Iris: Expressive Traffic Analysis for the Modern Internet

Thea Rossman Diana Qing Gerry Wan Zakir Durumeric
Stanford University

Abstract

In this work, we investigate the needs of modern traffic analysis, and we introduce Iris, a framework for efficiently building complex, high-performance traffic analysis applications. Iris's key contribution is a compiler that transforms user-defined traffic filters, stream transformations, and computation written in Rust into an optimized processing pipeline. The Iris compiler eliminates redundant logic across analysis tasks to generate a unified runtime that minimizes aggregate work, allowing it to scale to hundreds of concurrent workloads. Rather than restricting users to a domain-specific query language, Iris provides a flexible development environment by exposing connection- and application-layer semantics as Rust data types to user-defined functions. We show that Iris can execute hundreds of analysis tasks concurrently at 100 Gbps+ on a single commodity server, and we demonstrate its flexibility through three use cases drawn from prior work.

1 Introduction

For decades, researchers and operators have analyzed network traffic to understand Internet behavior, security, resilience, and performance. Passive traffic analysis captures the real-world activity of clients, attackers, and infrastructure, yielding insights about security incidents, quality-of-service, protocol evolution, user behavior, and more. Yet, despite an abundance of proposed traffic analysis frameworks [12, 16, 22, 26, 28, 30, 46, 51, 63], many researchers and developers continue to build custom connection analysis pipelines on top of low-level packet processing libraries like DPDK [25, 39, 52, 55, 61, 62, 64, 67]. This is foremost because existing tools face a fundamental tradeoff between expressiveness and performance, which prevents complex analysis of reassembled flows and application-layer sessions on production networks (Section 3).

In this work, we introduce Iris, a software framework that enables both expressive and performant traffic analysis by modeling the problem as a compile-time optimization task. Iris's key contribution is a compiler that transforms user-defined filters, stream transformations, and computation written in Rust into an optimized, end-to-end pipeline. Iris provides a flexible development environment by exposing connection- and application-layer semantics as Rust data types, raising the level of abstraction compared with packet-processing frameworks [23, 41, 45]. Under-the-hood, Iris identifies redundant processing across analysis workloads, which it coalesces into a single optimized runtime.

To ensure that Iris meets real-world needs, we systematize interface requirements through a literature review of 71 recent traffic analysis papers. We find that modern measurement requires complex computation over both common, stream-level abstractions (e.g., application-layer headers) and custom features derived from low-level connection data (e.g., packet inter-arrival times). Iris balances high-level semantics with programmability by allowing users to define arbitrary Rust functions and data structures that filter on, analyze, and compose connection- and application-layer data representations.

Iris builds on the observation that despite a heterogeneity of analysis tasks, many require similar operations and data. Drawing on techniques in compilers and complex event processing [3, 33, 34, 37, 71, 73, 75], Iris decomposes analysis tasks into incremental processing blocks that it later unifies and deduplicates. The Iris compiler transparently integrates application-specific logic with framework primitives (e.g., connection tracking, application-layer parsing) to generate an optimized runtime tailored to the application. The Iris compiler outputs Rust code, delegating instruction-level optimizations and type safety enforcement to the Rust compiler.

We evaluate Iris on a commodity server that receives 95–120 Gbps of mirrored traffic from the Stanford University network (Section 6). This setup—a single rack unit with commodity NICs and no external appliances—reflects typical resources available to research groups that we collaborate with. We show that Iris scales to hundreds of concurrent analysis tasks by merging and deduplicating granular blocks of common logic at compile-time. We evaluate Iris's flexibility by implementing three case studies inspired by our literature review: explorations of end-user security practices [17], censorship techniques and defenses [70], and video quality-of-service [14]. Using Iris, we implement these case studies with $<$250 lines of code each and execute them all simultaneously with under 1% packet loss on 12 cores.

We release Iris under the Apache 2.0 license at https://github.com/stanford-esrg/iris. We hope that Iris provides a high-performance, general interface for building traffic analysis applications over connections, streams, and application-layer sessions.

2 Background and Related Work

There is a long history of network traffic analysis tools. We discuss the most expressive tools used by researchers for analyzing connection- and session-level data, Zeek [46] and Retina [63], as well as the state-of-the art in higher performance, but more rigid, analysis systems.

Zeek. Many researchers leverage Zeek (formerly Bro [46]) for its ability to analyze reassembled flows and application-layer sessions [4, 15, 17, 36, 47, 60, 69, 70]. Zeek provides a rich set of protocol parsers, flow and application events (e.g., "ssl established"), and data types (e.g., "ssl handshake"), as well as a Turing-complete, domain-specific language for writing custom event handlers. Contrasting many intrusion detection systems, Zeek does not simply issue "alerts" or capture packets [22, 51]—it delivers structured data.

Though Zeek facilitates relatively open-ended analysis over high-level abstractions, its primary limitation is performance. Prior work estimates that, assuming perfect scaling, Zeek requires over 100 CPU cores to process 100 Gbps of traffic [17, 31, 58, 63]. While a complete analysis of Zeek's architecture is beyond the scope of this work, we note that Zeek prioritizes dynamic runtime adaptability and a Python-like interpreted DSL (over potential compile-time optimizations) and deep inspection required for broad intrusion detection (over aggressive filtering). In addition, Zeek's scripting language has inherent limitations, including in its type system, library ecosystem, and tooling support [46]. Due to its performance and limited extensibility, we find that researchers primarily use Zeek as a parsing and logging tool, predominantly conducting analysis offline [17, 60, 69, 70].

Retina. Retina [63] offered a step forward in performance over Zeek by aggressively discarding out-of-scope traffic rather than monitoring all packets or dynamically managing multiple events. It provides expressiveness by allowing developers to execute an arbitrary Rust callback on relevant connections and application sessions, which they identify using a Wireshark-like filter. This approach achieved higher throughput, but fundamentally relied on executing only a single analysis function on a single data type within a fixed analysis pipeline. For example, a user could request TLS handshakes or DNS transactions, but not both, nor could they filter on user-defined heuristics. In trying to modify Retina, we found that supporting multiple analysis tasks requires joint optimizations, and extending Iris's filter language and data types requires a more flexible connection processing model.

Packet Analysis Frameworks. While Zeek and Retina provide stream-level abstractions to user-defined functions, other works have proposed optimizing for performance by translating domain-specific query languages to packet-level operations [9, 10, 21, 30, 38, 41, 44, 48, 68, 72, 74]. While these introduce exciting optimization techniques, they are limited in their ability to express flow-level research questions or arbitrary analysis logic. For example, Sonata [30] allows developers to aggregate packets (e.g., by connection five-tuple) but does not support queries that require reassembling bytestreams, and its database-inspired language cannot express open-ended analysis. In addition, many of these systems rely on specialized and programmable hardware, which is impractical for many research groups.

Each of these tools selects different tradeoffs between expressiveness and performance. With Iris, we aim to provide flow- and session-level abstractions, flexibility on par with general-purpose programming, and high performance on commodity hardware across arbitrary concurrent workloads.

3 System Requirements

To ensure that Iris meets real-world needs, we systematized 71 recent research papers that analyze network traffic.[1] In this section, we characterize prior traffic analysis research and establish system requirements for Iris. We conclude that Iris must provide extensibility, expose intuitive abstractions that absorb common processing tasks, and scale to high (100 Gbps+) throughputs across concurrent workloads.

3.1 Literature Overview

To identify relevant papers, one author reviewed 2018–2024 proceedings of SIGCOMM, NSDI, IMC, USENIX Security, and IEEE S&P (identifying 47 papers) and performed a Google Scholar keyword search for "network traffic analysis," "passive network measurement," and "network traffic monitoring" in the same date range (identifying 24 papers from PAM, NDSS, Euro S&P, SIGMETRICS, and CoNEXT). We scoped to works that relied on stream-level abstractions and passively observed traffic from an in-network vantage point, excluding those that used infrastructure statistics, tracked packet-level semantics only, or monitored a single host.

In total, we identified 71 papers, which reflect a diversity of network operations, Internet measurement, security and privacy, and censorship circumvention use cases. About half (32) deployed a system on a production network, of which 65% (21) built a custom application over a packet-level development kit. Ten studies used Zeek, downsampling traffic and typically performing significant analysis offline.

3.2 Requirements

Drawing from our literature review, we identify design requirements for Iris, which we detail below.

3.2.1 Extensibility and Programmability

We first characterize what researchers aim to collect and interpret from network traffic. To do so, we systematize analysis tasks across three dimensions: (1) traffic selection, (2) data extraction, and (3) analysis.

Traffic Selection. Most analysis targets only a subset of traffic, and filtering out-of-scope connections is vital to performance [63]. While most use cases in our study selected traffic by protocol or parsed protocol fields (e.g., "tls," "http.user-agent contains Mozilla"), 12 papers identified relevant traf-

[1] Our literature review is available on GitHub at https://github.com/stanford-esrg/iris.

fic using complex conditions, including timing [69], packet size [47], volume [14], payload patterns [67], and cross-connection state [4, 13, 61]. Such filtering is impossible in Retina's Wireshark-like DSL or Zeek's BPF-based filters. *Iris must allow users to define traffic of interest with arbitrary features in a general-purpose programming language.*

Data Extraction. Researchers extract a diverse set of data representations from network flows, connections, and application sessions. In our analysis, the majority (70%) of studies parsed structured session header fields; however, 65% of papers (most often studying QoE, privacy, or censorship [8, 13, 14, 50, 55, 61, 61, 64, 67]) derived complex features such as packet sizes, inter-arrival times, or byte entropy. About half (48%) of these calculated statistics in windowed chunks of packets, seconds, and bytes (e.g., [14,55,67,69,70]). Anecdotally, researchers indicate that complex data extraction requirements paired with the performance needs of analyzing real-world networks is a key driver for building end-to-end applications on top of low-level packet frameworks like DPDK [23] rather than using existing traffic analysis tools. *Iris must allow users to easily derive custom features from network connections and application sessions.*

Analysis Logic. Analysis tasks vary significantly in what they do with identified connections and extracted data. In our study, we found computation ranging from simple aggregation and logging, to cross-connection and cross-protocol correlation [4, 17, 43, 60], to complex machine learning pipelines [11, 13, 32, 43, 56, 69]. *Iris must support arbitrary analysis on filtered, reconstructed data in a general purpose programming language.*

Cross-Layer Processing. Across the above components, modern traffic analysis requires abstractions that cross traditional OSI boundaries. For example, Xue et. al. [69] classified proxy traffic by comparing inter-arrival times within TCP handshakes, TLS handshakes, and initial packets of TLS ciphertext. Such analysis requires reassembling and parsing payloads (to filter for TLS connections and track handshake boundaries), recording packet arrival times (before reassembly), and feeding the resulting features into a classifier. *Iris must allow users to arbitrarily compose data across the networking stack and connection lifetimes.*

R1: Extensibility. *Iris must provide programmable control over data construction, analysis, and filtering as a first-order component of its interface.*

3.2.2 Easy and Safe Development

Despite the heterogeneity of research, applications share significant logic, including packet capture, load-balancing, hardware configuration, connection tracking, TCP reassembly, payload parsing, and traffic filtering. Like any software framework, Iris must provide modular implementations of such common processing tasks. In addition, many use cases share common data abstractions; in our literature review, the majority (70%) of studies leveraged parsed L5–L7 header fields, such as HTTP user-agents or TLS handshakes (e.g., [4,14,17,36,50,53,60,64]), and most (65%) derived features over flows and connections (§3.2.1). Iris must support such intuitive abstractions out-of-the-box.

