Columnar Packet Traces for Scalable Encrypted-Internet Measurement

Anonymous Author(s)
Submitted for double-blind review

***Abstract*—Internet traffic volumes continue to grow while transport and application encryption increasingly limit payload visibility. However, modern transport protocols such as QUIC complicate traditional packet-trace analytics built around `pcap` (e.g., tshark-to-CSV), and row-oriented extraction increases the cost of repeated analysis. This paper makes repeated offline measurement practical by converting packet traces into a column-oriented representation that separates one-time parsing from downstream analytics. Concretely, we introduce the architecture of a software pipeline that converts `pcap` files to the column-oriented Apache `parquet` format. To the best of our knowledge, this is the first end-to-end pipeline that (i) converts `pcap`/`pcapng` traces into `parquet` with a schema aimed at repeated offline measurement and (ii) is engineered around a convert-once, query-many workflow with measured end-to-end time-to-insight benefits. The pipeline leverages parallel parsing to produce a columnar dataset that can be queried efficiently by standard analytics engines. We benchmark end-to-end time-to-insight and storage costs across representative workloads and diverse datasets. Our results show that after a one-time conversion, subsequent analyses can exploit column pruning and vectorized scans to reduce query time compared with re-parsing `pcap` or re-extracting row-oriented text formats, while maintaining storage costs competitive with compressed `pcap` and CSV baselines.**

***Index Terms*—Packet trace analysis, columnar storage, Apache `parquet`, encrypted traffic, network measurement**

I. Introduction

Packet traces are records of packets collected at a network measurement point, typically stored in standard formats like `pcap` or `pcapng`. Each file contains a sequence of records that include timestamps and byte counts, covering link headers, network and transport headers, and, when available, part or all of the payload [1], [2]. Packet traces can be captured for various purposes, such as diagnosing incidents, monitoring traffic on router interfaces to assess user experience, or gathering data on monitoring ports for security. In research, organizations such as the Center for Applied Internet Data Analysis (CAIDA) provide backbone link traces for traffic analysis and research [3]. On the other hand, Measurement Lab captures network measurements, including packet data from speed tests, to analyze performance from the user's perspective [4].

A packet trace is a descriptive data set that does not directly indicate flows or application outcomes. Instead, these insights are derived from offline processing that reconstructs flows and computes statistics based on packet timing and sizes. Packet traces can be high-volume, often containing millions to billions of records, and are heterogeneous due to varying packet sizes, protocols, and header options over time. Captures may be truncated with a capture length limit, known as snaplen, to manage storage and privacy. Additionally, packets may be dropped during overload, and timestamp precision can differ across platforms. These factors affect the reliability of inferences and the cost of repeated processing.

Packet traces are key for understanding network behavior, despite some limitations. Concretely, they support studies that require packet-level insight, such as identifying retransmission patterns and verifying handshake behavior. Packet traces also play a crucial role in security audits, where analysts search for indicators like scanning behavior or protocol misuse. In practical environments, packet capture tools such as `tcpdump` built on `libpcap` and analysis tools such as Wireshark are routinely used for these purposes [5]–[7]. In academic contexts, packet traces serve as a foundation for characterizing network traffic and validating protocol behavior [3], [4]. As trace collections grow, efficient offline processing faces challenges because the same dataset is often analyzed repeatedly to answer multiple questions, to refine hypotheses, or to reproduce results.

The first challenge is scale. Public reports and forecasts show that global total internet traffic volumes have reached hundreds of exabytes per month, and growth continues as video, cloud services, and mobile access expand [8], [9]. This growth is reflected in the size of packet trace collections produced by operators and measurement platforms. The scale trend, therefore, motivates data representations and processing pipelines that reduce redundant work and improve time-to-insight without requiring specialized infrastructure. Encryption also poses a significant challenge in trace analysis workflows by limiting the effectiveness of payload inspection. More precisely, it reduces the usefulness of payload inspection for most applications and shifts analyses toward signals that remain observable without decryption, such as timestamps, packet sizes, addressing information, and selected header fields present in the capture. This shift matters for passive analysis because QUIC moves key transport behavior behind encryption and replaces several TCP-visible signals with alternative mechanisms, limiting the applicability of tools based on TCP [10]–[13]. As a result, offline processing pipelines must be effective when analysis is constrained to packet headers and limited metadata. Together, growing trace volumes and reduced payload visibility increase the cost of repeated parsing and make it harder to iterate quickly using traditional `pcap`-based workflows.

In this paper, we explore a different design perspective: representing packet traces as a *columnar dataset*. This dataset

focuses on network and transport header fields, along with limited metadata, and is designed to support common measurement tasks while retaining minimal information. In other words, the columnar trace representation enables "convert once, query many" analytics with materially lower time-to-insight, while maintaining competitive storage efficiency and predictable measurement fidelity under encryption and snaplen constraints.

We develop a software tool that converts packet traces in `pcap` format to Apache `parquet` files. To the best of our knowledge, we developed the initial implementation designed for efficient repeated trace analytics using a column-oriented representation. The tool leverages parallel processing to accelerate conversion processes and uses a typed schema that emphasizes network and transport header fields. We benchmark end-to-end time-to-insight and storage costs across representative analysis workloads and multiple trace datasets. The results show that a single conversion step enables subsequent analyses to read only the required fields and process them efficiently using column-oriented scans, reducing query latency relative to repeated parsing of the original `pcap` file or field extraction into row-oriented text formats. At the same time, the resulting `parquet` datasets remain space-efficient, with storage overhead competitive with commonly used compressed baselines such as compressed `pcap` (`pcap.gz`) and CSV outputs.

A. Contributions

This paper makes three contributions:

- **A column-oriented packet trace representation for encrypted traffic.** We define a structured format for packet traces that minimizes information loss. It splits required fields from optional metadata. This format supports the extraction of observable fields that enable QUIC traffic classification when available.
- **An efficient `pcap-to-parquet` conversion pipeline.** We design and implement an open-source pipeline that converts `pcap` traces into datasets with different output formats (CSV, `json`, and `parquet`). The pipeline parses packet headers and writes typed columns with configurable row-group sizing and compression. It also supports both single-process and multi-process modes, partitioning input traces and processing chunks in parallel to reduce latency on multi-core machines. The implementation is compatible with existing analytics systems and does not need specialized storage or databases
- **A representative workload suite with end-to-end benchmarking methodology.** We evaluate complete pipelines end-to-end, capturing conversion time, query latency, and output size rather than focusing on isolated benchmarks. We also report various metrics, including time-to-insight across repeated queries and the effect of selective column reads.

