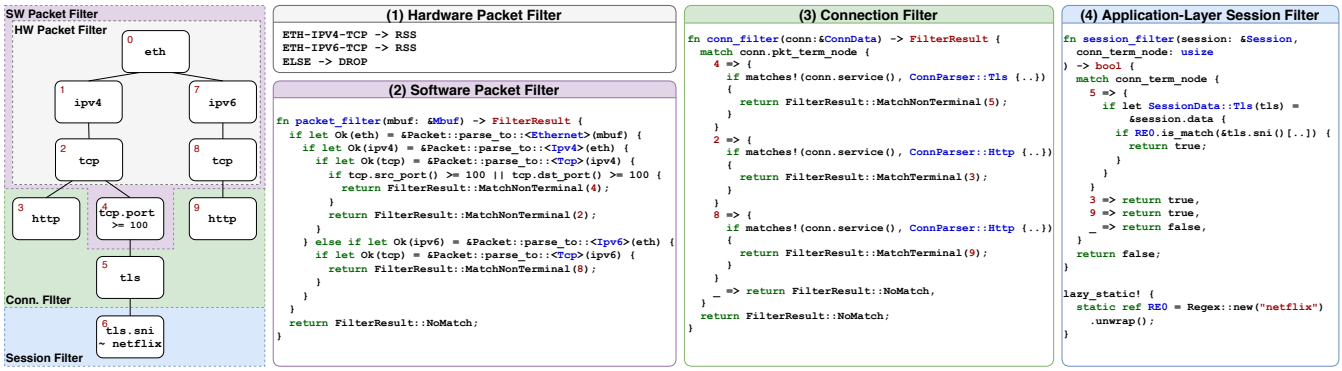


Figure 3: Filter Decomposition and Code Generation—We show how the filter `(ipv4 and tcp.port>=100 and tls.sni~'netflix') or http` is decomposed and executed. Retina parses the specified filter expression into a predicate trie representation, and generates (1) a set of NIC-compatible flow rules, (2) 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 3× higher 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) [93]. 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, 65]) 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 99^{th} 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.

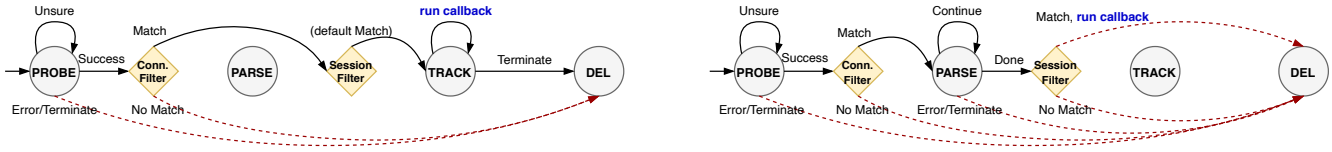
Light-Weight Stream Reassembly. Traditional approaches to TCP reassembly involve allocating data buffers to hold packet payloads as they arrive from the network [46, 67, 92]. While this design provides a convenient stream abstraction for applications that require access to in-order bytes, it is wasteful 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 `Probe` the connection for protocol messages, `Parse` the application-layer protocol, `Track` the connection without parsing, or `Delete` 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-Specific Connection State Diagrams—Retina automatically derives state transitions to efficiently parse connections according to the filter 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 filter (which can happen as early as the Client Hello) are immediately dropped, and likewise for those that do not match the session filter `tls.sni ~ '.*\\.com$'`. On a full filter 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 traffic while remaining flexible over different subscription types.

5.3 Callback Execution

Callbacks are closures that define 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 flexibility to write arbitrary analysis functions in a general purpose programming language. By building on Rust, Retina provides memory safety guarantees that prevent user-defined 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 unified processing environment. This significantly 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 efficient 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 filter or improve callback efficiency 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 traffic rate (i.e., incurs sustained non-zero packet loss), the user may consider using a buffered writer, increasing the number of cores, or even narrowing the filter. 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.

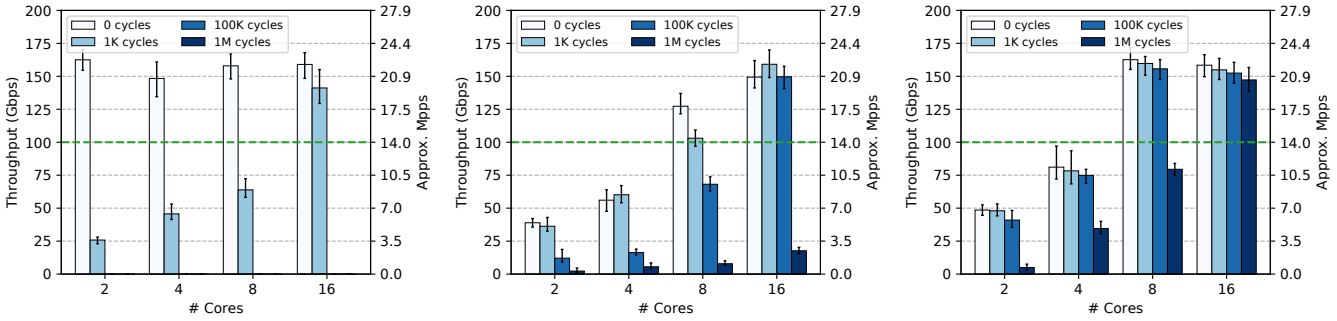
6 PERFORMANCE EVALUATION

In this section, we evaluate Retina's performance, and demonstrate its ability to process *over 100 Gbps of real-world traffic* 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 traffic 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 traffic from a large university network, with packets duplicated across the two links such that we receive double the regular traffic. 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 traffic 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 traffic generators like DPDK Pktgen [89] and TRex [78] are unable to synthetically generate a realistic distribution of flows with proper payloads at line rate either. While using real traffic exactly matches our intended use-case, it is inherently inconsistent and difficult 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 traffic in Appendix C.

Ethical Considerations. As part of our evaluation, we measured whether Retina was capable of processing high-speed traffic on our campus network. This analysis was approved by our university's privacy and security office. We did not investigate human behavior, surface or investigate any individual flows or IP addresses, or store any traffic 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 traffic 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 traffic 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 significant variation that can occur depending on the filter, the data representation, and the complexity of the callback. We first 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 traffic rate on the observed network, we adjust the rate of traffic 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 effective ingress rate at the CPU without sacrificing flow consistency. This technique can optionally be used to sample traffic for extremely heavy workloads, but none of our applications have required us to do so (Section 7). Unfortunately, flow sampling cannot be enabled with hardware flow rules, so we disable hardware filtering to give a lower bound on the maximum throughputs measured in this experiment.