Beyond lowering development effort, an extensible framework brings security benefits. Purpose-built systems are often written in low-level languages like C/C++, process untrusted traffic, and operate in privacy-critical settings. Iris should consolidate common logic for easy validation and restrict user-defined code to a memory-safe language.

R2: Developer Overhead. *Iris must expose intuitive abstractions and absorb common processing tasks, leaving developers to focus on application-specific logic.*

3.2.3 Production-Level Performance

Researchers often build custom applications or downsample traffic to investigate campus and ISP-scale network traffic. For example, multiple researchers using Zeek analyzed only 10–15% of traffic on 20–50 Gbps links due to CPU bottlenecks even after modifying the framework to improve performance [17,60,69,70]. Comprehensively analyzing high-speed links enables more accurate results, particularly when attempting to uncover infrequent security phenomenon or understand the "long tail" of Internet traffic. While specialized hardware has shown promise, it is expensive, difficult to customize, and operationally fragile. Scaling server clusters is costly for operators and unrealistic for most research groups.

The studies we analyze require not only scaling a single experiment to high speeds, but also supporting multiple simultaneous, conceptually distinct analysis tasks at line rate. Around half of the papers in our literature review processed multiple data abstractions, traffic categories, or analysis streams. Anecdotally, many research groups also share infrastructure across multiple projects. Iris must support running multiple concurrent analysis tasks without secondary processes, redundant work, or configuration complexity. This introduces a joint optimization challenge, which we discuss in Section 5.

R3: Performance. *Iris must support multiple concurrent traffic analysis tasks at production (100 Gbps+) network speeds on widely-available hardware.*

4 Iris Development Interface

In this section, we present Iris's development interface and show how it enables users to easily analyze traffic for simple use cases (e.g., analyzing SNI values across TLS connections) while simultaneously offering programmable flexibility for building complex analysis pipelines (e.g., extracting custom features across multiple OSI layers). This interface not only provides flexibility to developers by allowing them to interact with connections through native Rust code, but also provides structure for Iris to merge analysis tasks into a single optimized pipeline (Section 5).

4.1 Traffic Subscriptions

Iris is a framework for passively analyzing five-tuple-defined network connections,[2] including reassembled payloads and parsed application-layer headers, from an in-network vantage point. Applications that aggregate and correlate data across connections can be built on top of Iris, much as Iris is built on top of DPDK [23].

Iris developers interact with network connections through *subscriptions* [63]. Each subscription consists of one or more data types (data to analyze), a filter (traffic of interest), and a callback (analysis code). Iris developers build traffic analysis applications by defining one or more (*R3: performance*) subscriptions, which Iris compiles into an optimized runtime that incorporates user logic with connection tracking, reassembly, parsing, and other generic framework utilities. In simple use cases, users can rely on Iris-defined datatypes and a Wireshark [65]-like filter language (*R2: dev overhead*). For example, only a few lines of code are needed to capture HTTP requests with a specific user agent:

```
#[callback("http.user-agent contains 'Mozilla'")]
fn log_http(http: &HttpRequest, ft: &FiveTuple) {
    log::info!("{}: {}", ft.src_subnet(24), http);
}
```

While simple use cases are similar to Retina [63], Iris supports complex applications by allowing users to define an arbitrary number of subscriptions, implement filter and data construction logic in Rust, and stream data within a connection (§2).

4.2 Complex Data, Filters, and Callbacks

Iris supports complex use cases by allowing developers to define all aspects of a subscription (data types, filters, and callbacks) in Rust (*R1: extensibility*). Mechanically, Iris models connections as a set of protocol-specific state machines, which developers hook into to attach state and computation to connection-processing logic. We design these state machines to both reflect the most common abstractions identified in our literature review (§3) and enable compile-time decomposition (§5). In particular, Iris allows developers to define Rust functions and data structures over parsed L6–L7 headers (required by 70% of works in our review) and streaming connections (65% of works) (*R2: dev overhead*). Connection state machines also provide structure for the Iris compiler to identify shared incremental logic across subscriptions and generate a single optimized pipeline.

In this section, we introduce the Iris state machines and use a running example to describe custom data types, filters, and callbacks. We elaborate on Iris's interface in Appendix B.

State Machine Representation. Iris processes packets in a connection as they arrive, advancing connections through a set of extensible protocol state machines informed by protocols (e.g., TCP state machine). To develop filters, data types, and callbacks, developers can "hook" into both states and state transitions to extract data, perform additional computation, or attach state for later use. Table 1 shows the state transitions that Iris currently exposes. Because each layer exposes its state machine to others, this model is composable (e.g., to support encapsulation) and extensible (e.g., to add protocols).[3]

Name	Description	Data
Layer-4 (TCP or UDP)		
First Packet	First packet in TCP or UDP connection (TCP SYN/first observed UDP).	Packet (L4Pdu)
End of Hshk.	TCP handshake completed.	
L4 Conn.	New observed packet in a TCP/UDP connection.	Packet (L4Pdu)
Reass. Stream	New reassembled segment in a TCP connection.	Packet (L4Pdu)
Conn. Terminated	Connection terminated or timed out.	
Layer 5–7 (Encapsulated in TCP or UDP)		
L7 Proto. Discovered	An L6/L7 parser identified a protocol (e.g., HTTP, TLS).	Protocol
End L7 Headers	An L6/L7 module parsed protocol-specific "headers" (e.g., TLS handshake).	Parsed Headers
L7 Headers	New packet or reass. segment (specify) in L6/L7 Headers.	Packet (L4Pdu)
End L7 Session	An L6/L7 parser identified the end of an L7 session (e.g., start of a new pipelined HTTP request).	Session Data
L7 Payload	New packet or reass. segment (specify) in L6/L7 payload (e.g., HTTP response body, TLS ciphertext).	Packet (L4Pdu)

Table 1: Primitive Data Types. Iris models each connection with protocol-specific state machines. State transitions function as both hooks for user logic and primitive data types that user-defined functions build upon. This model is composable and extensible to other protocols and layers. We select these as Iris's base abstractions based on the most common use cases identified in our literature review (§3).

Developers hook into connection processing by annotating structs and functions with procedural macros that Iris exports. Procedural macros enable compile-time code generation in Rust, allowing us to inspect and transform application source code without implementing a compiler from scratch.

Use Case Example. To concretely show Iris' interface, we illustrate building an Iris application inspired by Bronzino et al.'s video QoE inference work as a running example [14]. Our Iris application filters for video streams, derives per-connection features in a sliding window, and uses these features to estimate video resolution. We track each video

[2]We define a "connection" as bidirectional TCP or UDP packets associated with the same five-tuple until a configurable inactivity timeout or, for TCP, a FIN/ACK sequence or RST.

[3]For brevity, we refer to protocols encapsulated in TCP or UDP as "L7" (or "application-layer").

```
// A custom Iris data type
#[datatype]
struct FeatureChunk { /* ... */ }

impl FeatureChunk {
    // All data types must implement a constructor
    fn new(first_pkt: &L4Pdu) -> Self { /* ... */ }
    // Request each new L4Pdu in the L4 connection
    // and update self
    #[datatype_fn("FeatureChunk,InL4Conn")]
    fn update(&mut self, pdu: &L4Pdu) {
        // Wait 60s at start of connection
        // Then update features for last 10s sliding
        ↪ window
    }
}
```

Listing 1: Data Type for §4.2 Working Example. A data type that extends an Iris primitive abstraction (streaming IP packets in a connection) to derive custom features (e.g., throughput). The `ResolutionEst` callback in Listing 3 subscribes to this type.

stream's inferred resolution over time, accumulating new estimates every 10 seconds after the first 60 seconds of each connection. We implement this by defining:

- A `FeatureChunk` data type that transforms a stream of connection frames into QoS features;
- A `ResolutionEst` callback that feeds each updated `FeatureChunk` into a classifier to infer video resolution;
- A Wireshark-like filter for "tls" traffic matching a set of SNI names, provided in a text file;
- A custom filter, `video_lookup`, that matches new TLS connections with recent DNS transactions in a global cache.

We describe each subscription component (data type, filter, and callback) in the remainder of this section.

Data Types. Applications can define Iris "data types" that build higher-level abstractions over low-level primitives (Table 1 lists these primitives). By annotating a Rust struct as an Iris data type, a developer can later access it in filters and callbacks. This separation of concerns—data construction from filtering and callback analysis—provides a modular, object-oriented programming model. It allows multiple applications to share implementations of common abstractions, and it allows the Iris compiler to share work across subscriptions, as we describe in Section 5 (if multiple callbacks and filters require the same data type, Iris constructs it once).

To build our video QoE application, our `FeatureChunk` data type (Listing 1) requires two functions: (1) an initial constructor, which Iris invokes at the beginning of each connection (`FirstPacket` in Table 1), and (2) a function that receives an update at each new packet in the connection (`L4Conn`). These functions derive and accumulate data over the lifetime of a connection, which our callback (Listing 3) subscribes to for analysis.