B. Paper organization

The remainder of this paper is organized as follows. Section II briefly outlines the related work and Section III introduces the system architecture. Section IV describes the datasets used to assess the proposed tool and Section V shows the system performance results. Finally, Section VI summarizes the conclusions.

II. Related Work

This work relates to prior efforts in four directions: interactive packet-trace tooling, log-based measurement pipelines, large-scale processing, and column-oriented analytics formats.

Packet capture is commonly performed using `tcpdump` and `libpcap`, and traces are inspected using protocol dissectors such as Wireshark and `tshark` [5]–[7], [14]. These tools expose detailed packet fields, provide extensive protocol support, and enable inspection when the payload is available. The primary limitation of these tools is that they are optimized for interactive inspection and single-pass extraction, rather than for repeated analytics across large repositories. This disadvantage is exacerbated by encryption, which reduces visibility into the payload and diminishes the marginal value of full-fidelity packet dissection for many applications. Our approach reduces redundant parsing by converting packet traces into a structured dataset, enabling repeated queries that read only the necessary columns while keeping the original `pcap` available for exploring when needed.

Another common approach is to transform packets into structured logs. Systems such as Bro/Zeek parse traffic and emit event logs and connection summaries that are easier to store, index, and query than raw packets, and they are widely used for security monitoring and operational visibility [15], [16]. The key advantages of log-based pipelines are reduced data volume, clear semantics, and straightforward integration with search and analytics systems. The main limitation is that the extracted representation is inevitably lossy and reflects design-time choices about which fields and aggregations matter. Our approach preserves substantially more temporal and per-packet detail than flow logs while still enabling analytics-style queries.

In the distributed setting, early work proposed Hadoop-based tools for trace analysis, including custom input formats that enable MapReduce jobs to efficiently read packet records from `pcap` data stored in distributed file systems [17]. More recent systems use Spark-based pipelines to process traffic at scale, often focusing on flow extraction and feature computation for downstream analytics [18]. These systems provide clear advantages for archival search and computation, but they also have practical limitations for many users: they may require specialized infrastructure, they often retain `pcap` as the primary storage format. Our approach focuses on a standard analytics format (`parquet`) that can be queried efficiently on a single machine and processed in distributed engines when needed.

Column-oriented data management has long been used to speed analytical workloads by enabling selective access, high compression, and vectorized execution. `parquet` and `arrow` have since become widely used building blocks for analytic pipelines and engines [19], [20]. These formats provide powerful primitives, but they do not solve the

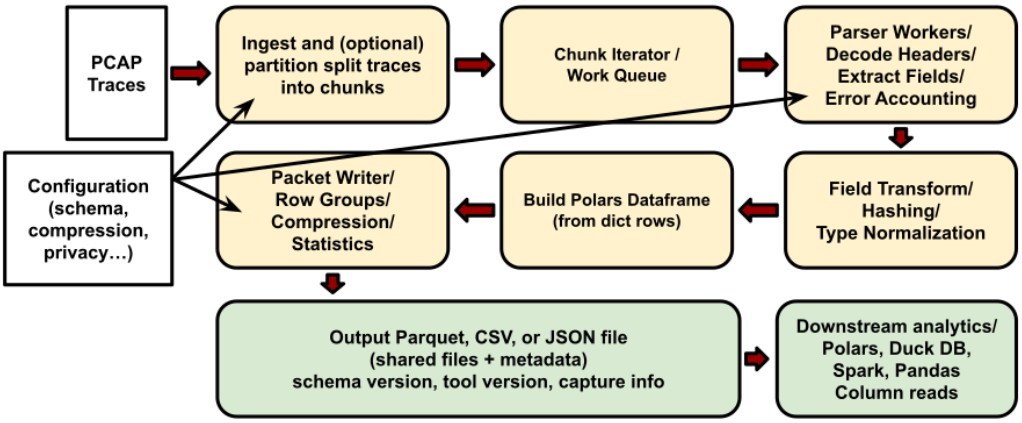

Fig. 1. Architecture of proposed software tool. The pipeline converts `pcap`/`pcapng` traces into standard format datasets using optional trace partitioning and parallel workers. Configuration controls schema selection, compression, row-group sizing, and privacy transformations. The output dataset is compatible with standard analytics engines that support selective column reads and filter pushdown.

domain-specific problem of mapping packet traces into a stable schema while preserving the information required for measurement tasks under snaplen and encryption constraints. Our contribution builds on these foundations by (i) defining a schema centered on header-derived fields that remain useful under encryption, (ii) providing a conversion pipeline with a multi-process option and a bottleneck characterization, and (iii) evaluating end-to-end time to insight, storage cost, and the amortization behavior when multiple queries are issued over the same traces.

III. System Architecture

This section describes the architecture of our software tool, which transforms packet traces stored in datasets in the `pcap` and `pcapng` formats into different standard formats (`parquet`, `json`, CSV) designed for repeated offline data analysis. The primary objective is to decouple the expensive, low-level work of packet parsing and field extraction from downstream analytics. By using a column-oriented approach, the pipeline facilitates further analyses by providing access only to the required fields, thereby avoiding the need to repeatedly scan and decode the original trace files.

Fig. 1 shows the architecture of the proposed tool. The system begins by opening the input capture and handling compressed files. It determines whether the input file is in the `pcap` or `pcapng` format. Based on a predefined decision rule, the system selects between sequential and parallel processing: files smaller than a specified threshold are processed sequentially, while larger files are processed in parallel. In the parallel mode, each processor decodes the packets, extracts the required fields, and creates a partial dataframe, using the `polars` open-source library for data manipulation. To ensure efficient data transfer, the workers serialize their partial dataframes into an in-memory buffer. Once all chunks are complete, the main process merges the outputs generated by the parallel workers into a single table. Since the chunks may finish in a different order, the implementation sorts the partial results by their initial timestamp before concatenating them. After this, the pipeline renumbers the packets to reestablish a consistent sequential index. The merged dataframe then undergoes any configured post-processing callbacks (if applicable) before being exported to the requested output format. When the output format is set to `parquet`, the exporter creates a single analytics-ready file in a specified output directory using the predefined encoding options. This allows downstream tools to scan only the necessary columns during analysis. The rest of this section explains the role and operation of each module shown in Fig. 1.