For each trial, we start at the full ingress traffic rate and slowly increase the percentage of flows dropped by the NIC until we observe zero packet loss for five 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 traffic rate, or (2) the throughput saturates the ingress link at above 100 Gbps. The dynamic nature of the input traffic makes it hard to find a stable comparison point between configurations 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 traffic 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 traffic, and with 16 cores for more complex (100K+ cycles) processing. For context, logging connection records to a shared file takes around 12K cycles on our platform. Naturally, there are far fewer connections than individual packets, enabling higher maximum traffic 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 traffic rates exceeding 160 Gbps, even for heavy per-handshake workloads. Although subscriptions for TLS handshakes require application-layer parsing, Retina's filtering 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 fulfill 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],

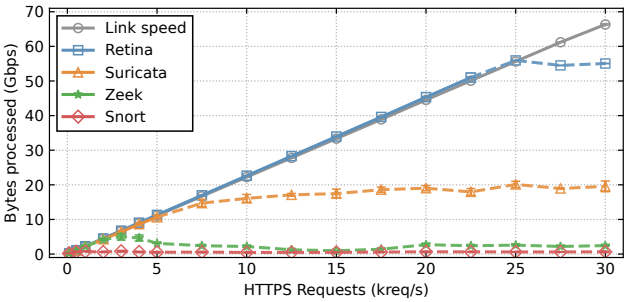


Figure 6: Comparison with Network Monitors—Retina is able to sustain 5–100× higher ingress traffic rates on a single core compared to Suricata+DPDK, Snort+DPDK, and Zeek+AF_PACKET. Dashed lines indicate packet losses above 1%.

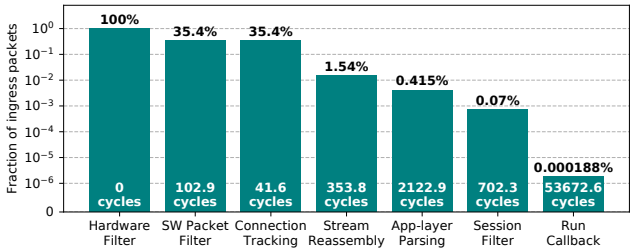


Figure 7: Effect of Filter Decomposition— Retina hierarchically reduces the fraction of ingress packets that trigger each processing stage, dramatically lowering the average end-to-end per-packet processing time. We show the break down of average CPU cycles and the fraction of packets consumed by each stage (note the log scale). Filter: `tcp.port = 443 and tls.sni ~ '(.+?\.)?nflxvideo\.net'`

and PacketMill [34] that lack the expressiveness and ease-of-use required for complex network analysis.

Setup. To enable measurement under controlled, though less realistic, workloads, we use two HTTPS client and server machines to generate traffic towards the system under test (instead of a production network). The client generates 128 parallel closed-loop 256 KB HTTPS requests using wrk2 [42] at different rates towards an Nginx server interconnected with 100 GbE Mellanox ConnectX-5 NICs. Each machine uses Linux Traffic Control to mirror the traffic received towards a 100 GbE port linked to the system under test.

Retina is configured to log connections that match the TLS server name as well as output its own performance logs. In an effort to provide a fair comparison, we disable all but the SSL protocol analyzer in Zeek and generate only SSL and diagnostics logs. Suricata is configured with a single rule to match on the SNI, and we disable all preprocessors except Stream5, TCP, and SSL on Snort. In addition, we extended Snort and Suricata to support DPDK in order to prevent performance bottlenecks that stem from packet I/O. We attempted to use a recent prototype Zeek DPDK plugin [32], but were unable to achieve higher performance compared to AF_PACKET. This observation, combined with recommendations from Zeek maintainers, led us to evaluate Zeek with AF_PACKET instead. Since Snort is not multi-threaded and the testbed is limited to 70 Gbps without packet losses, we restrict each system to a single core and disable hardware offloads so that all systems run fully in software.

Results. Figure 6 shows that Retina performs best for this analysis task, only dropping packets at above 49 Gbps using a single core and no hardware filtering. Suricata follows, with less than half of Retina's processing throughput, but drops packets at above 10 Gbps. Zeek only achieves around 5 Gbps (4 Gbps without packet loss). This is on par with advertised Zeek performance numbers [20] and estimates from prior work [76]. Snort achieves 1 Gbps at best, but only 400 Mbps of zero-loss throughput. Snort is particularly slow due to its inability to run the pattern matching algorithm on select packets only, despite configuring our single rule to match SSL connections in the Client Hello state. While the ease of writing simple Rust callbacks for different data abstractions is difficult to quantify, Retina is able to process 5 to 100 times higher traffic rates than popular network monitoring systems due to its compile-time, strict to the point-and-nothing-more pipeline. Zeek, Snort, and Suricata, on the other hand, are primarily designed for full visibility into all traffic, making them more capable of intrusion detection and security monitoring but less suitable for answering specific questions about high-speed network traffic.

6.3 Multi-Layer Filtering

A key aspect of Retina's performance is its filter decomposition and ability to quickly discard out-of-scope traffic. To directly show how this affects processing throughput, we show the effect for an example application that extracts transport-layer features of video traffic from 100 GbE networks (detailed in Section 7.3). We break down the time spent in each processing stage, and show how Retina hierarchically reduces the frequency that each stage needs to run.

Setup. We subscribe to TCP connection records on our campus network and filter for connections to a subset of Netflix video servers on port 443. We enable hardware filtering and record the number of times each major processing stage runs as well as the average number of CPU cycles spent per stage.

Results. Figure 7 depicts the fraction of packets that trigger each processing stage, along with the average number of CPU cycles consumed when each stage is run. We see that 35.4% of packets satisfy the packet filter and require a lookup or insert into the connection table. However, only 1.54% of packets ever need to be reassembled. This is because the connection filter eliminates non-TLS connections as early as the TLS Client Hello and the session filter removes non-Netflix connections as soon as TLS handshake parsing completes. Packets from discarded connections are not reassembled, while those in remaining connections no longer need to be parsed. The end result is that the user callback, which executes relatively expensive analysis code, runs on only 0.000188% of the incoming packets. By efficiently reducing the frequency that each processing stage runs, Retina dramatically lowers the average end-to-end per-packet processing time.

We note that not all subscriptions may be satisfiable across all traffic. For instance, if we were instead monitoring a private peering link to Netflix, it would be reasonable to assume that most connections would not be discarded by the filter, thus reducing the observed performance benefits. If the specified filter still results in too much traffic given the compute resources available, users can enable connection-aware sampling (as described in Section 6.1).

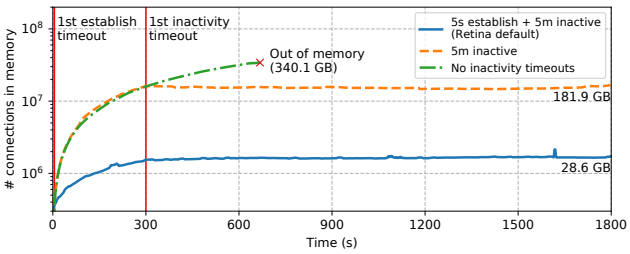


Figure 8: Memory Usage Over Time— Retina's timeout mechanism allows it to support real-world, high-volume networks without exhausting memory resources.