Filters. Developers define filters to identify relevant connections. However, filters are also critical for performance: they allow Iris to discard out-of-scope data, traffic, and operations at multiple stages in connection processing. Iris supports an extended version of Iris's Wireshark-like filter language [63], which can express protocols and protocol fields. Unlike in prior work, developers can customize this language by defining custom filter predicates in Rust. Filters can be stateless functions or, like data types, attach state to a connection (Appendix B includes an example of a stateful filter).

```
// An Iris callback that receives all DNS
// transactions and caches some subset globally.
// This callback is stateless over the connection,
// but accumulates global state across invocations.
#[callback("dns")]
fn dns_record(dns: &DnsTransaction) {
    /* If video service, cache DNS mapping */
}

// A custom Iris filter predicate.
// Requests Iris data types, performs custom logic,
// and returns a match result.
#[filter]
fn video_lookup(&mut self, tls: &TlsHandshake, ft:
↪ &FiveTuple) -> FilterResult {
    /* Look up IP in DNS cache, return Accept/Drop */
}

// Omitted in `main`: init global data structures
```

Listing 2: Custom Filter for §4.2 Working Example. An Iris filter that associates TLS connections to previous DNS requests. The `dns_record` subscription populates a global, user-defined DNS cache (omitted for brevity), which the `video_lookup` function uses to filter connections. Iris will apply this `video_lookup` filter to a connection to determine whether to invoke the `ResolutionEst` callback (Listing 3).

For our video QoE example, we define a custom filter predicate that correlates TLS flows to preceding DNS queries. This requires cross-connection state, which cannot be expressed in our domain-specific filter language. We thus define (1) an additional subscription to DNS transactions, which populates a global DNS cache, and (2) a filter that references this cache (Listing 2).

Callbacks. Iris callbacks execute arbitrary Rust code on Iris data types for connections that meet filter conditions. Iris invokes each callback function as early as possible, facilitating real-time analysis. Because immutable access is sufficient for most passive analysis tasks—which aggregate and record data without modifying it—Iris gives callbacks read-only access to parameters, avoiding deep copies.

For our video QoE example, we define a stateful callback, `ResolutionEst` (Listing 3), which streams `FeatureChunks` within a connection. To track resolution over time, we define the callback as a Rust data structure that records resolutions for each window in a vector. This requires three functions: a constructor, an "update" function (to receive and process `FeatureChunks`), and an "end" function (to save

```rust
// An Iris callback with associated filter string.
// Filter strings can refer to protocol modules,
// text files that list predicates, or
// the names of custom filter predicates.
#[callback("tls and (file=video_sigs.txt or
↪  video_lookup)")]
struct ResolutionEst {
    // Vector of observed resolutions for each
    // window in the connection.
    resolutions: Vec<usize>,
}

impl ResolutionEst {
    // Stateful callbacks must implement a constructor
    fn new(first_pkt: &L4Pdu) -> Self { /* ... */ }
    // Request each new FeatureChunk in the conn.
    // Iris invokes after each update to FeatureChunk.
    #[callback_fn("ResolutionEst,L4Conn")]
    fn update(&mut self, feats: &FeatureChunk) -> bool
    ↪  {
        /* if at end of window, invoke classifier,
        ↪  store result in self.resolutions */
        /* optionally return false to stop receiving
        ↪  data for connection ('unsubscribe')*/
    }
    #[callback_fn("ResolutionEst,L4Terminated")]
    fn end(&mut self) {
        /* save self.resolutions (e.g., to disk) */
    }
}
```

Listing 3: Callback for §4.2 Working Example. A stateful Iris callback that receives updated `FeatureChunk` data throughout a connection and an alert on termination. This subscription filters for connections that are TLS and match either the custom predicate `video_lookup` (Listing 2) or a filter read from an input file (omitted; contains SNI values of known video platforms).

predictions to disk when the connection terminates). Our `ResolutionEst::end` function has no arguments, so we add the `L4Terminated` tag to specify that Iris should invoke it when an L4 connection terminates.

End-to-End Application. An Iris application consists of one or more subscriptions, which attach computation and data structures to Iris's connection-processing state machines. As we describe in the next section, Iris compiles these subscriptions into an end-to-end traffic analysis pipeline built around these state machines.

5 Iris Architecture

Iris achieves high throughput across a large number of concurrent subscriptions by compiling them into a single, optimized runtime pipeline. The Iris compiler leverages the state-machine-based development interface described in Section 4 as scaffolding for a unified intermediate representation (IR) that captures the data construction, filtering, and callback invocation requirements of all subscriptions. Iris's interface makes each subscription statically analyzable, allowing the compiler to reason about subscriptions in terms of incremental requirements and filters in per-connection state machines.

After building an IR for each subscription, Iris combines and optimizes across them—fusing shared filter prefixes, deduplicating computation, and eliminating redundant state. Iris then generates Rust code for a runtime that advances connections through layered per-connection state machines, applying incremental filter predicates at state transitions to determine how and if to continue processing the connection (Figure 1). The resulting runtime is tailored to the subscriptions in the Iris application, correctly fulfilling each one while aggressively sharing computation and data across them.

In summary, the Iris compiler proceeds as follows:

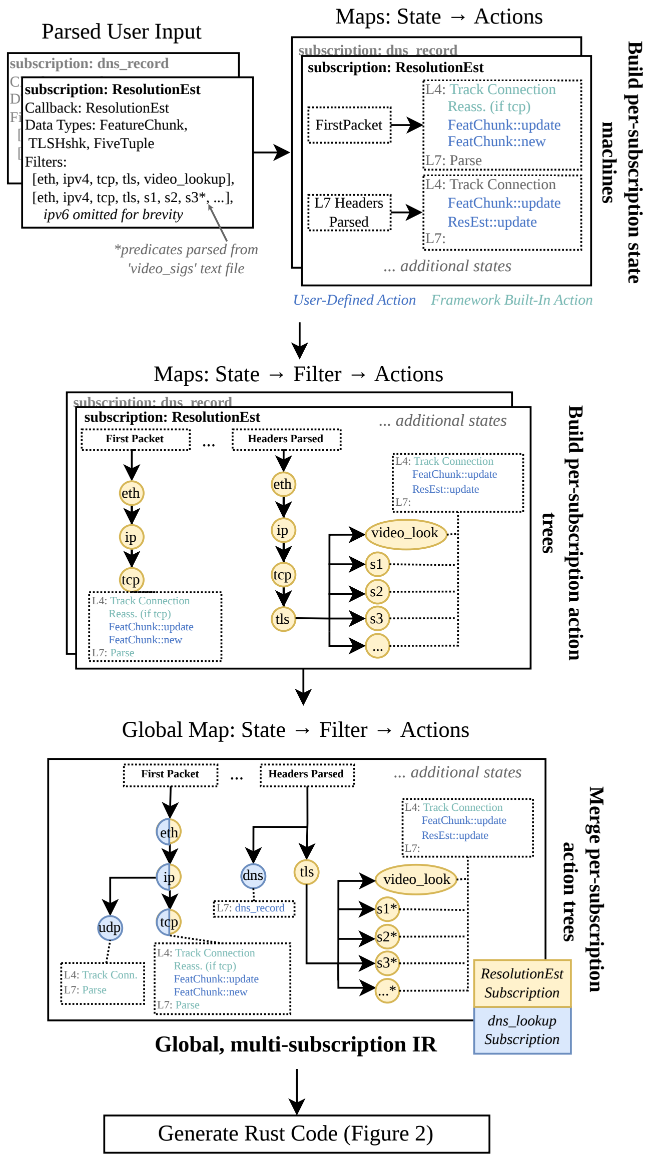
1. **Construct Per-Subscription State Machines.** Iris extracts the data types, filters, and callbacks from each subscription and translates these components into a per-connection state machine with attached "actions" that incrementally fulfill a subscription.
2. **Build Per-Subscription Match-Action Pipeline.** For each connection state, Iris builds an "action tree" that maps incremental filter patterns to subsequent actions. At runtime, Iris invokes action trees to determine if and how to continue processing a connection.
3. **Fuse Action Trees.** Iris combines per-subscription action trees into global, per-state action trees, unifying shared filter prefixes and computation at each state transition. It optimizes each action tree by removing redundant nodes and (where possible) reordering predicates.
4. **Generate Code.** Iris generates Rust code, producing a specialized "match-action" connection processing pipeline that fulfills all subscriptions.

The state machine-based connection model is critical to Iris's IR and runtime. We note that Iris maintains multiple state machines for each connection as needed (e.g., TCP connection and HTTP state machines). Within each state machine, a connection is in at most one state and follows a strict ordering, but ordering is not guaranteed across state machines.[4] For brevity, we refer to each entry in Table 1 as a "state transition", including self-loops (e.g., *In L4 Conn* may terminate the connection or leave it in the same state).

5.1 Compiling One Subscription

We first detail how Iris constructs its IR for a single subscription, using the §4.2 video resolution application as a working example. At compile time, Iris represents each subscription as a "match-action pipeline" built around its per-connection state machines. Mechanically, Iris builds a map from connection states to incremental filters (match conditions on connection data) and actions (computations required to correctly fulfill the subscription). Actions range from generic framework operations (e.g., reassemble), to storing data (e.g., cache a TLS handshake for a future callback), to invoking a user-defined API, and include both operations to perform immediately (e.g., invoke a callback) and streaming operations to maintain

[4]For example, a connection can, by definition, be in both *L4Conn* and *L7Headers*.