A. Inputs and configuration

The proposed design accepts one or more packet capture files in `pcap` or `pcapng` format. In practice, the tool also supports compressed captures on disk (e.g., `.gz`, `.bz2`, `.xz`) by transparently opening them via an optional routine before decoding. After opening the file, the implementation detects whether the capture is `pcap` or `pcapng` and derives basic link-layer context to determine which header decoders are applicable. The configuration module controls (i) which protocol decoders are enabled, (ii) which fields are extracted and post-processed, (iii) optional privacy transformations applied to selected columns, (iv) whether parallel processing is enabled, and (v) the output format and destination. The output format can be `parquet`, CSV, or `json`; when `parquet` is selected, encoding and compression are handled by the underlying `polars` writer. In multi-process mode, the configuration is propagated to all worker processes, ensuring that each chunk is decoded with the same decoder set and field definitions. This ensures that partial dataframes produced by workers have consistent column names and data types and can be merged deterministically by the main process before export.

B. Ingest and optional partitioning

The ingest stage is responsible for opening an input capture and presenting it as a well-defined sequence of packet records to the rest of the pipeline. The released implementation

accepts `pcap` and `pcapng` files and can transparently open captures that are compressed on disk (e.g., `.gz`/`.bz2`/`.xz`) before decoding. After opening the file, the ingest logic detects the capture format and iterates over packet records in timestamp order. For each record, ingest makes available the capture timestamp, the captured length (`caplen`), the original wire length (`len`), and the captured bytes. In practice, this stage uses standard capture readers that iterate over `pcap` records in file order while preserving record boundaries. This preservation is crucial for the accuracy of later stages, which need to decode headers and interpret length fields correctly.

Our implementation supports two execution paths. In single-process mode, ingest streams packet records directly to the decoder and extractor, producing a single sequential pass over the input file. In multi-process mode, ingest performs an additional partitioning step that enables parallel decoding on multi-core machines. This partitioning follows the design: the system first scans the capture to identify safe split points at packet-record boundaries, then groups records into chunks, and finally materializes each chunk as an in-memory byte buffer that can be processed independently by a worker process. Each worker replays its buffer through the same capture reader and decoding logic as the sequential path. Because worker processes may complete out of order, later stages merge partial results deterministically (e.g., by ordering chunks according to their first timestamp) before export.

Partitioning aims to reduce conversion time on multi-core machines while also limiting memory usage. Let $T_{\mathrm{conv}}(1)$ represent the end-to-end conversion time in single-process mode, and let $T_{\mathrm{conv}}(p)$ denote the time taken when using p worker processes. A simplified decomposition can clearly illustrate the purpose of partitioning:

$$T_{\mathrm{conv}}(p) \approx T_{\mathrm{scan}} + T_{\mathrm{split}} + \frac{T_{\mathrm{parse+extract}}}{p} + T_{\mathrm{write}} + T_{\mathrm{over}}(p), \tag{1}$$

where T_{scan} and T_{split} capture the ingest and split-point identification costs, $T_{\mathrm{parse+extract}}$ captures the packet decoding and field extraction work that parallelizes across chunks, T_{write} captures output encoding and I/O, and $T_{\mathrm{over}}(p)$ captures parallelization overheads (process startup, queueing, and coordination). Equation (1) highlights two practical effects observed in our evaluation. First, the benefits of multi-process execution arise when the parallelizable term outweighs the fixed costs, which happens with sufficiently large traces. Second, chunking naturally limits memory usage, as each processor can handle only a limited number of packets, storing them in column buffers and flushing them to the writer incrementally, rather than buffering an entire trace in memory at once. Finally, partitioning does not affect the logical output: for a fixed schema and configuration, both single-process and multi-process modes generate identical datasets. The only difference lies in how the input trace is assigned to workers.

C. Chunk iterator and work scheduling

This module connects data ingestion and partitioning with parallel parsing. Its main responsibilities are to present work units to the processing stage in a manner that (i) balances the workload among workers, (ii) ensures deterministic execution for a fixed configuration, and (iii) allows for tracking progress and errors. A *chunk descriptor* is a fundamental unit of work. Each descriptor identifies an input capture file and a contiguous range of packet records within that file, as produced by the partitioning stage (Section III-B). In multi-process mode, the scheduler manages a work queue of chunk descriptors and assigns them to p worker processes. The implementation follows a standard producer-consumer design: the main process enqueues chunk descriptors while the workers dequeue them and process the chunks independently. The system concludes when all descriptors have been processed. This design ensures stable throughput as p increases.

Since the processing time for chunks can vary based on the mix of packets and the complexity of headers, the scheduler employs dynamic assignment rather than static partitioning among workers. This means that workers request new chunks as soon as they complete their current tasks. This approach helps minimize tail effects, in which one worker might be assigned a particularly challenging chunk and thereby become a bottleneck for the entire process. In single-process mode, the scheduler behaves as a simple iterator that yields a single chunk representing the complete input trace. If the user requests chunking for memory management, the iterator can yield a sequence of chunks even without parallel processing. This approach maintains a consistent code path for downstream modules, ensuring that both parsers and writers always operate on a stream of chunk descriptors, regardless of whether the execution is parallel or sequential.

D. Packet decoding and field extraction

The packet-decoding and field-extraction stage converts each capture record into a row of typed fields according to the output schema. The implementation follows a two-level design. A lightweight per-packet decoder parses common link, network, and transport headers and emits a dictionary of extracted values. These per-packet dictionaries are then assembled into a `polars` dataframe for export. The same decoding logic is used in both sequential and parallel execution; parallelism only changes how packets are fed to the decoder. For each capture packet record, the decoder first reads the capture timestamp, the captured bytes `caplen`, and the original wire length `len`. It then parses protocol headers from the captured bytes in layers. At the link layer, the decoder identifies the encapsulation and, when present, accounts for common tags (e.g., VLAN) to locate the network-layer header. At the network layer, it parses IPv4 or IPv6 and extracts source and destination addresses, the next-header identifier, and basic header properties. At the transport layer, it parses TCP/UDP/ICMP and extracts fields required by most offline measurement tasks, including ports, protocol identifiers, packet sizes, and transport signals such as TCP flags. The implementation uses a lightweight packet-parsing library (`dpkt`) for these standard headers to keep per-packet overhead low while maintaining broad format coverage [21].

In addition to the core headers, the implementation can extract application-level data to help characterize encrypted

traffic. A notable example is Server Name Indication (SNI), which is commonly used for service identification and traffic classification. For TLS over TCP, the SNI is included in the TLS ClientHello as the `server_name` extension. It can be observed by parsing the handshake bytes without access to session keys, unless Encrypted ClientHello is used [22]–[24]. In other words, to extract the SNI for TLS, one generally only needs to parse the unencrypted ClientHello record.