However, we expect that typical high-speed links contain a mixture of traffic types, much of which can be filtered out to improve performance for the desired analysis task.

6.4 State Management

Beyond conserving CPU cycles, Retina must also maintain connection state without running out of memory. Here, we show that Retina is able to handle long-term, stateful analysis of high-volume production networks.

Setup. We subscribe to all TCP connection records on our campus network for 30 minutes using different connection inactivity timeout schemes. We record the number of simultaneous connections tracked in memory as well as the overall memory usage every second. To minimize temporal variations in the observed traffic rate, we run each experiment back-to-back and use 16 cores to saturate the ingress link.

Results. In Figure 8, we show that by using Retina's default timeout mechanism to early expire non-established connections, we can fulfill the same subscription using 6.4 times less memory at steady-state and tracking 7.7 times fewer concurrent connections than using a fixed 5 minute inactivity timeout. With no timeouts, Retina runs for just over 11 minutes before running out of memory. This improvement comes from the observation that the majority of new connections on a large, modern network consist of a single unanswered SYN, which can be quickly delivered to the user to minimize the number of concurrent connections in memory.

7 VERSATILITY

In this section, we evaluate Retina's ability to easily answer complex questions about large-scale network traffic. This includes investigating cryptographic anomalies, IP anonymization, extracting traffic features for model inference, traffic profiling, flow export and analysis, and more. We show that Retina provides a flexible way for operators and researchers to run investigations like these in just a few lines of code, drastically lowering the bar to perform sophisticated studies in a timely manner. In this section, we explore three of these applications in detail as case studies.

7.1 Cryptographic Anomalies

Cryptographic protocols like TLS and SSH are notoriously difficult to implement and deploy correctly. Many vulnerabilities, such as the Logjam [1] and DROWN [4] attacks, have been found through large-scale empirical measurement studies. However, most empirical research has focused on server-side weaknesses since they can be surveyed through Internet-wide scanning [28, 31, 75]. Studies about client behaviors have been relegated to manually investigating a small number of popular applications [12] or relying on heavily sampled traffic [30, 51]. Unfortunately, these techniques miss the long-tail of client configurations in less popular applications, which are more likely to contain vulnerabilities.

Retina enables large-scale empirical investigations through its ability to extract parsed protocol data from high speed network traffic without sampling. To illustrate this, we investigate the distribution of random nonces chosen by TLS client implementations. A fundamental assumption behind cryptographic nonces is that the likelihood of observing repeat values should be exceptionally low; however, this assumption is difficult to verify empirically without the ability to analyze large quantities of handshakes. Using just 40 lines of Rust code, we developed an application to measure the frequency of distinct client randoms observed on our network. We ran the experiment with 16 cores for 10 minutes with an average ingress rate of 157.4 Gbps with zero loss.

During this time, we observed 13.4M TLS handshakes. Interestingly, the value `738b712a...dee0dbe1` appears the most frequently, with 8,340 occurrences in just 10 minutes of monitoring. `417a7572...00000000` and `0` are the second and third most common nonces with 493 and 309 instances, respectively, likely indicating issues with device entropy or incorrect TLS implementations.

7.2 Anonymized Packet Analysis

Raw network packets are leveraged for many traffic analysis tasks, including device and application identification [11, 81], malicious traffic classification [53] and intrusion detection [88]. While such tasks require a substantial amount of data for training and testing, a lack of shareable packet traces remains as one of the most apparent obstacles to achieve realistic progress. Many researchers use laboratory setups to mimic real-world environments [81], or decades-old public traces [88]. Both types of datasets do not represent up-to-date, realistic network traffic and thus can bias evaluation results. Researchers have also spent considerable effort manually filtering raw traces to extract packets relevant to their analysis [11].

To accommodate privacy concerns, researchers and operators can use Retina to anonymize packets' IP addresses and develop arbitrary analysis functions on filtered packets in a realistic and timely manner. We implemented a sample application where we encrypt the source and destination IP addresses of all HTTP packets while preserving subnet structures. Due to Retina's seamless third-party library integration, we were able to write the application in just 11 lines of code by calling functions from a public Rust crate [3] that implements format-preserving IP encryption. We ran the application with 16 cores for 10 minutes at an average processing throughput of 164.7 Gbps with zero packet loss.

7.3 Feature Extraction for Model Inference

The ability to perform large-scale passive analysis provides the means to easily extract diverse traffic features for a variety of operational use cases. One such use case is inferring streaming video

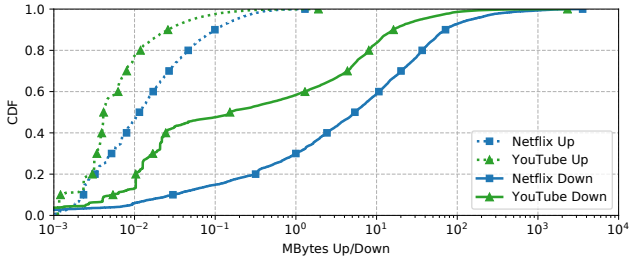


Figure 9: CDF of byte counts up/down for video sessions from Netflix and YouTube—Retina easily enables traffic feature extraction while filtering on application-layer data in real time.

quality, an increasingly important application for Internet Service Providers (ISPs) seeking to accurately measure customer experience. Researchers have attempted to develop methods to infer the quality of specific video services, but face challenges isolating video traffic and collecting the features needed to perform model inference. Previous work has required embedding traffic collection systems in volunteers' homes, which presents deployment and data storage challenges [17], or emulating network conditions in a laboratory testbed, which has disadvantages compared to analyzing real-world traffic [60]. Here, we demonstrate Retina's ability to isolate traffic and extract network features for model inference.

We developed a Retina application that subscribes to TCP connections on the network, filtered by the targeted video streaming service. To isolate HTTPS traffic sent from Netflix and YouTube servers, we filter on TCP port 443 and the server name (SNI) field for domains that carry video traffic: `tls.sni ~ '(.+?\.)?nflxvideo\.net'` for Netflix and `tls.sni ~ 'googlevideo'` for YouTube. In around 100 lines of Rust code, we implemented a subscription callback that aggregates network flows within video sessions to extract and log several features used by Bronzino et al. [17] to infer video quality metrics, including the number of parallel flows, total bytes up/down, average number of out-of-order packets up/down, and total download throughput. As a proof-of-concept, we collected data for both Netflix and YouTube traffic using 16 cores for one hour, averaging 152.8 Gbps of aggregate input across both runs with zero loss. We show the distribution of total bytes upstream and downstream for video sessions in Figure 9.

8 RELATED WORK

There is a significant body of prior work focused on processing and analyzing high-speed network traffic. In this section, we discuss related work and work that inspired aspects of Retina's design.