Parsed User Input
subscription: dns_record
subscription: ResolutionEst
Callback: ResolutionEst
Data Types: FeatureChunk, TLSHshk, FiveTuple
Filters:
[eth, ipv4, tcp, tls, video_lookup],
[eth, ipv4, tcp, tls, s1, s2, s3*, ...],
ipv6 omitted for brevity
*predicates parsed from 'video_sigs' text file
Maps: State → Actions
FirstPacket
L4: Track Connection
Reass. (if tcp)
FeatChunk::update
FeatChunk::new
L7: Parse
L7 Headers Parsed
ResEst::update
L7:
... additional states
User-Defined Action
Framework Built-In Action
Build per-subscription state machines
Maps: State → Filter → Actions
First Packet
Headers Parsed
eth
ip
tcp
tls
video_look
s1
s2
s3
...
Build per-subscription action trees
Global Map: State → Filter → Actions
udp
dns
L7: dns_record
L4: Track Conn.
s1*
s2*
s3*
...*
ResolutionEst Subscription
dns_lookup Subscription
Merge per-subscription action trees
Global, multi-subscription IR
Generate Rust Code (Figure 2)

Figure 1: Iris Compiler Steps. The Iris compiler decomposes each subscription into a map from *states* in per-connection state machines to *actions* required at each state. It then incorporates subscription *filters*, adding *match conditions* to actions at each state transition. Finally, it *merges* these match-action components together and generates Rust code that shares incremental filtering, data, and computation across all subscriptions. This figure illustrates these steps for the working example from §4.2.

through a subsequent state (e.g., continue tracking a connection). This match-action representation allows the Iris compiler to subsequently unify incremental requirements across subscriptions (§5.2) and generate code that eagerly discards out-of-scope data and computation to minimize unnecessary computation and memory (§5.3).

Build Per-Subscription State Machines. While data types, filter predicates, and callbacks have different semantics, they are fundamentally similar: each exposes one or more APIs that Iris must invoke during connection processing. Iris starts by decomposing each subscription into these user-defined APIs. Iris then maps each API to the connection state machine (i.e., at which state transition(s) the API should be invoked) by determining when its parameters can be delivered and, for callbacks, filter predicates applied. Iris infers each function's state recursively, with recursion ending at primitive data types (listed in Table 1). This process produces a mapping from each state to the user-defined APIs that can be invoked at that state. For example, Iris maps *L7EndHeaders* to *video_lookup* and *L4Conn* (new packet in connection) to *FeatureChunk::update* and *ResolutionEst::update* (Listings 1, 2, 3).

Iris next adds framework actions to its state-to-action mapping. The state machine abstraction makes it straightforward to infer these framework actions. Each state exposes an API indicating the sequence of framework operations required to advance a connection to that state, and Iris must advance connections to all states with user-defined APIs associated. For example, reaching *L7EndHeaders* to parse the TLS handshake for *video_lookup* (Listing 2) requires reassembly through L7 protocol discovery and headers.

Construct Action Trees. The previous step determined which actions need to take place at each state transition to fulfill a subscription. Next, Iris incorporates the subscription's filter to determine which connections to apply these actions to. At each state transition, Iris builds an *action tree*, in which each node contains a filter predicate and associated actions. At runtime, action trees indicate decision points: if a connection matches a root-to-node path (filter prefix), the Iris runtime should apply a corresponding set of actions (those accumulated along the path).

Like Retina [63], Iris expands each filter into predicate vectors ("filter patterns") and re-orders predicates based on when in a connection's lifetime each can be applied. For example, Iris expands *"tls.sni contains 'nflxvideo"'* into *ethernet* $\rightarrow$ *ip* $\rightarrow$ *tcp* $\rightarrow$ *tls* $\rightarrow$ *tls.sni contains 'nflxvideo'*. We note that, if a developer provides an input text file with additional filter patterns (e.g., Listing 3 provides SNIs in a file), Iris parses and expands it at compile time.

Iris iteratively builds an action tree for each state by inserting filter prefixes—the predicates applicable at that state—and storing the actions identified in step #1 at leaf nodes. At each state, a root-to-leaf branch may indicate an end-to-end pattern that triggers a callback or an initial prefix that dictates a subsequent processing step. For example, fulfilling the *ResolutionEst* subscription requires actions at *FirstPacket* only if the connection fulfills the initial filter prefix *eth* $\rightarrow$ *ip* $\rightarrow$ *tcp*.

Record Tracked Data. The above-described action trees form the core IR of Iris's match-action pipeline. While constructing them, Iris also identifies data that the runtime must track on its per-connection data structure. Per-connection "tracked" data includes user-defined data structures (e.g., the

FeatureChunk data type), stateful framework modules (e.g., TCP reassembly), and information that Iris must cache for future use (not applicable to the *ResolutionEst* example).

5.2 Multiple Subscriptions

Many Iris applications require multiple subscriptions. A naive multi-subscription runtime pipeline would process each independently, unnecessarily duplicating many of the most computationally and memory intensive operations such as connection tracking and TCP reassembly. Instead, Iris constructs a runtime pipeline that *shares as much work as possible across subscription components*. For example, if multiple subscriptions require reassembly, Iris should reassemble once (Figure 3). Iris performs multi-subscription optimization at compile time by combining per-state action trees and tracked data from all subscriptions.

Combine Action Trees. Iris generates a single action tree for each state by merging per-subscription action trees based on shared filter prefixes and actions. Then, Iris prunes sub-branches that contain no new information (e.g., child node with actions already present at its parent) and prefixes that do not drop data or disambiguate between subscriptions. When Iris cannot remove nodes, it orders predicates with respect to generality (e.g., "tcp" is parent to "tcp.port = 80") and marks mutually-exclusive sibling nodes with an *xor* tag (e.g., "tcp.port = 80" and "tcp.port = 443" cannot both be true at the same layer), to build a tree that minimizes the predicates that Iris must apply to each connection at runtime.

Combine Tracked Data. As described in Section 5.1, Iris also accumulates a per-subscription set of data that Iris must track on a connection. Iris combines these across subscriptions into a global set, ensuring that each stateful component is tracked exactly once. For example, if two callbacks require the same data type, Iris will construct it at most once and give each callback read-only access.

By merging action trees and tracked data, the Iris runtime avoids unnecessary computation, eagerly discards data, and correctly disambiguates between subscriptions (i.e., only invokes a callback if its associated filter matches).

5.3 Generated Code

The Iris compiler produces an IR of control flow (action trees) and state (tracked data). Iris generates packet- and connection-processing logic that directly interacts with framework and user-defined APIs, relying on the Rust compiler to implement low-level performance optimizations and safety guarantees.

5.3.1 Runtime Overview

We describe the key components of a typical generated Iris runtime. Figure 1 illustrates a simplified version of the runtime for the video resolution example from Section 4.2. Recall that the runtime pipeline that the Iris compiler generates depends uniquely on the subscriptions in each application.

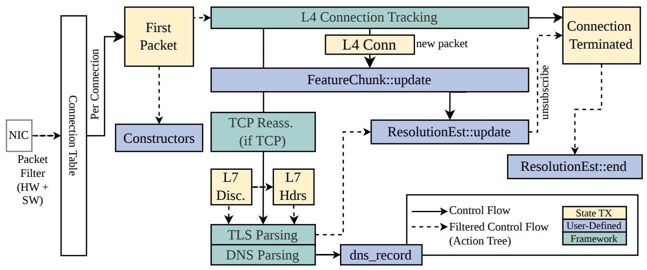


Figure 2: Iris Runtime for §4.2 Working Example. Iris advances connections through a match-action pipeline generated by the compiler and structured around per-connection state machines.

We note that Iris draws high-performance implementations of some generic primitives from Retina [63]: protocol parsers, lightweight TCP reassembly, DPDK integration, NIC hardware filtering, and a connection timeout mechanism. We leverage these runtime similarities to benchmark Iris against Retina in Section 6. Compared to Retina, Iris supports an extensible filter language, expressive data model (as hooks into state machines), and scalable multi-subscription architecture (through joint compile-time optimizations).

Initial Packet Processing. The Iris compiler generates a per-packet filter, which the runtime tries to offload to the NIC at startup to reduce CPU load. The runtime immediately drops any packets that do not match this filter (in Figure 2, packets that are not TCP or UDP), short-circuiting all connection processing. Symmetric RSS [66] shards packets across cores, allowing zero-lock connection tracking [27]. To bypass the kernel networking stack and reduce copies [10], Iris uses DPDK [23] to pull raw packets directly from the NIC to user-space memory.

Connection Tracking. Each core maintains a hash table with a per-connection data structure keyed by UDP or TCP five-tuple [27] and configurable with establishment and inactivity timeouts to bound memory usage [63]. Iris associates each incoming packet with its connection data structure, creating an entry if none exists. This per-connection data structure is a Rust struct generated by the Iris compiler. By inferring types and generating a struct with associated methods at compile-time, Iris avoids expensive dynamic dispatch and allows the Rust compiler to inline and optimize function calls.

State Machines. On the first packet of each connection, the Iris runtime initializes state machine handlers: in our example (Figure 2), one for Layer-4 and one for Layer-7. As new packets arrive, each handler advances each connection through its state machine. For example, in Figure 2, Iris's L4 handler reassembles packets and passes them to the L7 handler, which executes its first state transition when a protocol parser identifies a connection as TLS, DNS, or neither.

Each state machine handler exposes an API for each state transition, which executes action tree logic. The Iris compiler translates action trees (§5.1) to native Rust code. These state transition functions parse headers and access connection data, apply filter predicates (as `if` blocks), invoke functions inline, and accumulate ongoing actions on a connection. For example, the *FirstPacket* state transition depicted in Figure 2 parses packet headers (Ethernet, IPv4/IPv6, and TCP/UDP), invokes constructors, and marks the connection with ongoing actions (track, reassemble, *FeatureChunk::update*, and parse).

We note that, because Iris stores actions as per-connection metadata (see below), it need not invoke action trees on each new packet. The Iris compiler generates a pipeline that invokes action tree logic only when actions might change. As shown in Figure 2, Iris excludes irrelevant state transitions altogether (e.g., TCP handshake tracking). It only invokes the self-transition on *InL4Connection* if the *ResolutionEst::update* user-defined function unsubscribes.

Executing Actions. As Iris advances a connection through state machines, it invokes actions. There are two types of actions: one-shot functions that the runtime can invoke within a state transition (e.g., *dns_record* in Listing 2) and streaming actions that the runtime may sustain over multiple packets in a connection (e.g., reassembly, streaming to a user-defined function like *ResolutionEst*). For the former case, the Iris compiler inlines each function call. For the latter, Iris updates a bit vector stored on its per-connection data structure representing "current actions", which the Iris runtime checks to determine how to process each new packet. If a state transition function returns with no ongoing actions, Iris removes the connection from its tracking table.