In QUIC, the situation differs from other protocols. QUIC operates over UDP and incorporates TLS 1.3, with the TLS handshake contained within QUIC CRYPTO frames and secured during transmission [10]–[12]. To obtain the server name in QUIC, one typically needs to decrypt the QUIC initial packets. The decoder identifies the QUIC header initial packets, removes the header protection, and decrypts the initial payload to access the CRYPTO frames that include the TLS ClientHello [11]. The implementation includes this initial decryption path and attempts to extract the server name when the trace contains enough bytes to cover the relevant initial payload and when the handshake fragments needed to reconstruct the ClientHello are present. When the capture is truncated or when the ClientHello does not expose SNI (e.g., due to ECH), the system emits a null value rather than guessing.

Robustness is a crucial design requirement because production traces often include malformed packets, uncommon encapsulations, partial captures, and mixed-protocol traffic. Each worker, therefore, performs explicit error accounting. When a packet cannot be decoded, if an expected header is truncated by the snaplen, or if a required field in the schema is missing, the worker records structured error indicators—such as a parse error flag and an error code—and continues processing subsequent packets. This approach enables downstream analyses to quantify both coverage and fidelity. In particular, it allows evaluation to report the percentage of packets that contain the required fields and to differentiate between true absence (for example, non-TCP traffic) and missing data due to capture truncation or decoding failures.

E. Field transforms and normalization

Following header decoding and field extraction, our pipeline includes an optional transformation stage that aims to achieve two goals: privacy protection and type stability.

Privacy Transforms. The pipeline supports policy-driven transformations of identifiers that may be sensitive or unnecessary for a given analysis. Typical examples include source and destination IP addresses, hostnames, and limited application hints when enabled. Depending on the configuration, the transform module can remove these fields entirely or replace them with privacy-preserving surrogates such as keyed hashes. Hashing preserves joinability within a dataset (and, if a stable key is reused, across datasets) while reducing direct identifiability; however, it does not eliminate re-identification risk, especially when auxiliary information is available. For analyses that require structural properties of the address space (e.g., prefix aggregation), prefix-preserving anonymization is a standard option and can be implemented. Because privacy requirements vary across deployments, our tool treats privacy transforms as configurable and documents them in the output metadata so downstream users can interpret results correctly.

Type Normalization. Packet headers contain fields that vary in width and encoding, and traces often include multiple versions of protocols. To ensure that the `parquet` dataset is stable and ready for analysis, the transform module maps extracted values into a consistent type system before assembling the columns. Integers are represented with fixed widths (for example, 32-bit ports and 64-bit timestamps), addresses are displayed in stable formats (such as fixed-length binary for IPv6 and canonical binary form for IPv4), and enumerated fields (like protocol identifiers) can be represented as small integers with optional dictionary encoding. This normalization ensures that different pipeline runs produce compatible datasets, which is essential for reproducibility and for executing the same queries across multiple trace collections.

In addition, the transform module standardizes the representation of missing or inapplicable values. When a field is not defined for a protocol (e.g., TCP flags for UDP packets) or is unavailable due to capture truncation (`caplen`) or decode failures, the module emits explicit nulls. This choice allows downstream queries to distinguish between "not applicable" and "not observed" cases.

F. Column builder and batching

The column builder is the bridge between packet-level extraction and `parquet` output. Its role is to convert the stream of per-packet fields produced by the decoder and transform stages into a sequence of typed, column-oriented batches suitable for efficient encoding and downstream analytics.

1) Column-Oriented Materialization: Instead of representing each packet as a row-wise object, such as a Python dictionary, the builder maintains an append-only buffer for each schema field. For every processed packet, the value of that field is appended to its corresponding buffer. In our implementation, the builder materializes batches using Apache Arrow data structures (e.g., `pyarrow` Arrays and RecordBatches), which provide a standard in-memory columnar format and a direct path to `parquet` encoding [19], [20].

2) Batching and Memory Bounds: The builder generates a batch once a specified target batch size is reached. This batch size can be defined as either a maximum number of packets per batch or as an approximate memory budget. If B represents the number of packets in a batch and $\bar{s}$ denotes the average number of bytes of extracted data per packet according to the schema, the builder's working set size scales approximately as $O(B \cdot \bar{s})$, plus a small constant overhead for array metadata. By imposing a limit on B, the system can control peak memory usage, even when handling very large traces.

3) Row groups and downstream performance: Batch boundaries also interact with the `parquet` layout. When writing, the writer groups one or more batches into row groups, which serve as the unit for encoding and statistics. Larger row groups can enhance compression and minimize

metadata overhead, while smaller row groups can facilitate selective reads and filter pushdown for highly selective queries. For this reason, the builder offers batch sizing as a configurable parameter and records the selected settings in the output metadata. This ensures that performance results can be reproduced and understood accurately.

G. `parquet` Writer and Dataset Assembly

The writer stage transforms column batches into a dataset. It involves (i) efficiently encoding columns, (ii) organizing records into row groups that balance compression and query performance, and (iii) generating sufficient metadata to make the dataset self-describing. Our implementation uses the standard `parquet` toolchain (e.g., `pyarrow.parquet`) to ensure compatibility with common analytics engines [19], [20]. The `parquet` format stores data in row groups, each of which contains contiguous blocks for every column. Row groups are the primary units for encoding, compression, and statistics, and they strongly influence downstream scan performance. The writer, therefore, exposes row-group size as a configuration parameter and constructs row groups by packing one or more column batches generated by the builder. Larger row groups typically improve compression efficiency and reduce metadata overhead, while smaller row groups may enhance performance for selective queries by minimizing the amount of data that needs to be decompressed when filters are applied. In addition, the writer selects the configured compression codec (e.g., Snappy or Zstd) and applies it to each column, leveraging `parquet`'s ability to compress each field based on its value distribution.

The output is written as a dataset (a directory of files) rather than as a single monolithic file. In multi-process mode, each worker writes one or more files using the same schema and encoding settings. This process enables near-linear output parallelism when storage bandwidth permits. In single-process mode, writing a dataset remains beneficial for manageability (incremental flush, bounded memory) and for compatibility with distributed analytics systems. To support reproducibility and schema evolution, the writer records metadata at two levels. File-level metadata includes the schema definition (column names and types) and encoding parameters. Dataset-level metadata records the schema version, tool version, and capture context, including input file identifiers, capture snaplen, and the configuration used for privacy transformations. Recording this information enables users to interpret nulls correctly (e.g., field not present vs. truncated) and to re-run analyses consistently as the schema evolves.

H. Downstream query path

The output is a dataset designed for repeated analysis using standard engines that can scan column-oriented files. Many offline measurement tasks require only a small set of fields, such as timestamps, packet lengths, and protocol identifiers. Column-oriented execution allows queries to read only the necessary columns and, when supported by the engine, to apply filters during the scan. This helps skip irrelevant row groups or files, reducing the amount of decompression work.