Network Analysis Tools. Several passive analysis frameworks have been developed in both academia and industry. Most similar is Zeek [68], an open-source network monitor commonly used in research studies and for intrusion detection [22, 30, 82, 83]. While Zeek can answer similar questions as Retina, it is natively single-threaded, requiring large cluster architectures to accommodate modern network speeds [76]. Snort [73] is another popular IDS that provides similar support for stream reassembly and application-layer parsing, but lacks the ability to run generic analyses and is estimated to require hundreds of cores to reach 100 Gbps, even with perfect scaling [100]. Retina supports 100 GbE links on a single server and outperforms I/O optimized versions of these related systems for a given analysis task (Section 6.2). We note that Retina is uniquely designed to not have to inspect every packet, flow, or connection (which is necessary for intrusion detection), and instead optimizes the processing pipeline for specific analysis questions by discarding extraneous traffic as early and as often as possible.

Several commercial solutions marketed for 100 Gbps visibility exist (e.g., [2, 20, 62, 98]). While some commercial products offer capabilities like intrusion prevention, graphical interfaces, and packet storage and indexing that go beyond Retina's target use cases, few support the same programmability that enables users to write arbitrary analysis code on data types ranging from raw packets to application-layer handshakes. Moreover, commercial products often require specialized hardware such as proprietary sensor nodes, whereas Retina supports 100 Gbps traffic analysis on commodity hardware.

Prior systems have exposed callback interfaces similar to Retina's subscription programming model [55, 96]. mOS [55] provides fine-grained control over TCP flow events for middlebox programming, but does not provide application-layer abstractions. Retina differs by hiding the complexities of application-layer parsing and filtering, enabling users to easily analyze high-level network data without needing to handle low-level protocol events. MiddleClick [9] also focuses on middlebox programming. While it is not suited for passive analysis, it has a more limited but similar ability to express offset/mask-based packet filters that will be partially offloaded. However, it requires a complex network-oriented graph-based configuration, and lacks the ability to process those filters at compile-time. dShark [96] uses a domain-specific language for analyzing different types of packet aggregations. However, dShark focuses on distributed trace analysis for diagnosing network failures and does not support analyzing connections or application-layer messages. Packet-o-matic [59] is another event-based network sniffer that provides access to packets, connections, and application-layer sessions, but does not scale to meet our 100 Gbps performance objectives and does not support any filtering. Perna et al. [69] recently released an identically named command-line tool that extracts statistics from RTP network traffic. However, it performs only offline analysis of RTP-based traffic and is not compatible with high-speed links.

Many previous works have improved real-time packet processing performance using kernel bypass [6, 8, 34, 94, 99] and hardware offloads [7, 46, 70]. These systems are packet-oriented and lack the expressiveness required for complex analyses. Some frameworks aim for a middle-ground between performance and expressiveness by using a domain-specific query language [13, 45, 63, 97], but are still limited in their ability to express arbitrary analysis tasks. For example, Sonata does not support queries that require reassembling byte-streams [45]. Retina allows users to write arbitrary analysis code on data at multiple layers of the networking stack, while transparently discarding out-of-scope traffic to accommodate 100+ GbE network environments.

Filtering. Berkeley Packet Filter (BPF) [61] is the de facto filtering mechanism used by most network monitoring tools [36, 44, 68, 73, 91]. However, BPF does not support application-layer predicates and cannot be extended to support arbitrary protocols or complex

semantics like regular expression matching. eBPF programs can be used to perform more advanced filtering, but require tedious and error-prone manual implementation. Retina supports user-friendly semantics for both packet headers and application-layer data, and it statically verifies all filters using the Rust compiler.

The Camus subscription language [56] is similar in syntax to Retina filters, but focuses on compiling filter expressions to programmable switches for packet forwarding rather than traffic analysis. Camus supports arbitrary packet formats, but does not support filtering on application-layer fields due to hardware constraints. These filtering engines served as inspiration for Retina's filter language, and we build upon them to encompass more complex semantics and real-time application-layer filters.

9 CONCLUSION

In this paper, we introduced Retina, an expressive network analysis framework capable of asking questions of high-speed traffic. Retina introduces a subscription programming model with a user-friendly filter language to help researchers and operators understand large-scale networks without spending significant effort optimizing low-level data collection mechanics. We explored several applications written using Retina and showed that it can easily answer questions on 100+ Gbps links for a diverse set of use cases, closing the gap between performance and expressiveness that has eluded prior solutions. Looking forward, we believe that further optimizations to filtering, such as including a P4-capable device in the filtering layers or using the results of inference algorithms to filter on metadata, are possible. We also leave support for different callback execution models to future work. Retina is readily deployable on commodity hardware, and is openly available to the community.

ACKNOWLEDGEMENTS

We thank our shepherd Roland van Rijswijk-Deij as well as Tina Wu, Michalis Kallitsis, Zane Ma, Nick Feamster, Francesco Bronzino, Renata Teixeira, and the anonymous reviewers for their helpful comments and feedback. We thank Christian Kreibich for his help and guidance for benchmarking Zeek. We also thank Deepti Raghavan, Alex Ozdemir, Stephen Ibanez, Theo Jepsen, Kostis Kaffes, Nick McKeown, Keith Winstein, and members of the Stanford Empirical Security Research Group for valuable discussions. We thank the Stanford University security and networking teams, including Andrej Krevl, Will Johnson, and Michael Duff, for facilitating access to campus traffic. This work was supported in part by the National Science Foundation under award CNS-2124424, and through gifts from Google, Inc., Cisco Systems, Inc., and Comcast Corporation. Tom Barbette was funded by an FSR fellowship from UCLouvain.

REFERENCES

[1] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J. Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, Benjamin VanderSloot, Eric Wustrow, Santiago Zanella-Beguelin, and Paul Zimmermann. 2015. Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice. In *ACM Conference on Computer and Communications Security*.

[2] APCON. 2022. Network Monitoring. https://apcon.com/solutions/network-monitoring.

[3] Jean-Philippe Aumasson. 2018. Rust ipcrypt. https://github.com/stbuehler/rust-ipcrypt.

[4] Nimrod Aviram, Sebastian Schinzel, Juraj Somorovsky, Nadia Heninger, Maik Dankel, Jens Steube, Luke Valenta, David Adrian, J. Alex Halderman, Viktor Dukhovni, Emilia Käsper, Shaanan Cohney, Susanne Engels, Christof Paar, and Yuval Shavitt. 2016. DROWN: Breaking TLS using SSLv2. In *USENIX Security Symposium*.

[5] David F. Bacon, Rodric Rabbah, and Sunil Shukla. 2013. FPGA Programming for the Masses. In *Communications of the ACM*.

[6] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. 2018. High-Speed Software Data Plane via Vectorized Packet Processing. *IEEE Communications Magazine*.