5.3.2 Multiple Subscriptions

Any single-subscription runtime pipeline is relatively simple: at each state transition, the Iris runtime "drops" a connection or "continues" processing it with a predetermined set of actions. However, developers must be able to leverage Iris to express any number of concurrent subscriptions. As described in Section 5.2, the Iris compiler combines filter predicates, actions, and tracked data to share work and computation across subscriptions.

Figure 3 illustrates how these compile-time reduction techniques dramatically reduce the overhead of adding an additional subscription. In our video resolution example application, *ResolutionEst* and *dns_record* are different subscriptions, but both share significant work: packet processing, connection tracking, reassembly (for tls and dns over tcp), and application-layer parsing. By combining subscriptions at compile-time, Iris generates a single, optimized pipeline that fuses computation and data across an arbitrary number of subscriptions.

5.3.3 Rust Compiler

Generating Rust code has two key benefits. First, native code outperforms the dynamic data structures and function pointers that Iris would otherwise need to represent filter predicates [63] and heterogeneous data types [18, 35]. Second, Iris offloads significant work to the Rust compiler, relying on it to

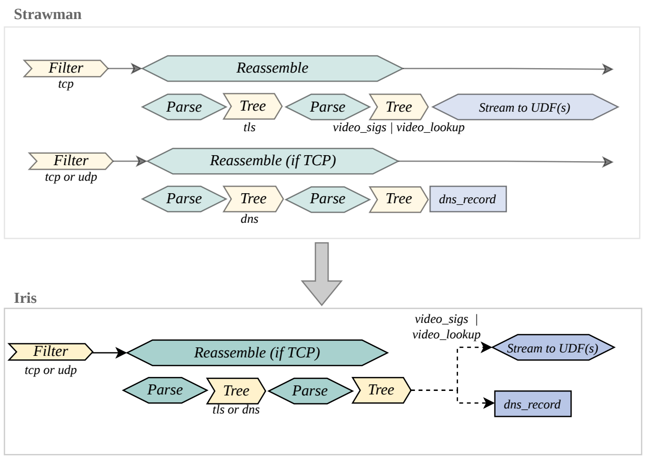


Figure 3: Pipeline for Two Subscriptions. We show two example subscriptions: (1) the *ResolutionEst* subscription and (2) the *dns_record* subscription from §4.2. A naive approach would execute each subscription separately. In contrast, Iris compiles a single pipeline that identifies and combines shared data and operations, using action trees built from filters to determine next steps when requirements diverge.

implement low-level CPU- and memory-level optimizations and enforce safety, typing, and isolation. This comes with two drawbacks: runtime flexibility and compilation time. Adding or removing subscriptions requires recompiling and restarting the application, though users can define functions that branch on global state. For example, the *video_lookup* function in Listing 2 queries a dynamically-updated global DNS cache. We measure compilation time in Section 6.5. Overall, we believe that trading compile-time complexity for runtime performance is reasonable, especially for many researchers.

6 Evaluation

Iris's flexibility makes it challenging to evaluate; network traffic and application characteristics fundamentally shape its performance. The number of subscriptions requested, memory and compute load of each subscription, amount of shareable work (which the compiler consolidates), and what proportion of traffic the runtime is able to filter out all impact throughput. Consequently, we choose to evaluate Iris in three ways. First, we evaluate Iris against Retina, the current state-of-the-art in performance for a single subscription (§6.1), for two applications that both support. Second, we show that the Iris compiler effectively consolidates shared work, allowing Iris to scale to a large number of concurrent subscriptions (§6.2). Finally, we demonstrate that Iris can support running complex real-world experiments on a production network by implementing three use cases inspired by prior work (§6.3).

Deploying Iris requires a commodity server with a DPDK-compatible NIC [23]. Our testbed uses dual Xeon Gold 6248R CPUs (24 cores, 3GHz), 384 GB DDR4-2933 ECC memory (12×32 GB DIMMs), and two 100 GbE Mellanox ConnectX-

5 NICs. We receive mirrored traffic from a large University network and run experiments at peak hours, observing 95–120 Gbps of traffic. Like Wan et al. [63], we use zero-loss throughput (ZLT) as our key metric. ZLT, the amount of traffic that Iris can process without dropping packets, is a measure of Iris's practical performance for real-world measurement. We measure ZLT by iteratively running an Iris application and decreasing the flow sampling rate until a trial achieves <1% packet loss after a two-second warm-up period. Note that achieving a meaningful ZLT measurement requires restricting the number of RX cores available until Iris is unable to sustain the throughput available. We also benchmark memory and compilation time in §6.4 and §6.5.

Our experiments use TCP and UDP inactivity timeouts of 5 minutes and 1 minute, respectively, as well as a TCP establishment timeout of 5 seconds (to quickly discard unanswered SYNs). We run our benchmark experiments (§6.1 and §6.2) for five minutes and our case studies (§6.3) for one hour. We received approval for this work from our University's Privacy and Security office, and we anonymized and aggregated any traffic data written to disk. We describe additional ethical considerations in Appendix A.

6.1 Single Subscription

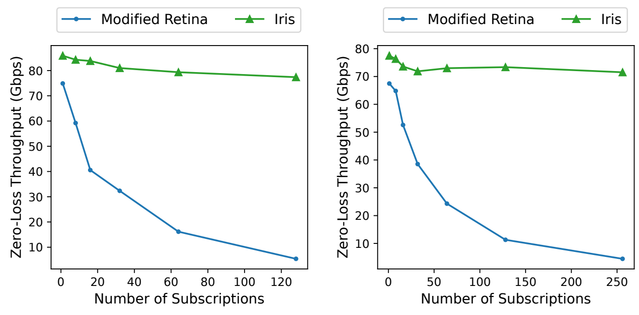
We first evaluate Iris against Retina [63] for two single-subscription applications supported by both frameworks. Retina is a useful baseline for two reasons. First, while Retina limits users to a single subscription, it can execute that subscription at 100 Gbps+ on commodity hardware. Second, Iris draws implementations of key primitives from Retina (DPDK integration, connection tracking, reassembly, and parsing).

We select two use cases in which the runtime that Iris generates is comparable to Retina's and run them for five minutes each. One collects TLS handshakes and the other a set of connection features (e.g., TCP flags, packet interarrival times, flow duration). Iris does not introduce overhead compared to Retina, slightly outperforming the latter by 5.52–9.89% on average. Both achieve zero loss at 6 cores. We point to Wan et al.'s benchmarks against Suricata, Zeek, and Snort [63]: for a subscription to a single TLS connection, Retina provides 5–100× throughput per core compared to prior work.

6.2 Multiple Subscriptions

We next stress Iris's ability to scale to a large number of concurrent subscriptions and evaluate our match-action architecture. Because no existing system supports multiple connection-based analysis tasks at 100 Gbps+, we modify Retina to support multiple subscriptions without a fundamental rearchitecture. Our goal is to show how a *naive* approach to multiple subscriptions scales compared to Iris.

Baseline. As noted in §6.1, Iris shares low-level traffic processing primitives with Retina, yielding similar runtime behavior for single-subscription applications. Iris optimizes across multiple subscriptions by deduplicating shared work



(a) Each subscription filters for an IP subnet and subscribes to a connection, exercising the systems' long-term connection-tracking logic. Four cores.

(b) Each subscription filters for a malicious HTTP host and requests an HTTP transaction, exercising the systems' reassembly and parsing logic. Two cores.

Figure 4: Multi-Subscription Benchmarks. Iris minimizes overhead across subscriptions by sharing operations, data, and filter predicates; avoiding per-subscription state; and inlining functions. In contrast, a naive implementation replicates shared logic, manages each subscription separately, and uses dynamic dispatch to invoke functions. We conduct two experiments to exercise different aspects of Iris.

and data and tracking "actions" rather than individual subscriptions. We modify Retina to naively support multiple subscriptions without these optimizations.

Our naive baseline shares core framework infrastructure across subscriptions: packet-processing, connection management, TCP reassembly, and protocol identification. However, it duplicates most per-*subscription* logic. It applies each filter predicate and constructs each data element separately for each subscription, regardless of shared work. Rather than generating a struct with named fields at compile-time, our naive implementation maintains a vector with dynamic dispatch that stores different types. Our implementation also tracks subscriptions using identifiers, rather than collapsing shared behavior into "actions."

Tracking Connections by IP Address. We first subscribe to derived features for N=1, 2, 4 ..., 128 IP subnets. We shard the IPv4 address space across subscriptions (i.e., each subscription processes a decreasingly sized portion of the IP space) to evaluate scalability while keeping the overall amount of matched traffic consistent. As can be seen in Figure 4a, Iris scales from 8 to 128 connection subscriptions with only 8.2% loss in throughput. In stark contrast, our naively-modified Retina immediately drops in throughput as redundant work increases, operating at only 5.41 Gbps for 128 subscriptions (versus 77 Gbps for Iris).

Parsing Sessions. We next configure each system to search for HTTP requests to malicious domains drawn from Abuse.ch [1]. Our naive Retina-based approach sees a significant (93.1%) drop in throughput to just 4.5 Gbps for 256 subscriptions. Iris scales to 256 subscriptions on just two

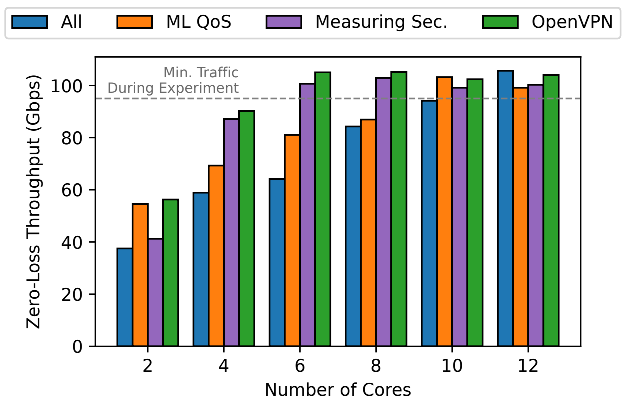


Figure 5: Research Case Studies. We implement the use cases described in §6.3 in Iris and deploy them on an enterprise network. These examples present diverse callback, data type, and filtering requirements, illustrating Iris's flexibility for real-world use cases. Iris can process all of the applications concurrently at 100 Gbps+ with just 12 cores. (Variations above 98 Gbps are due to traffic fluctuations.)

cores with only 6.3% throughput reduction (Figure 4b).