TABLE I
SUMMARY OF DATASETS USED IN EVALUATION. "SNAPLEN" DENOTES THE CAPTURE LENGTH LIMIT WHEN KNOWN.

Dataset	#Files	Size range	Total size	Snaplen
Benchmark (Network)	10	10 kB–300 MB	∼445 MB	none
Benchmark (Client)	9	10 kB–100 MB	∼145 MB	none
M-Lab NDT7	1340	up to ∼100 MB	4.4 GB (688 MB tgz)	94 B
CAIDA	5	up to ∼1.9 GB	∼9 GB	40 B

This approach supports an iterative workflow in which analysts can refine their questions and perform multiple analyses on the same trace collection without needing to repeatedly parse the original `pcap` file. The original captures can still be retained for occasional deep inspection and protocol debugging, while routine characterization and performance measurements are conducted on the structured dataset.

IV. DATASETS

We evaluate the proposed software tool using three dataset families collected under different operating conditions: (i) a controlled-size ladder used for benchmarking conversion time and output size, (ii) a collection from Measurement Lab that reflects large-scale, real-world access-network measurements, and (iii) backbone-style traces from CAIDA that stress short capture lengths and higher traffic diversity. Table I displays a summary of the datasets used to evaluate the proposed tool.

A. Benchmark dataset (controlled size ladder)

The benchmark dataset comprises two collections of `pcap` files used to study scaling effects and identify breakpoints at which overheads are amortized. These collections are referred to as *Network traces* and *Client traces*. It is worth noting that these collections were captured using synthetic datasets on a real network. The *Network traces* collection contains 10 network-side captures, ranging in size from 10 kB to 300 MB. On the other side, the *Client traces* collection contains 9 client-side captures spanning 10 kB to 100 MB, with the same intermediate steps up to 30 MB. These traces are not truncated by a capture-length limit (i.e., no snaplen restriction is imposed), isolating the performance impact of file size and workload structure from the effects of truncated headers.

This dataset is dominated by longer transfers between a single client and server, which reduces variability in some header-derived fields compared with backbone traffic. This property is useful for controlled benchmarking because it mitigates harmful effects from highly diverse packet mixes and emphasizes the cost of repeated parsing.

B. Measurement Lab dataset (NDT7 packet headers)

To assess the proposed pipeline on a large collection of real-world measurements, we use packet captures from NDT7 tests obtained via the Measurement Lab (M-Lab) data platform. M-Lab is an open measurement consortium that publishes datasets on Internet performance along with the measurement tools needed to support transparent and reproducible research [25]. The traces used in this work come from M-Lab's packet-headers service. For each incoming

TCP flow observed during an NDT7 test, the service records packet headers and stores the results in a separate capture file (one file per flow). The service can optionally anonymize IP addresses and limit capture length to 94 bytes, retaining transport and network headers while reducing storage and privacy exposure compared with full-payload captures.

Our snapshot dataset comprises multiple NDT7 archives totaling 688 MB when compressed. After extraction, the dataset contains 1,340 per-flow capture files, totaling 4.4 GB. The dataset includes large transfers, with the largest individual files reaching approximately 100 MB. This structure matches a realistic bulk-processing setting, where a pipeline must process many medium-sized traces and support repeated queries over the same collection. This dataset was selected to evaluate bulk conversion and repeated querying over a large collection of access-network measurements captured as per-flow traces. Specifically, we consider a more diverse packet mix and a shorter capture length, which stress both field availability and compressibility.

C. CAIDA dataset (backbone-style passive traces)

This dataset was selected to stress the pipeline under backbone conditions with higher traffic diversity and a shorter capture length (snaplen), which reduces field availability and changes compression behavior compared to access-test traces. We use passive backbone traces from CAIDA, specifically the CAIDA UCSD `passive-2016` collection (2016-01-21). The dataset consists of 5 `pcap` files with sizes ranging from 1.5 GB to 1.9 GB. In contrast to the M-Lab packet-header captures, CAIDA traces use a snaplen of 40 bytes. This shorter snaplen may truncate transport headers and options, reducing the amount of information available for certain analyses. Moreover, the small snaplen can affect compressibility, as fewer fields are included and the remaining fields may show greater variability when mixed with backbone traffic.

V. Results

In this section, we describe the outcomes of applying the software tool to the three testing datasets: (i) the benchmark dataset, (ii) the M-Lab dataset, and (iii) the CAIDA dataset. We implement the software architecture in Python using libraries such as `dpkt`, `polars`, `cryptography`, and `psutil`[1]. Furthermore, we conduct experiments on a multi-core server equipped with an Intel(R) Core(TM) i7-7820HQ CPU quad-core (8 threads) running at 2.90GHz, 56 GB of RAM, 1.5 TB of SSD storage, and Ubuntu 24.04 LTS. We report medians over $n = 5$ runs and include 95% confidence intervals where applicable.

A. Benchmark dataset

We evaluate the proposed approach by comparing it against widely used alternatives for offline `pcap` processing. Our goal is to quantify both (i) storage cost, i.e., the size of the resulting artifacts, and (ii) processing cost, i.e., the time required to produce those artifacts. For comparison purposes, we consider five different approaches:

- **`our_sp`:** single-process using our tool that converts `pcap` to `parquet`.
- **`our_mp`:** multi-process using our tool with $p = 8$ worker processes on a quad-core (8-thread) server.
- **`gzip`:** compression of the original `pcap` using `gzip` (archive baseline; no parsing or field extraction).
- **`tshark`:** extraction of selected fields from the `pcap` into a text-based format CSV using `tshark`.
- **`tshark_gz`:** `tshark`-to-CSV extraction followed by `gzip` compression of the resulting CSV file.

All experiments are run over the benchmarking datasets. For each input trace and each pipeline, we record (i) the resulting output size and (ii) the elapsed wall-clock time to complete the pipeline. Elapsed time is measured using `perf stat` on Linux, which reports runtime based on high-resolution performance counters for a command execution. Note that the results are reported as the median of repeated runs to reduce the impact of transient system noise (e.g., caching and background activity).