[7] Tom Barbette, Georgios P. Katsikas, Gerald Q. Maguire Jr., and Dejan Kostić. 2019. RSS++: load and state-aware receive side scaling. In *Conference on emerging Networking EXperiments and Technologies*.

[8] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2015. Fast Userspace Packet Processing. In *ACM/IEEE Symposium on Architectures for Networking and Communications Systems*.

[9] Tom Barbette, Cyril Soldani, and Laurent Mathy. 2021. Combined Stateful Classification and Session Splicing for High-Speed NFV Service Chaining. *IEEE/ACM Transactions on Networking*.

[10] Simon Bauer, Benedikt Jaeger, Fabian Helfert, Philippe Barias, and Georg Carle. 2021. On the Evolution of Internet Flow Characteristics. In *Applied Networking Research Workshop*.

[11] Laurent Bernaille, Renata Teixeira, and Kavé Salamatian. 2006. Early Application Identification. In *Conference on emerging Networking EXperiments and Technologies*.

[12] Benjamin Beurdouche, Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Markulf Kohlweiss, Alfredo Pironti, Pierre-Yves Strub, and Jean Karim Zinzindohoue. 2015. A Messy State of the Union: Taming the Composite State Machines of TLS. In *IEEE Symposium on Security and Privacy*.

[13] Kevin Borders, Jonathan Springer, and Matthew Burnside. 2012. Chimera: A Declarative Language for Streaming Network Traffic Analysis. In *USENIX Security Symposium*.

[14] Robert T. Braden. 1988. A Pseudo-Machine for Packet Monitoring and Statistics. In *ACM Special Interest Group on Data Communications*.

[15] Mary Branscombe. 2018. The Year of 100GbE in Data Center Networks. https://www.datacenterknowledge.com/networks/year-100gbe-data-center-networks.

[16] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Hyojoon Kim, Renata Teixeira, and Nick Feamster. 2022. Traffic Refinery: Cost-Aware Data Representation for Machine Learning on Network Traffic. In *ACM Special Interest Group on Measurement and Evaluation*.

[17] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. 2019. Inferring Streaming Video Quality from Encrypted Traffic: Practical Models and Deployment Experience. In *Proceedings of the ACM on Measurement and Analysis of Computing Systems*.

[18] CAIDA. 2019. Passive Monitor: equinix-chicago. https://www.caida.org/catalog/datasets/monitors/passive-equinix-chicago.

[19] Marco Caselli, Emmanuele Zambon, Johanna Amann, Robin Sommer, and Frank Kargl. 2016. Specification Mining for Intrusion Detection in Networked Control Systems. In *USENIX Security Symposium*.

[20] Corelight. 2022. Compare to open source Zeek. https://corelight.com/products/compare-to-open-source-zeek.

[21] crates.io. 2022. The Rust community's crate registry. https://crates.io.

[22] Louis F. DeKoven, Audrey Randall, Ariana Mirian, Gautam Akiwate, Ansel Blume, Lawrence K. Saul, Aaron Schulman, Geoffrey M. Voelker, and Stefan Savage. 2019. Measuring Security Practices and How They Impact Security. In *ACM Internet Measurement Conference*.

[23] Pedro M. Santiago del Rio, Dario Rossi, Francesco Gringoli, Lorenzo Nava, Luca Salgarelli, and Javier Aracil. 2012. Wire-Speed Statistical Classification of Network Traffic on Commodity Hardware. In *ACM Internet Measurement Conference*.

[24] CVE Details. 2019. Bro: Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-16374/BRO.html.

[25] CVE Details. 2021. Suricata: Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-17948/Suricata-ids.html.

[26] Sarang Dharmapurikar and Vern Paxson. 2005. Robust TCP Stream Reassembly In the Presence of Adversaries. In *USENIX Security Symposium*.

[27] Holger Dreger, Anja Feldmann, Vern Paxson, and Robin Sommer. 2004. Operational Experiences with High-Volume Network Intrusion Detection. In *ACM Conference on Computer and Communications Security*.

[28] Zakir Durumeric, James Kasten, Michael Bailey, and J. Alex Halderman. 2013. Analysis of the HTTPS Certificate Ecosystem. In *ACM Internet Measurement Conference*.

[29] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. 2014. The Matter of Heartbleed. In *ACM Internet Measurement Conference*.

[30] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J. Alex Halderman, and Vern Paxson. 2017. The