Analysis. To understand the most significant sources of overhead, we estimate latency for similar parts of the systems' codepaths using BPF hooks. We focus on areas where we expect our naive approach to see the most duplicated work: applying filters and updating data types. Our BPF hooks fire when a function is called and when it returns, recording timestamps at each to estimate latency. With 128 IP subscriptions, our naive approach takes 5× longer in each packet filter and 11.5× for each data update than Iris. For each packet, the naive solution iterates through subscription filters and sets a match flag per subscription. Iris applies a single "ipv4" filter by merging consecutive IPv4 subnets and simply returns true/false, deferring disambiguation until the subscriptions diverge. When updating subscription data, our modified Retina replicates similar logic across 128 subscriptions. For a single subscription, these components are similar (within 8%) for Iris and the naive implementation. For our HTTP experiment, we see the most overhead from cloning data to transfer ownership to user callbacks and parsing packet payloads.

Despite sharing packet-processing, connection tracking, and reassembly infrastructure, a naive approach to managing multiple subscriptions rapidly degrades at scale. Iris, in contrast, incurs only marginal overhead from additional similar subscriptions, indicating that our compiler effectively amortizes shared work.

6.3 Case Studies

We next evaluate Iris's practicality and performance for conducting complex traffic analysis on production networks. We implement three use cases from prior work that exercise different aspects of Iris: an analysis of user security practices and malware downloads [17], a fingerprinting attack on OpenVPN [70], and the video resolution estimator described in §4.2 [14]. We are able to run all applications, both independently and concurrently, at 100+ Gbps throughput (the maximum amount of traffic available on our network) for an hour each with 6–12 cores (Figure 5). We describe each below:

Measuring Security Practices. DeKoven et al. (2019) inferred security-related behaviors (e.g., downloading software updates and using password managers) and incidents (e.g., malware infection) from campus network traffic [17]. The original work deployed a Bro [46] cluster, augmented with an optimized packet-processing layer, to collect parsed application-layer records and anonymized five-tuples, and applied application fingerprints and correlated data offline. The authors use Zeek primarily as a parsing and logging tool (i.e., apply all application fingerprints offline) and report that their system could handle 4–6 Gbps.

We replicate their data collection pipeline in 227 lines of code. We use a custom filter to drop suspected high-volume, internal, and IoT traffic, and we define subscriptions to HTTP, TLS, QUIC, DNS, and anonymized connection five-tuple abstractions. Despite parsing diverse application-layer headers and managing a variety of subscriptions, our implementation in Iris achieves 100+ Gbps using only 6 cores.

Fingerprinting OpenVPN. Xue et. al. evaluate fingerprinting attacks and defenses on OpenVPN traffic [70]. The authors develop two fingerprints, each based on the first 150 packets of a TCP or UDP connection, analyzing packet sizes or payload prefixes to identify OpenVPN ACKs and OpCodes, respectively. The original authors used a Zeek cluster (enhanced with a zero-copy packet layer), sampled 1/8 of flows from 20 Gbps of traffic, and applied their fingerprints offline.

In Iris, generating and subscribing to these fingerprints for all TCP and UDP connections requires 194 LOC. We leverage Iris's programmability to extract highly-custom patterns from grouped connection data. Because our callback "unsubscribes" from connections after receiving fingerprints (i.e., after the first 150 packets), Iris can quickly remove long-lived connections from its connection table. Our implementation achieves ZLT for 100 Gbps with 6 cores.

Machine Learning on Connection Features. We implement the video resolution estimation from [14], a simplified version of which appears in Section 4.2. The original work evaluated models with multiple feature combinations; for our experiment, we use their reported top-ten most significant features. We train a random forest classifier on the authors' original training data (700K labeled samples). The original authors built a fixed-function tool for the experiment. While we do not have access to their code, we note that it was an early version of traffic refinery [13], which is implemented in 5,708 lines of Go. Because Iris absorbs significant developer overhead, our application requires 161 lines of code.

Iris achieves ZLT at 100 Gbps with 10 cores. Though our

	PSS	DPDK Mempool
Measuring Sec.	3.9 GB	2.2 GB
OpenVPN	2.5 GB	1.0 GB
Video Res.	3.5 GB	1.5 GB
Combined	5.0 GB	3.5 GB

Table 2: Peak memory usage for each case study over an hour. We measure both memory required by connection tracking (proportional set size) and packet Mbufs (DPDK). Long-term connection tracking and reassembly dominate memory usage. When composing applications, Iris unifies shared data, leading to a sublinear increase in memory usage.

callback is relatively expensive (inference takes 150K cycles on average), it is also invoked infrequently (Iris performed inference on only ≈300 intervals per second, on average, in our one-hour experiment). Additionally, Iris parses and reassembles at most an initial TLS handshake or DNS transaction.

Combining Applications. Finally, we evaluate Iris's ability to handle multiple, diverse subscriptions by running all four use cases simultaneously in a single Iris application. We emphasize that these applications have non-overlapping filters, diverse user-defined functions (e.g., applying a fingerprint vs. applying ML models), and place different demands on the framework (e.g., parsing vs. streaming). However, Iris is still able to share significant work across subscriptions. This application achieves ZLT at 100 Gbps using 12 cores.

6.4 Memory Usage

To further evaluate Iris's ability to execute long-running experiments, we estimate peak memory utilization for each case study application (§6.3) with 12 cores (Table 2). We sample *PSS* (proportional set size) from */proc/$PID/smaps_rollup* to capture memory used by Iris connection tracking and user-defined data structures [24]. We also query DPDK's memory pool utilization, which reflects the raw packets (`Mbufs` [23]) that Iris retains for zero-copy reassembly [63] and those that DPDK holds in memory while waiting for an available thread.

Across applications, connection tracking and reassembly dominate memory usage. Fingerprinting OpenVPN (case study #2) sees the lowest memory usage; it does not require reassembly, and it quickly discards long-lived connections after an initial prefix. Measuring Security Practices (case study #1) requires significant reassembly and application-layer parsing to identify applications. Estimating Video Resolution (case study #3) must accumulate and track all matched connections to termination. For the composition of all applications ("combined"), higher memory usage reflects the non-overlapping components of subscriptions. We note that, as each application progresses, memory utilization increases as the connection table grows in size and plateaus as connections terminate and time out (this matches observations from Retina [63]).

6.5 Compilation Time

One drawback to code generation is compilation time. In Iris, we find that compilation time increases with the number and complexity of subscriptions. However, Rust release optimizations—not Iris-specific logic—account for the vast majority of compilation time. Debug builds typically complete within a few seconds, allowing for rapid development.

First, we compile the combined (i.e., all replicated use cases in a single application) example from §6.3. With caching (libraries pre-built), a release build takes 30 seconds and a debug build takes 2.5 seconds. Of this, Iris's compile-time optimizations take less than 50 milliseconds. Second, as stress tests, we compile the two multi-subscription benchmarks from §6.2. Compiling one subscription takes 30 seconds (release) and 5 seconds (debug), with 15 milliseconds for Iris logic. Compiling 5,000 HTTP subscriptions takes 2 minutes (release) and 28 seconds (debug), with 18 seconds for Iris compiler logic. Building action trees for the IPv4 subnet sharding application takes longer (2 minutes for Iris logic for 1,024 and 6 minutes for 2,048 subnets), because Iris must generate and optimize trees for all state transitions through connection termination.

7 Conclusion

In this paper, we introduced Iris, a framework for efficiently developing applications that passively analyze network connections and reassembled flows. Iris compiles user-defined filters, data types, and callbacks writtein in Rust into an optimized, end-to-end pipeline that aggressively shares computation and data across analysis tasks. We showed that Iris can be used to more easily and performantly implement three diverse use cases from recent research in security, censorship circumvention, and network operations at 100 Gbps+ with 12 cores and less than 250 lines of code. We release Iris under the Apache 2.0 license to enable future research.

Acknowledgments

We thank the reviewers and our shepherd, Jia Wang, for their feedback, as well as the researchers who spoke with us about Internet measurement, including Ariana Mirian, Eric Wustrow, Jackson Sippe, Zimo Chai, Roya Ensafi, Diwen Xue, Ethan Katz-Bassett, Shuyue Yu, Francesco Bronzino, Nadia Heninger, and Nick Feamster. We also thank Tom Barbette and members of the Stanford Systems Community (Keith Winstein, Phil Levis, Akshay Srivatsan, Matthew Sotoudeh, Katherine Mohr, Yuhan Deng, Benjamin Driscoll, Alexander Root, Kimberly Ruth, Rumaisa Habib, Agur Adams, Nurullah Demir, and Phillip Stephens) for their feedback. We thank the Stanford security and networking teams for facilitating access to campus traffic. This work was supported in part by a Sloan Research Fellowship and the National Science Foundation under Grant Number #2319080. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the funding organizations.

References

[1] Abuse.ch. Urlhaus. https://urlhaus.abuse.ch/api/, 2025.

[2] David Adrian, Karthikeyan Bhargavan, Zakir Durumeric, Pierrick Gaudry, Matthew Green, J Alex Halderman, Nadia Heninger, Drew Springall, Emmanuel Thomé, Luke Valenta, et al. Imperfect forward secrecy: How diffie-hellman fails in practice. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 5–17, 2015.

[3] F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, 1970.

[4] Mark Allman. Putting DNS in context. In *Proceedings of the ACM Internet Measurement Conference*, 2020.

[5] Johanna Amann, Oliver Gasser, Quirin Scheitle, Lexi Brent, Georg Carle, and Ralph Holz. Mission accomplished? HTTPS security after diginotar. In *Proceedings of the 2017 Internet Measurement Conference*, pages 325–340, 2017.

[6] Johanna Amann and Robin Sommer. Exploring Tor's activity through long-term passive TLS traffic measurement. In *Passive and Active Measurement: 17th International Conference (PAM 2016)*, pages 3–15. Springer, 2016.

[7] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the mirai botnet. In *26th USENIX security symposium (USENIX Security 17)*, pages 1093–1110, 2017.

[8] Sangwook Bae, Mincheol Son, Dongkwan Kim, CheolJun Park, Jiho Lee, Sooel Son, and Yongdae Kim. Watching the watchers: Practical video identification attack in LTE networks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1307–1324, 2022.