1) Output size: Figs. 2 and 3 report output size as a function of input `pcap` size for the Network traces and Client traces datasets, respectively (log–log scale; 10 kB to 300 MB). We compare four pipelines: (i) `gzip` compression of the original `pcap` (baseline), (ii) `tshark` field extraction to a text-based format, (iii) `tshark` followed by `gzip` on the extracted output, and (iv) our approach, which converts `pcap` traces to `parquet`. The gray diagonal ($y = x$) indicates the point where output size equals input size; curves below this line correspond to compression, while curves above it

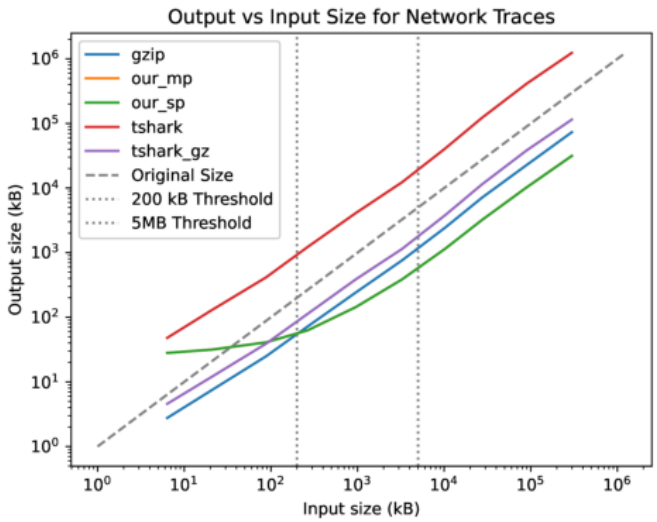

Fig. 2. Output size versus input `pcap` size for the Network traces dataset (log–log scale). Curves report the sizes of the artifacts produced by each pipeline: `gzip`-compressed `pcap`, our approach (single-process and multi-process; `our_mp` is not visible because it overlaps with `our_sp`), `tshark` extraction, and `tshark` output additionally compressed with `gzip`. The dashed diagonal ($y = x$) marks equal input and output size. Vertical dotted lines denote the 250 kB and 5 MB input-size thresholds used to highlight regime changes in overhead and compression behavior.

[1]The source code of the proposed software architecture can be downloaded from: `anonymized for the revision stage.`

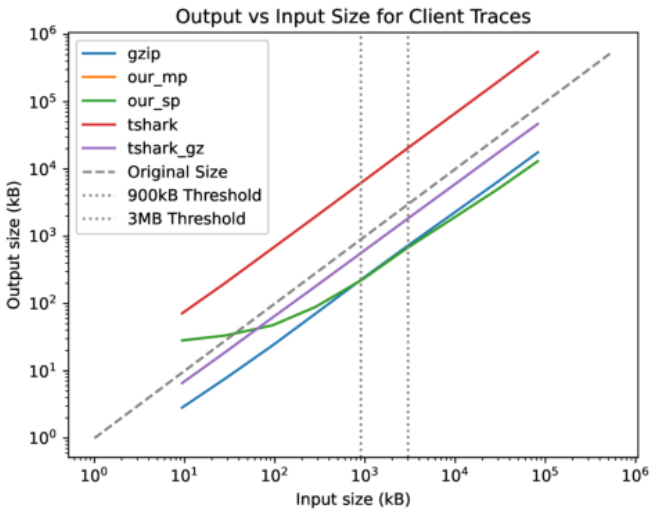

Fig. 3. Output size versus input `pcap` size for Client traces dataset (log–log scale). Vertical dotted lines denote the 900 kB and 3 MB input-size thresholds. Curve `our_mp` is not visible because it overlaps with `our_sp`.

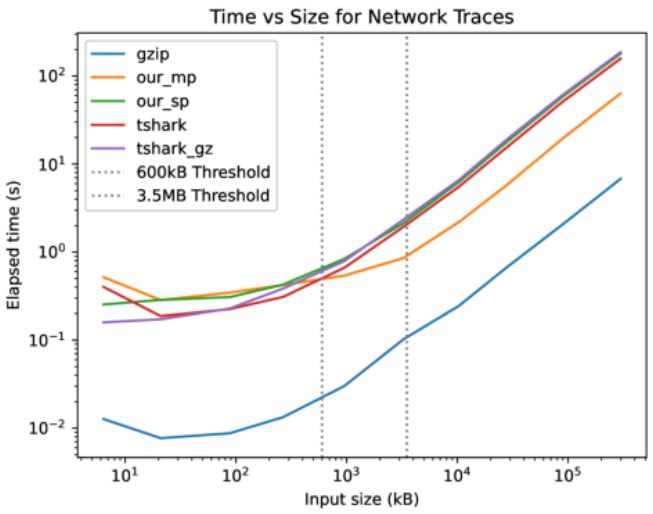

Fig. 4. Conversion time versus input `pcap` size for the Network traces dataset (log scales). Vertical dotted lines mark the 600 kB and 3.5 MB thresholds.

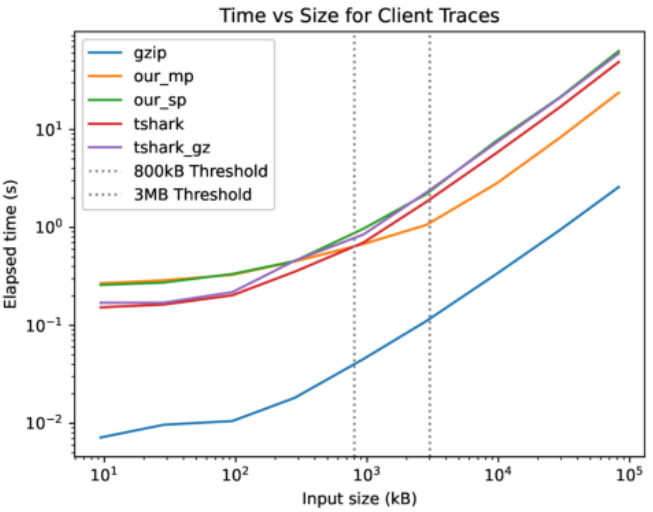

Fig. 5. Conversion time versus input `pcap` size for the Client traces dataset (log scales). Vertical dotted lines mark the 800 kB and 3 MB thresholds.

indicate expansion. Because the single-process and multi-process variants of the proposed tool differ only in execution strategy, they produce identical `parquet` outputs; therefore, their size curves overlap and are shown as a single trend.

Two observations stand out. First, for very small inputs, `parquet` may expand relative to the original `pcap`, as expected due to fixed per-file metadata and row-group overhead. Second, beyond a few hundred kilobytes, the `parquet` outputs consistently fall below the $y = x$ line and become increasingly storage-efficient as file size grows.