Security Impact of HTTPS Interception. In *Network and Distributed System Security Symposium*.
[31] Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. 2013. ZMap: Fast Internet-Wide Scanning and its Security Applications. In *USENIX Security Symposium*.
[32] ESNET. 2022. ESnet DPDK Plugin. https://github.com/esnet/dpdk-plugin.
[33] Dhia Farrah and Marc Dacier. 2021. Zero Conf Protocols and their numerous Man in the Middle (MITM) Attacks. In *IEEE Security and Privacy Workshops*.
[34] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. 2021. PacketMill: Toward Per-Core 100-Gbps Networking. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*.
[35] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chand appa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *USENIX Symposium on Networked Systems Design and Implementation*.
[36] Open Information Security Foundation. 2022. Suricata. https://suricata.io.
[37] The Linux Foundation. 2022. Data Plane Development Kit. https://www.dpdk.org.
[38] Sergey Frolov, Jack Wampler, Sze Chuen Tan, J Alex Halderman, Nikita Borisov, and Eric Wustrow. 2019. Conjure: Summoning Proxies from Unused Address Space. In *ACM SIGSAC Conference on Computer and Communications Security*.
[39] Sergey Frolov and Eric Wustrow. 2019. The use of TLS in Censorship Circumvention. In *Network and Distributed System Security Symposium*.
[40] Sebastián García, Karel Hynek, Dmtrii Vekshin, Tomáš Čejka, and Armin Wasicek. 2021. Large Scale Measurement on the Adoption of Encrypted DNS. In *arXiv preprint arXiv:2107.04436*.
[41] Massimo Girondi, Marco Chiesa, and Tom Barbette. 2018. High-speed Connection Tracking in Modern Servers. In *IEEE Conference on High-Performance Switching and Routing*.
[42] Will Glozer and Gil Tene. 2020. wrk2. https://github.com/giltene/wrk2.
[43] Qian Gong, Wenji Wu, and Phil DeMar. 2018. GoldenEye: Stream-based Network Packet Inspection using GPUs. In *IEEE Conference on Local Computer Networks*.
[44] The Tcpdump Group. 2022. Tcpdump and libpcap. https://www.tcpdump.org.
[45] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-Driven Streaming Network Telemetry. In *ACM Special Interest Group on Data Communications*.
[46] Sangjin Han, Keon Jang, Aurojit Panda, Shoumik Palkar, Dongsu Han, and Sylvia Ratnasamy. 2015. *SoftNIC: A Software NIC to Augment Hardware*. Technical Report. EECS Department, University of California, Berkeley.
[47] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: a GPU-Accelerated Software Router. In *ACM Special Interest Group on Data Communications*.
[48] Rick Hofstede, Idilio Drago, Anna Sperotto, Ramin Sadre, and Aiko Pras. 2013. Measurement Artifacts in NetFlow Data. In *International Conference on Passive and Active Measurement*.
[49] Jordan Holland, Paul Schmitt, Nick Feamster, and Prateek Mittal. 2021. New Directions in Automated Traffic Analysis. In *ACM Conference on Computer and Communications Security*.
[50] Ralph Holz, Johanna Amann, Olivier Mehani, Matthias Wachs, and Mohamed Ali Kaafar. 2015. TLS in the wild: An Internet-wide analysis of TLS-based protocols for electronic communication. In *Network and Distributed System Security Symposium*.
[51] Ralph Holz, Lothar Braun, Nils Kammenhuber, and Georg Carle. 2011. The SSL Landscape – A Thorough Analysis of the X.509 PKI Using Active and Passive Measurements. In *ACM Internet Measurement Conference*.
[52] Ralph Holz, Jens Hiller, Johanna Amann, Abbas Razaghpanah, Thomas Jost, Narseo Vallina-Rodriguez, and Oliver Hohlfeld. 2020. Tracking the deployment of TLS 1.3 on the Web: A story of experimentation and centralization. In *ACM SIGCOMM Computer Communication Review*.
[53] Ren-Hung Hwang, Min-Chun Peng, Van-Linh Nguyen, and Yu-Lun Chang. 2019. An LSTM-Based Deep Learning Approach for Classifying Malicious Traffic at the Packet Level. In *Applied Sciences*.
[54] Marios Iliofotou, Prashanth Pappu, Michalis Faloutsos, Michael Mitzenmacher, Sumeet Singh, and George Varghese. 2007. Network Monitoring using Traffic Dispersion Graphs (TDGs). In *ACM Internet Measurement Conference*.
[55] Muhammad Jamshed, YoungGyoun Moon, Donghwi Kim, Dongsu Han, and KyoungSoo Park. 2017. mOS: A Reusable Networking Stack for Flow Monitoring Middleboxes. In *USENIX Symposium on Networked Systems Design and Implementation*.
[56] Theo Jepsen, Ali Fattaholmanan, Masoud Moshref, Nate Foster, Antonio Carzaniga, and Robert Soulé. 2020. Forwarding and Routing with Packet Subscriptions. In *Conference on emerging Networking EXperiments and Technologies*.
[57] Georgios P. Katsikas, Tom Barbette, Marco Chiesa, Dejan Kostić, and Gerald Q. Maguire Jr. 2021. What You Need to Know About (Smart) Network Interface Cards. In *International Conference on Passive and Active Measurement*.
[58] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. In *ACM Asia-Pacific Workshop on Systems*.
[59] Guy Martin. 2017. Packet-o-matic. http://www.packet-o-matic.org.
[60] M. Hammad Mazhar and Zubair Shafiq. 2018. Real-time Video Quality of Experience Monitoring for HTTPS and QUIC. In *INFOCOM*.
[61] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Winter USENIX Conference*.
[62] Napatech. 2022. Napatech NT100A01 SmartNIC. https://www.napatech.com/products/nt100a01-smartnic-capture.
[63] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *ACM Special Interest Group on Data Communications*.
[64] ntop. 2022. PF_RING. https://www.ntop.org/products/packet-capture/pf_ring.
[65] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving High CPU Efficiency for Latency-sensitive Datacenter Workloads. In *USENIX Symposium on Networked Systems Design and Implementation*.
[66] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *USENIX Symposium on Operating Systems Design and Implementation*.
[67] Antonis Papadogiannakis, Michalis Polychronakis, and Evangelos P. Markatos. 2013. Scap: Stream-Oriented Network Traffic Capture and Analysis for High-Speed Networks. In *ACM Internet Measurement Conference*.
[68] Vern Paxson. 1999. Bro: A System for Detecting Network Intruders in Real-Time. In *Computer Networks*.
[69] Gianluca Perna, Dena Markudova, Martino Trevisan, Paolo Garza, Michela Meo, and Maurizio M. Munafò. 2022. Retina: An open-source tool for flexible analysis of RTC traffic. In *Computer Networks*.
[70] Salvatore Pontarelli, Roberto Bifulco Marco Bonola, Carmelo Cascone, Marco Spaziani, Valerio Bruschi, Davide Sanvito, Giuseppe Siracusano, Antonio Capone, Michio Honda, Felipe Huici, and Giuseppe Bianchi. 2019. FlowBlaze: Stateful Packet Processing in Hardware. In *USENIX Symposium on Networked Systems Design and Implementation*.
[71] Alexander Yu. Privalov and Alexander Tsarev. 2014. Analysis and Simulation of WAN Traffic by Self-similar Traffic Model with OMNET. In *International Wireless Communications and Mobile Computing Conference*.
[72] Deepti Raghavan, Philip Levis, Matei Zaharia, and Irene Zhang. 2021. Breakfast of Champions: Towards Zero-Copy Serialization with NIC Scatter-Gather. In *Workshop on Hot Topics in Operating Systems*.
[73] Martin Roesch. 1999. Snort - Lightweight Intrusion Detection for Networks. In *USENIX Systems Administration Conference*.
[74] David Sidler, Gustavo Alonso, Michaela Blott, Kimon Karras, Kees Vissers, and Raymond Carley. 2015. Scalable 10 Gbps TCP/IP Stack Architecture for Reconfigurable Hardware. In *International Symposium on Field-Programmable Custom Computing Machines*.
[75] Drew Springall, Zakir Durumeric, and J. Alex Halderman. 2016. Measuring the Security Harm of TLS Crypto Shortcuts. In *ACM Internet Measurement Conference*.
[76] Vincent Stoffer, Aashish Sharma, and Jay Krous. 2015. *100G Intrusion Detection*. Technical Report. Lawrence Berkeley National Laboratory.
[77] Stratosphere. 2015. Stratosphere Laboratory Datasets. https://www.stratosphereips.org/datasets-overview.
[78] TRex Team. 2022. TRex Traffic Generator. https://trex-tgn.cisco.com.
[79] Accolade Technology. 2021. Accolade ANIC-100Kq. https://accoladetechnology.com/portfolio-item/anic-100kq.
[80] Martino Trevisan, Alessandro Finamore, Marco Mellia, Maurizio Munafò, and Dario Rossi. 2017. Traffic Analysis with Off-the-Shelf Hardware: Challenges and Lessons Learned. In *IEEE Communications Magazine*.
[81] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. 2020. Packet-Level Signatures for Smart Home Devices. In *Network and Distributed System Security Symposium*.
[82] Robert Udd, Mikael Asplund, Simin Nadjm-Tehrani, Mehrdad Kazemtabrizi, and Mathias Ekstedt. 2016. Exploiting Bro for Intrusion Detection in a SCADA System. In *ACM International Workshop on Cyber-Physical System Security*.
[83] Alisha Ukani, Ariana Mirian, and Alex C. Snoeren. 2021. Locked-In during Lock-Down: Undergraduate Life on the Internet in a Pandemic. In *ACM Internet Measurement Conference*.
[84] Benjamin VanderSloot, Johanna Amann, Matthew Bernhard, Zakir Durumeric, Michael Bailey, and J Alex Halderman. 2016. Towards a Complete View of the Certificate Ecosystem. In *ACM Internet Measurement Conference*.