[9] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-speed software data plane via vectorized packet processing. In *IEEE Communications Magazine*, volume 56, pages 97–103. IEEE, 2018.

[10] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast userspace packet processing. In *2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 5–16. IEEE, 2015.

[11] Diogo Barradas, Nuno Santos, and Luís Rodrigues. Effective detection of multimedia protocol tunneling using machine learning. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 169–185, 2018.

[12] Kevin Borders, Jonathan Springer, and Matthew Burnside. Chimera: A declarative language for streaming network traffic analysis. In *21st USENIX security symposium (USENIX Security 12)*, pages 365–379, 2012.

[13] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Hyojoon Kim, Renata Teixeira, and Nick Feamster. Traffic refinery: Cost-aware data representation for machine learning on network traffic. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 5(3):1–24, 2021.

[14] Francesco Bronzino, Paul Schmitt, Sara Ayoubi, Guilherme Martins, Renata Teixeira, and Nick Feamster. Inferring streaming video quality from encrypted traffic: Practical models and deployment experience. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 3(3):1–25, 2019.

[15] Albert Choi, Mehdi Karamollahi, Carey Williamson, and Martin Arlitt. Zoom session quality: A network-level view. In *International Conference on Passive and Active Network Measurement*, pages 555–572. Springer, 2022.

[16] Baek-Young Choi and Supratik Bhattacharyya. On the accuracy and overhead of Cisco sampled Netflow. In *ACM Sigmetrics Workshop on Large-Scale Network Inference*, 2005.

[17] Louis F DeKoven, Audrey Randall, Ariana Mirian, Gautam Akiwate, Ansel Blume, Lawrence K Saul, Aaron Schulman, Geoffrey M Voelker, and Stefan Savage. Measuring security practices and how they impact security. In *Proceedings of the Internet Measurement Conference*, pages 36–49, 2019.

[18] Karel Driesen and Urs Hölzle. The direct cost of virtual function calls in C++. In *Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 306–323, 1996.

[19] Zakir Durumeric, Zane Ma, Drew Springall, Richard Barnes, Nick Sullivan, Elie Bursztein, Michael Bailey, J. Alex Halderman, and Vern Paxson. The security impact of HTTPS interception. In *Network and Distributed System Security Symposium*, 2017.

[20] Ksenia Ermoshina, Benjamin Loveluck, and Francesca Musiani. A market of black boxes: The political economy of internet surveillance and censorship in russia. *Journal of Information Technology & Politics*, 2022.

[21] Alireza Farshin, Tom Barbette, Amir Roozbeh, Gerald Q. Maguire Jr., and Dejan Kostić. PacketMill: Toward per-core 100-Gbps networking. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021.

[22] Open Information Security Foundation. Suricata. 2022.

[23] The Linux Foundation. Data plane development kit. 2022.

[24] Free Software Foundation. *proc_pid_smaps(5) - Linux manual page*, 2025.

[25] Sergey Frolov and Eric Wustrow. The use of TLS in censorship circumvention. In *Network and Distributed System Security Symposium*, 2019.

[26] Francesco Fusco and Luca Deri. High speed network traffic analysis with commodity multi-core systems. In *ACM Internet Measurement Conference*, 2010.

[27] Massimo Girondi, Marco Chiesa, and Tom Barbette. High-speed connection tracking in modern servers. In *IEEE Conference on High-Performance Switching and Routing*, 2021.

[28] Qian Gong, Wenji Wu, and Phil DeMar. Goldeneye: Stream-based network packet inspection using GPUs. In *IEEE Conference on Local Computer Networks*, 2018.

[29] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection. In *USENIX Security Symposium*, 2008.

[30] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. Sonata: Query-driven streaming network telemetry. In *ACM Special Interest Group on Data Communications*, 2018.

[31] Qinwen Hu, Muhammad Rizwan Asghar, and Nevil Brownlee. Evaluating network intrusion detection systems for high-speed networks. In *2017 27th International Telecommunication Networks and Applications Conference (ITNAC)*, pages 1–6. IEEE, 2017.

[32] Johann Hugon, Paul Schmitt, Anthony Busson, and Francesco Bronzino. Cruise control: Dynamic model selection for ml-based network traffic analysis. *arXiv preprint arXiv:2412.15146*, 2024.

[33] Albert Jonathan, Abhishek Chandra, and Jon Weissman. Multi-query optimization in wide-area streaming analytics. In *Proceedings of the ACM symposium on cloud computing*, pages 412–425, 2018.

[34] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206, 1973.

[35] Steve Klabnik, Carol Nichols, and Chris Krycho. Using trait objects to abstract over shared behavior. `https://doc.rust-lang.org/book/ch18-02-trait-objects.html`. Accessed: July 2025.

[36] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. Coming of age: A longitudinal study of tls deployment. In *aCM Internet Measurement Conference*, 2018.

[37] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[38] Yuliang Li, Rui Miao, Changhoon Kim, and Minlan Yu. FlowRadar: A better NetFlow for data centers. In *USENIX symposium on networked systems design and implementation*, 2016.

[39] Sharat Chandra Madanapalli, Hassan Habibi Gharakheili, and Vijay Sivaraman. Know thy lag: In-network game detection and latency measurement. In *International Conference on Passive and Active Network Measurement*, 2022.

[40] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fifield, Sarah McKune, Arn Rey, John Scott-Railton, Ron Deibert, and Vern Paxson. An analysis of China's great cannon. In *USENIX Workshop on Free and Open Communications on the Internet*, 2015.

[41] Steven McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Winter USENIX Conference*, 1993.

[42] John McHugh, Ron McLeod, and Vagishwari Nagaonkar. Passive network forensics: behavioural classification of network hosts based on connection patterns. *ACM SIGOPS Operating Systems Review*, 2008.

[43] Asya Mitseva and Andriy Panchenko. Stop, don't click here anymore: boosting website fingerprinting by considering sets of subpages. In *USENIX Security Symposium*, 2024.

[44] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. Language-directed hardware design for network performance monitoring. In *ACM Special Interest Group on Data Communications*, 2017.

[45] ntop. PF_RING. 2022.

[46] Vern Paxson. Bro: A system for detecting network intruders in real-time. In *Computer Networks*, 1999.

[47] Julien Piet, Dubem Nwoji, and Vern Paxson. Ggfast: Automating generation of flexible network traffic classifiers. In *ACM SIGCOMM Conference*, 2023.

[48] 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. Flowblaze: Stateful packet processing in hardware. In *USENIX Symposium on Networked Systems Design and Implementation*, 2019.

[49] Ram Sundara Raman, Leonid Evdokimov, Eric Wurstrow, J Alex Halderman, and Roya Ensafi. Investigating large scale https interception in kazakhstan. In *ACM Internet Measurement Conference*, 2020.

[50] Jingjing Ren, Daniel J Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. Information exposure from consumer iot devices: A multidimensional, network-informed measurement approach. In *ACM Internet Measurement Conference*, 2019.

[51] Martin Roesch. Snort - lightweight intrusion detection for networks. In *USENIX Systems Administration Conference*, 1999.

[52] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking lte on layer two. In *IEEE Symposium on Security and Privacy*, 2019.

[53] Samuel Schüppen, Dominik Teubert, Patrick Herrmann, and Ulrike Meyer. FANCI: Feature-based automated NXDomain classification and intelligence. In *USENIX Security Symposium*, 2018.

[54] John Scott-Railton, Elies Campo, Bill Marczak, Bahr Abdul Razzak, Siena Anstis, Gözde Böcü, Salvatore Solimano, and Ron Deibert. Catalangate: Extensive mercenary spyware operation against catalans using pegasus and candiru. *'Citizen Lab Report No. 155'*, 2022.

[55] Taveesh Sharma, Tarun Mangla, Arpit Gupta, Junchen Jiang, and Nick Feamster. Estimating webrtc video qoe metrics without using application headers. In *ACM Internet Measurement Conference*, 2023.

[56] Meng Shen, Kexin Ji, Zhenbo Gao, Qi Li, Liehuang Zhu, and Ke Xu. Subverting website fingerprinting defenses with robust traffic representation. In *32nd USENIX Security Symposium*, 2023.

[57] Howie Shia. Forensic methodology report: How to catch nso group's pegasus. *Amnesty International*, 2021.

[58] Vincent Stoffer, Aashish Sharma, and Jay Krous. 100G intrusion detection. Technical report, Lawrence Berkeley National Laboratory, 2015.

[59] Ram Sundara Raman, Louis-Henri Merino, Kevin Bock, Marwan Fayed, Dave Levin, Nick Sullivan, and Luke Valenta. Global, passive detection of connection tampering. In *Proceedings of the ACM SIGCOMM 2023 Conference*, pages 622–636, 2023.

[60] Alisha Ukani, Ariana Mirian, and Alex C. Snoeren. Locked-in during lock-down: Undergraduate life on the internet in a pandemic. In *ACM Internet Measurement Conference*, 2021.

[61] Thijs van Ede, Riccardo Bortolameotti, Andrea Continella, Jingjing Ren, Daniel J. Dubois, Martina Lindorfer, David Choffnes, Maarten van Steen, and Andreas Peter. FLOWPRINT: Semi-supervised mobile-app fingerprinting on encrypted network traffic. In *Network and Distributed System Security Symposium*, 2020.

[62] Ryan Wails, George Arnold Sullivan, Micah Sherr, and Rob Jansen. On precisely detecting censorship circumvention in real-world networks. In *Network and Distributed System Security*, 2024.

[63] Gerry Wan, Fengchen Gong, Tom Barbette, and Zakir Durumeric. Retina: Analyzing 100gbe traffic on commodity hardware. In *ACM SIGCOMM*, 2022.

[64] Yifan Wang, Minzhao Lyu, and Vijay Sivaraman. Characterizing user platforms for video streaming in broadband networks. In *ACM Internet Measurement Conference*, 2024.

[65] Wireshark. Displayfilters. 2021.

[66] Shinae Woo and KyoungSoo Park. Scalable TCP session monitoring with symmetric receive-side scaling. Technical report, Korea Advanced Institute of Science and Technology, 2012.

[67] Mingshi Wu, Jackson Sippe, Danesh Sivakumar, Jack Burg, Peter Anderson, Xiaokang Wang, Kevin Bock, Amir Houmansadr, Dave Levin, and Eric Wustrow. How the great firewall of china detects and blocks fully encrypted traffic. In *USENIX Security Symposium*, 2023.