For the Network traces dataset, our approach produces the smallest outputs among the evaluated methods once files exceed roughly 0.2 MB (left vertical threshold). For files larger than 5 MiB (right vertical threshold), the `parquet` outputs are approximately 15% of the original `pcap` size, indicating strong compression for the retained header-derived fields. For the Client traces dataset, the same trend holds but with a later crossover: the proposed system becomes consistently more space-efficient than the alternatives for inputs above roughly 900 kB. The smaller gap between `parquet` and `pcap.gz` in Client traces suggests that compressibility depends on dataset characteristics (e.g., value diversity in header fields and flow structure), which affect per-column redundancy and thus compression effectiveness.

2) Conversion time: Figs. 4 and 5 show conversion time versus input size (log scale; 10 ms to 100 s). As expected, compressing the original `pcap` with `gzip` is the fastest option by a wide margin, since it performs no parsing or field extraction. This baseline is therefore best interpreted as an archival lower bound rather than a substitute for analytics-oriented preprocessing.

Among pipelines that parse packets and materialize structured outputs, our software in single-process mode achieves conversion times comparable to `tshark` across both datasets. Multi-process conversion reduces per-file latency for sufficiently large inputs. In the Network traces dataset, the multi-process variant becomes noticeably faster once inputs exceed roughly 600 kB; in the Client traces dataset, the corresponding breakpoint is closer to 3 MB. These breakpoints are consistent with fixed overheads (process startup, partitioning, and write initialization) being amortized as file size increases. Finally, `tshark` followed by `gzip` is consistently slower than `tshark` alone because it adds a second processing stage.

Overall, these results indicate that our approach can (i) produce compact outputs competitive with compressed baselines for sufficiently large traces and (ii) achieve conversion throughput comparable to established tools, while enabling multi-core acceleration to reduce per-file processing time for larger inputs. The remaining question is end-to-end time to insight when analysts execute multiple queries over the same dataset.

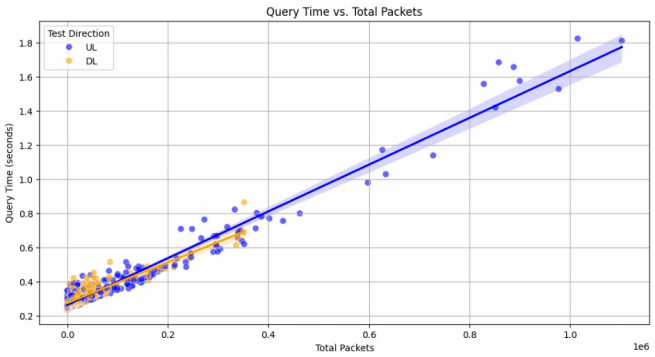

Fig. 6. Measurement Lab dataset. Query runtime versus total packets processed. Each point corresponds to one test trace, colored by test direction (UL/DL). The fitted trend with the 95% confidence interval shows near-linear scaling of query time with packet count.

B. Measurement Lab dataset

We evaluate the proposed pipeline using a large dataset of real-world access network measurements from M-Lab, which includes packet captures from NDT7 tests sourced through the M-Lab data platform [4]. This dataset reflects a realistic bulk-processing regime: a conversion tool must process many moderately sized traces and support repeated analytics on the same collection without re-parsing the original `pcap` files. We report two results on this dataset: (i) bulk conversion time and storage footprint of the resulting `parquet` outputs, and (ii) bulk query performance on the converted dataset.

1) Bulk conversion performance and output size: In this case, we convert the entire collection in a single run using the multi-process configuration to test bulk throughput. The aggregate input file size is 700,009.711 kB, and the resulting `parquet` output file size is 760,944.678 kB. This corresponds to an output-to-input ratio of 1.09×, i.e., approximately a 9% size increase relative to the compressed packet-header inputs. The end-to-end conversion time is 4266.978 s, yielding an average processing speed of 164.053 kbps in this bulk run (using the same size accounting as above). The key observation is that, even when the input traces are already compressed packet-header captures, the `parquet` representation remains close in size, while providing a structured format suitable for repeated analysis.

2) Bulk query performance: To quantify the benefit of the converted representation for repeated analysis, we implement an offline workload in `Polars` that uses lazy scanning over the generated `parquet` files. The query categorizes tests as uplink or downlink by comparing traffic volumes in both directions, and then identifies the file with the highest throughput in each category. The query reports a maximum uplink throughput of 1074.26 Mbps and a maximum downlink throughput of 918.60 Mbps. The total query runtime is 392.77 s, compared to 4266.978 s for the bulk conversion, corresponding to a query-to-conversion ratio of approximately 9.2% (about 10%). At the per-file level, larger traces tend to exhibit a lower query-time-to-conversion-time ratio, which is consistent with fixed conversion overheads being amortized across larger inputs.

Figure 6 further examines per-file query runtime as a function of trace size, using packet count as a workload proxy. The scatter and fitted trend show an approximately linear relationship between query time and the number of packets processed for both UL and DL traces. This behavior is consistent with a scan-dominated workload: for a fixed query plan, runtime is largely proportional to the number of records that must be read and aggregated, with a small constant overhead for query initialization and metadata access.

C. CAIDA dataset

We analyze the pipeline using passive backbone traffic from CAIDA. This dataset poses two main challenges: first, it has a shorter capture length (a snaplen of 40 bytes), which can truncate transport headers and options, limiting the available fields. Second, backbone traffic contains a diverse mix of protocols and endpoints, which increases variation in header-derived fields and reduces the effectiveness of per-column compression compared to traces with consistent, long transfers from a single client.

Table II presents the input `pcap` sizes and the corresponding output sizes produced by our framework and the `gzip` approach for the five files downloaded from CAIDA. Furthermore, this table includes the size ratio of the output files relative to the raw `pcap` size. Across the five traces analyzed, the Parquet outputs are consistently smaller than the raw pcap files, with size ratios ranging from 78.5% to 83.3%. This indicates an average compression rate of approximately 17% in this sample. In contrast, gzip produces significantly smaller archives, ranging from about 48.7% to 53.0% of the size of the raw pcap files, as it is specifically optimized for compressing byte streams. It is important to interpret these results as a trade-off between *archival compression* and *analytics readiness*. While `gzip`-compressed `pcap` files are typically more space-efficient, they remain byte-oriented archives that must be decompressed and fully re-parsed for most analyses. By contrast, the `parquet` representation stores selected header-derived fields in a column-oriented layout that supports repeated queries through selective column reads.

Prior work comparing `pcap` and `parquet` for distributed flow analysis reported cases where the two formats had similar sizes [26]. Our measurements on these CAIDA traces show that, under our schema and encoding choices, the `parquet` outputs can also provide a modest size reduction relative to raw `pcap` while enabling analytics-friendly access patterns.