[85] Benjamin VanderSloot, Sergey Frolov, Jack Wampler, Sze Chuen Tan, Irv Simpson, Michalis Kallitsis, J Alex Halderman, Nikita Borisov, and Eric Wustrow. 2020. Running Refraction Networking for Real. (2020).
[86] George Varghese and Anthony Lauck. 1997. Hashed and Hierarchical Timing Wheels: Efficient Data Structures for Implementing a Timer Facility. In *IEEE/ACM Transactions on Networking*.
[87] Giorgos Vasiliadis, Lazaros Koromilas, Michalis Polychronakis, and Sotiris Ioannidis. 2014. GASPP: A GPU-Accelerated Stateful Packet Processing Framework. In *USENIX Annual Technical Conference*.
[88] Wei Wang, Yiqiang Sheng, Jinlin Wang, Xuewen Zeng, Xiaozhou Ye, Yongzhong Huang, and Ming Zhu. 2018. HAST-IDS: Learning Hierarchical Spatial-Temporal Features Using Deep Neural Networks to Improve Intrusion Detection. In *IEEE Access*.
[89] Keith Wiles. 2019. DPDK-pktgen. https://github.com/Pktgen/Pktgen-DPDK.
[90] Wireshark. 2021. DisplayFilters. https://gitlab.com/wireshark/wireshark/-/wikis/DisplayFilters.
[91] Wireshark. 2022. About Wireshark. https://www.wireshark.org.
[92] Rafal Wojtczuk. 2010. Libnids. http://libnids.sourceforge.net.
[93] Shinae Woo and KyoungSoo Park. 2012. *Scalable TCP Session Monitoring with Symmetric Receive-side Scaling*. Technical Report. Korea Advanced Institute of Science and Technology.
[94] Xiaoban Wu, Yan Luo, Jeronimo Bezerra, and Liang-Min Wang. 2019. Ares: A Scalable High-Performance Passive Measurement Tool Using a Multicore System. In *IEEE International Conference on Networking, Architecture and Storage*.
[95] Wai xi Liu, Yi er Yan, Tang Dong, and Run hua Tang. 2012. Self-Similarity and Heavy-Tail of ICMP Traffic. In *Journal of Computers*.
[96] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. 2019. dShark: A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces. In *USENIX Symposium on Networked Systems Design and Implementation*.
[97] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative Network Monitoring with NetQRE. In *ACM Special Interest Group on Data Communications*.
[98] Nadeem Zahid. 2020. 100Gbps-Enabled High-Speed Network Visibility. https://www.cpacket.com/100g-enabled-high-speed-network-visibility.
[99] Tianzhu Zhang, Leonardo Linguaglossa, Massimo Gallo, Paolo Giaccone, and Dario Rossi. 2018. FlowMon-DPDK: Parsimonious Per-flow Software Monitoring at Line Rate. In *Network Traffic Measurement and Analysis Conference*.
[100] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *USENIX Symposium on Operating Systems Design and Implementation*.
[101] Liang Zhu, Johanna Amann, and John Heidemann. 2016. Measuring the Latency and Pervasiveness of TLS Certificate Revocation. In *International Conference on Passive and Active Network Measurement*.

Appendices are supporting material that has not been peer-reviewed.

A MODULE EXTENSIBILITY

Retina's filter language and subscription data model are extensible. Developers can add new protocols to filter on by adding a protocol module to the framework, as well as new subscribable data types by implementing a subscription module. This also encourages code reuse, as well-defined protocol parsers and subscription types can be integrated into the framework core and made publicly available for the community.

A.1 Protocol Modules

Protocol modules define how individual packets and reordered streams should be parsed. All protocols must implement either the `PacketParsable` trait or the `ConnParsable` trait, which define the minimum set of methods required for Retina to parse input traffic. `PacketParsable` protocols are those that apply to individual packets (e.g., IPv4, TCP) and must define methods for how to parse itself from an encapsulating packet header, as well as provide the header length and offset to the next header. `ConnParsable` protocols are those that can be parsed from a connection byte-stream (e.g., TLS, HTTP), and define methods to identify protocol messages and manage application-layer state. Figure 10 shows the trait prototypes.

```rust
trait PacketParsable<'a> {
    /// Reference to the underlying Mbuf.
    fn mbuf(&self) -> &Mbuf;
    /// Offset from beginning of header to start of payload.
    fn header_len(&self) -> usize;
    /// Next level IANA protocol number.
    fn next_header(&self) -> Option<usize>;
    /// Offset from beginning of mbuf to start of payload.
    fn next_header_offset(&self) -> usize;
    /// Parse Self from the encapsulating packet's payload.
    fn parse_from(
        outer: &'a impl PacketParsable<'a>
    ) -> Result<Self>
    where Self: Sized;
}

trait ConnParsable {
    /// Probe the L4 protocol data unit for the protocol.
    fn probe(&self, pdu: &L4Pdu) -> ProbeResult;
    /// Parse the L4 protocol data unit as the protocol.
    fn parse(&mut self, pdu: &L4Pdu) -> ParseResult;
    /// Removes session with ID `session_id`.
    fn remove_session(
        &mut self, session_id: usize
    ) -> Option<Session>;
    /// Removes all pending sessions in the connection.
    fn drain_sessions(&mut self) -> Vec<Session>;
    /// Default state of the connection on a matched filter.
    fn session_match_state(&self) -> ConnState;
    /// Default state of the connection on a non-matched filter.
    fn session_nomatch_state(&self) -> ConnState;
}
```

Figure 10: Protocol Module Trait Methods—Packet-level protocols (e.g., IPv4, IPv6, TCP, UDP) must implement the `PacketParsable` trait, while connection-level protocols (e.g., TLS, HTTP, SSH) implement the `ConnParsable` trait. These traits define how Retina parses incoming packets and byte-streams, allowing easy protocol extensibility.

A.2 Subscription Modules

Subscription modules define how data is reconstructed before invoking the callback. Each subscribable type must implement the `Subscribable` trait, which defines the abstraction-level (packet, connection, or session), any application-layer parsers required for processing (used to populate the Protocol Registry, Figure 2), and the initial packet processing behavior. The `Subscribable` trait has an associated `Trackable` type, which defines the processing behavior for a tracked connection before and after a full filter match (e.g., buffer packets, immediately execute the callback, etc.). A `Trackable` type is associated with every active connection, allowing Retina to lazily reconstruct network data as the connection transitions through each connection state. We show the `Subscribable` and `Trackable` trait prototypes in Figure 11.