[68] Xiaoban Wu, Yan Luo, Jeronimo Bezerra, and Liang-Min Wang. Ares: A scalable high-performance passive measurement tool using a multicore system. In *IEEE International Conference on Networking, Architecture and Storage*, 2019.

[69] Diwen Xue, Michalis Kallitsis, Amir Houmansadr, and Roya Ensafi. Fingerprinting obfuscated proxy traffic with encapsulated TLS handshakes. In *USENIX Security Symposium*, 2024.

[70] Diwen Xue, Reethika Ramesh, Arham Jain, Michalis Kallitsis, J. Alex Halderman, Jedidiah R. Crandall, and Roya Ensafi. OpenVPN is open to VPN fingerprinting. In *USENIX Security Symposium*, 2022.

[71] Rohan Yadav, Shiv Sundram, Wonchan Lee, Michael Garland, Michael Bauer, Alex Aiken, and Fredrik Kjolstad. Composing distributed computations through task and kernel fusion. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 182–197, 2025.

[72] Da Yu, Yibo Zhu, Behnaz Arzani, Rodrigo Fonseca, Tianrong Zhang, Karl Deng, and Lihua Yuan. dshark: A general, easy to program and scalable framework for analyzing in-network packet traces. In *USENIX Symposium on Networked Systems Design and Implementation*, 2019.

[73] Shuhao Zhang, Hoang Tam Vo, Daniel Dahlmeier, and Bingsheng He. Multi-query optimization for complex event processing in sap esp. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 1213–1224. IEEE, 2017.

[74] Tianzhu Zhang, Leonardo Linguaglossa, Massimo Gallo, Paolo Giaccone, and Dario Rossi. FlowMon-DPDK: Parsimonious per-flow software monitoring at line rate. In *Network Traffic Measurement and Analysis Conference*, 2018.

[75] Yuxuan Zhang and Sebastian Angel. Quilt: Resource-aware merging of serverless workflows. In *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, pages 907–927, 2025.

Appendix

A Ethics

Though we performed the majority of development and testing offline, we evaluated our system on University network traffic. The ethics of analyzing real-world traffic are not universally agreed-upon. We draw the below practices from ethics sections of other papers that passively analyze user network traffic. [5–7, 13, 14, 17, 29, 42, 60, 61, 63, 69].

1. **Ethics Review**. We received approval from our University's Privacy and Security office and consulted our Institutional Review Board (IRB), which deemed our work out-of-scope. Working with our network IT staff, we ensured all experiments fell within user terms of service.
2. **Data Processing**. We anonymize all IP addresses and do not have access to DHCP traffic (i.e., we cannot directly correlate IP addresses with device or user identity). We perform one experiment that tracks individual flows: our implementation of DeKoven et. al.'s work [17]. In this case, we delete all collected data at runtime (i.e., immediately after writing to disk). For all other evaluation, we collect only aggregated data.[5]
3. **Impact on Users**. Our server only receives a copy of traffic out-of-band to ensure no impact on network service.
4. **Operational Security**. We implement rigorous physical and network access controls to ensure we do not increase the attack surface for network users. We limit access to researchers directly involved with the project.

We open-source Iris with the intention of lowering barriers to security, anti-censorship, and QoE investigation on high-speed networks. However, as in much of the security field, we must ask whether Iris could be misused to expand network surveillance capabilities. We believe that organizations, nation-states, and attackers already monitor network traffic using in-house or commercial systems and large-scale infrastructure [2, 5, 19, 20, 40, 49, 54, 57, 59, 67]. Researchers, who operate with fewer resources, are much more limited. In addition to Iris's operational applications, we believe that network traffic is most secure when responsible researchers and organizations are able to surface risks and evaluate defenses in realistic settings. We hope that open-sourcing Iris can support such efforts.

B Interface Mechanics

As introduced in Section 4, an Iris subscription consists of a filter, datatype(s), and callback (conceptually: traffic of interest, format(s) that traffic data should be delivered in, and computation to perform on that data). Iris's programming model is built on connections; subscriptions are scoped to a single five-tuple-defined TCP or UDP connection, though users can aggregate data following a callback. We describe how to use each component of a subscription in more detail below.

[5]In some cases, Iris's high throughput allows us to write less data to disk than other systems. For example, Xue et. al. [70], restricted by Zeek's throughput and DSL, wrote raw connection data to disk for offline analysis. We perform fingerprinting online.

B.1 General Properties

Though user-defined data types, filter predicates, and callbacks have different semantics, they share certain properties. Each exposes one or more APIs for Iris to invoke at various points in a connection's lifetime, i.e., at the states and state transitions in Table 1, and each may optionally attach state to connection processing. We briefly describe shared properties.

Each of data types, filter predicates, and callbacks can be stateful or stateless over a connection. Stateful constructs are Rust structs, each of which must expose a constructor and one or more associated methods for Iris to invoke. The constructor accepts the first packet in a connection, and associated methods must take a mutable `self` parameter. To implement a destructor, developers customize the Rust `Drop` trait. Structs can also implement arbitrary helper methods; Iris only invokes APIs annotated with a macro.

All user-defined APIs hook into connection state machines, either explicitly as a macro argument (e.g., `update` in Listing 3) or implicitly through arguments (e.g., `dns_record` in Listing 2). In general, Iris infers when it should invoke each API based on function parameters. If parameters are insufficiently specific, compilation will fail. In these cases, developers must specify explicit state transitions as macro parameters. For example, the *ResolutionEst* callback in Listing 3 specifies that Iris should invoke its *end* method at connection termination.

Annotations on methods associated with a struct must provide the struct name (e.g., "ResolutionEst" in Listing 3). This is a syntax quirk of Rust macros, which cannot access state beyond their immediate inputs.

B.2 Iris Data Types

An Iris data type is a Rust struct built from one or more primitive data types. Primitive data types are the Iris state transitions, including self-transitions, along with associated data, as listed in Table 1. At a high level, developers construct data types over raw connection frames, reassembled streams, or parsed connection- and session-level constructs directly exposed by Iris.

Iris supports two kinds of data types: stateful and stateless. Stateless data types define only a constructor, which is marked `#[datatype]`, accepts a primitive data type, and returns `Self`. For example, we construct an anonymized five-tuple on the first packet of a connection in Listing 4.

Stateful data types—such as the *FeatureChunk* data type presented in Section 4.2—transform and accumulate data throughout the lifetime of a connection. Stateful data types are Rust structs (Appendix B.1) tagged with the `#[datatype]` macro, with a constructor and one or more `#[datatype_fn]`

```
impl AnonFiveTuple {
    #[datatype("AnonFiveTuple,L4FirstPacket")]
    fn new(first_pkt: &L4Pdu) -> Self {
        let mut five_tuple = FiveTuple::from_ctxt(first_pkt.ctxt.clone());
        five_tuple.orig.set_ip(Self::anon_ip(&five_tuple.orig.ip()));
        Self { data: five_tuple }
    }
}
```

Listing 4: Example data type that does not accumulate state over a connection.

associated methods, each of which accepts a mutable "self" parameter and a primitive data type.

After defining a data type, developers can filter on and request it in a callback. If multiple filters and callbacks require the same data type, Iris will construct it at most once.

B.3 Iris Filters

Every subscription has an associated filter, expressed as an annotation on a callback. A filter contains one or more patterns (conjunctive predicates). Iris allows users to specify additional patterns in an input file when doing so as a macro parameter would be cumbersome; for example, we leverage this in Section 4.2 to define a list of TLS SNIs associated with known video platforms.

Each predicate can refer to protocols and protocol fields or a user-defined filter predicate. Iris supports Retina's Wireshark-like filter language [63]. Inspired by Snort [51] and other intrusion detection use cases, we extend this language with a "contains" operator and raw byte matching.

User-defined filters are either (1) functions tagged as `#[filter]` or (2) structs tagged `#[filter]` with one more more associated `#[filter_fn]` methods, each of which returns a `FilterResult`. Filters take in any Iris datatype(s) (primitive or user-defined) and return a `FilterResult`: `Accept` (connection matches the filter predicate), `Drop` (connection does not match), or `Continue` ("maybe", requires more information). Custom filter predicates can use `Continue` to stream data over multiple windows within a connection before making a decision.

Section 4.2 presents the *video_lookup* filter, which is stateless over a connection (though it does access global state). Filters can also accumulate state over a connection. For example, we define a filter ensures that only "long-lived" connections are delivered to an associated callback in Listing 5.

B.4 Iris Callbacks

Iris callbacks define analysis logic to perform on constructed data types for connections that match a traffic filter. If a developer does not explicitly specify a state transition in a macro argument, the runtime invokes each callback as soon as it constructs all data types and matches a filter. We note that de-

```
#[filter]
struct IsLongLived {
    bytes: usize,
    start: Instant,
}

impl IsLongLived {
    // Constructor omitted for brevity
    #[filter_fn("ShortConnLen,L4Conn")]
    fn update(&mut self, pdu: &L4Pdu) -> FilterResult {
        self.bytes += pdu.payload().len();
        let dur = Instant::now() - self.start;
        if self.bytes > BTS || dur > DUR {
            return FilterResult::Accept;
        }
        FilterResult::Continue
    }
}
```

Listing 5: An example filter that accumulates state over a connection.

velopers must attend to compatibility; for example, a callback function that requests invocation on "FirstPacket" along with a "TLS handshake" data type will never be invoked. If Iris detects incompatibility, compilation fails.

Each Iris callback is a Rust function or data structure tagged with the `#[callback]` macro. This macro takes an associated traffic filter, which can refer to supported protocols, protocol fields, user-defined predicates, and/or input files. For stateful callbacks (i.e., Rust structs), developers specify associated methods with `#[callback_fn]`. Listing 3 shows a stateful callback (a Rust struct with associated methods), while Listing 2 shows a stateless callback (a function for analyzing DNS transactions).

Callbacks that stream data over a connection can optionally return a boolean value: `true` to continue receiving data for the connection and `false` to "unsubscribe" (see the *update* method in Listing 3). If a callback returns "false" for a connection, Iris will no longer invoke its methods for that connection; this is akin to "filtering out".