B FILTER CODE COMPILATION

Retina statically generates inlined code for each filter function to enable fast runtime filter execution in each stage of the processing pipeline. In this microbenchmark, we compare the performance benefits of natively compiled filter code to runtime interpreted filters. For an accurate comparison, we run Retina in *offline* mode, which ingests a pcap instead of packets from the network interface. We use publicly available Stratosphere [77] traces and measure the CPU time to process the trace on a single core with no hardware filtering. We log TLS handshakes with filters of differing complexities on

```rust
trait Subscribable {
    type Tracked: Trackable<Subscribed = Self>;

    /// Subscription level (Packet, Connection, or Session).
    fn level() -> Level;
    /// List of associated application-layer protocol parsers.
    fn parsers() -> Vec<ConnParser>;
    /// Process a single incoming packet.
    fn process_packet(
        mbuf: Mbuf,
        subscription: &Subscription<Self>,
        conn_tracker: &mut ConnTracker<Self::Tracked>,
    ) where Self: Sized;
}

trait Trackable {
    type Subscribed: Subscribable<Tracked = Self>;

    /// Instantiate a trackable entity for the connection.
    fn new(five_tuple: FiveTuple) -> Self;
    /// Update subscription data prior to a full filter match.
    fn pre_match(
        &mut self,
        pdu: L4Pdu,
        session_id: Option<usize>
    );
    /// Update subscription data on a full filter match.
    fn on_match(
        &mut self,
        session: Session,
        subscription: &Subscription<Self::Subscribed>
    );
    /// Update subscription data after a full filter match.
    fn post_match(
        &mut self,
        pdu: L4Pdu,
        subscription: &Subscription<Self::Subscribed>
    );
    /// Update subscription data on connection termination.
    fn on_terminate(
        &mut self,
        subscription: &Subscription<Self::Subscribed>
    );
}
```

Figure 11: Subscription Module Trait Methods—Subscription data types must implement the `Subscribable` and `Trackable` traits, which determine Retina's processing behavior for that type.

the following Stratosphere traces: CTU-Normal-7, CTU-Normal-12, CTU-Normal-20, and CTU-Normal-30.

In Figure 12, we show the runtime improvement from using native code generation over dynamically interpreted filters. Statically generated filter code produces a 5.4%–300.4% speedup, depending on traffic and filter complexity. Unsurprisingly, compiled filters are consistently faster than interpreted filters, but the benefits are less pronounced for basic filters (e.g., `ipv4`). However, for more complex filters, such as one[3] used by Bronzino et al. [16] and adapted to Retina's filter language, natively generated code can result in more than 3× higher processing throughputs. We note that filter code generation incurs a negligible increase in compilation time, but would necessitate recompilation for different filter expressions. On our platform, an incremental build with link time optimization takes 73 seconds on average. However, we argue that the runtime performance and memory safety benefits provided by the Rust compiler outweighs this cost.

[3]ipv4.addr in 23.246.0.0/18 or ipv4.addr in 37.77.184.0/21 or ipv4.addr in 45.57.0.0/17 or ipv4.addr in 64.120.128.0/17 or ipv4.addr in 66.197.128.0/17 or ipv4.addr in 108.175.32.0/20 or ipv4.addr in 185.2.220.0/22 or ipv4.addr in 185.9.188.0/22 or ipv4.addr in 192.173.64.0/18 or ipv4.addr in 198.38.96.0/19 or ipv4.addr in 198.45.48.0/20 or ipv4.addr in 208.75.79.0/24 or ipv6.addr in 2620:10c:7000::/44 or ipv6.addr in 2a00:86c0::/32 or tls.sni ~ 'netflix.com'or tls.sni ~ 'nflxvideo.net'or tls.sni ~ 'nflximg.net'or tls.sni ~ 'nflxext.com'or tls.sni ~ 'nflximg.com'or tls.sni ~ 'nflxso.net'

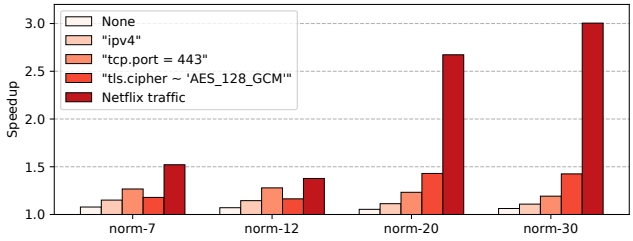


Figure 12: Speedup from Compiled Filter Code—Native code generation can result in 5.4%–300.4% speedups over runtime filter interpretation, depending on filter complexity and the offered traffic. "Netflix traffic" refers to a 32 predicate filter for known Netflix domains and IP prefixes used by Bronzino et al. [16] to collect Netflix traffic features.

Characteristics	Measure	Value	Unit
Packet size	Avg	895	bytes
Fraction of TCP connections	Avg	69.7	%
Fraction of TCP stream bytes	Avg	72.4	%
Fraction of UDP connections	Avg	29.8	%
Fraction of single SYN connections	Avg	65	%
Maximum time between segments in flow	P_{99}	163	sec.
Time to SYN/ACK	P_{99}	1	sec.
Fraction of incomplete flows	Avg	4.6	%
Fraction of out-of-order flows	Avg	6	%
Number of packets per connection	Avg	121	pkts
Number of packets to fill a sequence number hole	P_{50}	1	pkts
Maximum number of packets to fill a sequence number hole in a flow	P_{99}	471	pkts

Table 2: Campus traffic statistics.

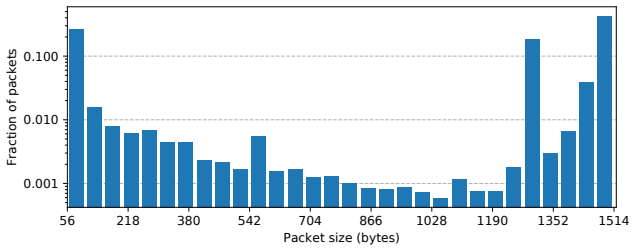


Figure 13: Distribution of packet sizes.

C NETWORK CHARACTERISTICS

At the time of this writing, ingress traffic rates on our university network (aggregated across both links in our monitoring setup) fluctuated between a maximum of approximately 170 Gbps and a minimum of approximately 35 Gbps. Table 2 shows a summary of several traffic characteristics in a 10 minute measurement window on our network. Figure 13 visualizes the distribution of packet sizes on the network. We note that this data is collected through measurement applications developed using Retina itself (with appropriate configurations where necessary, such as turning off inactivity timeouts).