VI. Conclusion and Future Work

This paper presented a "convert-once, query-many" tool to offline packet-trace analytics that transforms `pcap` traces into a transport-centric, columnar representation. The design targets practical measurement constraints (rapid traffic growth, pervasive encryption, and emerging transport protocols) while remaining compatible with modern data processing systems that benefit from columnar scans and selective access.

Our evaluation supports four main conclusions. First, the proposed pipeline achieves conversion throughput comparable to widely used extraction workflows (e.g., `tshark`).

TABLE II
CAIDA DATASET: RAW PCAP INPUT, PARQUET OUTPUT, AND GZIP-COMPRESSED PCAP (GZ). PERCENTAGES ARE RELATIVE TO THE RAW PCAP SIZE.

File name	PCAP (Bytes)	Parquet (Bytes)	Parquet (%)	GZ (Bytes)	GZ (%)
equinix-chicago.dirA.20160121-125911.UTC.anon	1,462,474,316	1,147,655,903	78.5%	712,421,621	48.7%
equinix-chicago.dirA.20160121-130000.UTC.anon	1,967,267,986	1,639,706,193	83.3%	1,036,945,440	52.7%
equinix-chicago.dirA.20160121-130100.UTC.anon	1,889,522,794	1,555,571,986	82.3%	994,270,302	52.6%
equinix-chicago.dirA.20160121-130200.UTC.anon	1,940,602,578	1,615,693,857	83.3%	1,022,735,289	52.7%
equinix-chicago.dirA.20160121-130300.UTC.anon	1,872,496,132	1,555,577,801	83.1%	991,884,386	53.0%

Then, parallel conversion delivered substantial speedups for individual traces once file sizes exceeded a small threshold (approximately 1 MB in our experiments). Third, the tool exhibited storage efficiency comparable to that of compressed baselines. More precisely, the resulting `parquet` datasets achieve compression levels comparable to the corresponding `gzip` files. Beyond parallel conversion and efficient storage, a central advantage of our design is improved time-to-insight across repeated analyses.

While the current system demonstrates the advantages of columnar packet traces, several directions remain open. We are interested in extending the tool to encompass a wider range of protocols in a modular manner. Future work should focus on ensuring cross-platform reliability and operational effectiveness. It is essential to transition from simply tracking traces to using analytics operators and learning pipelines.

VII. ETHICS

Authors regularly gather test measurements to evaluate the performance of mobile and fixed access networks in various countries. To comply with the GDPR, they follow a privacy-by-design approach that removes or transforms identifiers before releasing any data. The datasets are anonymized, ensuring that no direct user identifiers or operator identities are included. Network-level identifiers are either removed, aggregated, or replaced with privacy-preserving alternatives, such as hashed values, to minimize the risk of record linkage. This process ensures that the released datasets contain only the information necessary for performance analysis while preventing user re-identification and operator attribution.

REFERENCES

[1] G. Harris, "PCAP Capture File Format," Internet-Draft, IETF OPSAWG, Dec. 2025, internet-Draft (work in progress), Dec. 2025. Accessed: 2026-01-19. [Online]. Available: https://ietf-opsawg-wg.github.io/draft-ietf-opsawg-pcap/draft-ietf-opsawg-pcap.html

[2] M. Richardson and M. Tüxen, "PCAP Now Generic (pcapng) Capture File Format," Internet-Draft, IETF OPSAWG (draft-ietf-opsawg-pcapng-04), Aug. 2025, accessed: 2026-01-19. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-opsawg-pcapng/

[3] CAIDA, "The CAIDA Anonymized Internet Traces Dataset (April 2008–January 2019)," https://www.caida.org/catalog/datasets/passive_dataset/, accessed: 2026-01-18.

[4] C. Dovrolis, K. P. Gummadi, A. Kuzmanovic, and S. D. Meinrath, "Measurement Lab: Overview and an Invitation to the Research Community," *ACM SIGCOMM Computer Communication Review*, vol. 40, no. 3, pp. 53–56, Jun. 2010. [Online]. Available: https://dl.acm.org/doi/10.1145/1823844.1823853

[5] The Tcpdump Group, "libpcap," https://www.tcpdump.org/, accessed: 2026-01-18.

[6] ——, "tcpdump," https://www.tcpdump.org/, accessed: 2026-01-18.

[7] "Wireshark," https://www.wireshark.org/, accessed: 2026-01-16.

[8] International Telecommunication Union (ITU), "Facts and Figures 2024: Internet traffic," https://www.itu.int/itu-d/reports/statistics/2024/11/10/ff24-internet-traffic/, Nov. 2024, accessed: 2026-01-19.

[9] Ericsson, "Mobile network traffic update (Q3 2025) – Ericsson Mobility Report," https://www.ericsson.com/en/reports-and-papers/mobility-report/dataforecasts/mobile-traffic-update, Nov. 2025, accessed: 2026-01-19.

[10] J. Iyengar and M. Thomson, "QUIC: A UDP-Based Multiplexed and Secure Transport," RFC 9000, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9000

[11] M. Thomson and S. Turner, "Using TLS to Secure QUIC," RFC 9001, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9001

[12] J. Iyengar and I. Swett, "QUIC Loss Detection and Congestion Control," RFC 9002, May 2021. [Online]. Available: https://www.rfc-editor.org/info/rfc9002

[13] M. Bishop, "HTTP/3," RFC 9114, Jun. 2022. [Online]. Available: https://www.rfc-editor.org/info/rfc9114

[14] The Wireshark Foundation, "TShark: Terminal-based Wireshark," https://www.wireshark.org/docs/man-pages/tshark.html, accessed: 2026-01-18.

[15] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999. [Online]. Available: https://doi.org/10.1016/S1389-1286(99)00112-7

[16] "Zeek Network Security Monitor," https://zeek.org/, accessed: 2026-01-16.

[17] Y. Lee, W. Kang, and Y. Lee, "A Hadoop-Based Packet Trace Processing Tool," in *Traffic Monitoring and Analysis (TMA 2011), Lecture Notes in Computer Science*, vol. 6613. Springer, 2011, pp. 51–63. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-20305-3_5

[18] K. Jeřábek and O. Ryšavý, "Big Data Network Flow Processing Using Apache Spark," in *Proceedings of the 6th Conference on the Engineering of Computer Based Systems (ECBS '19)*. Association for Computing Machinery, 2019, pp. 1–9. [Online]. Available: https://dl.acm.org/doi/10.1145/3352700.3352709

[19] "Apache Parquet," https://parquet.apache.org/, accessed: 2026-01-16.

[20] "Apache Arrow," https://arrow.apache.org/, accessed: 2026-01-16.

[21] "dpkt: fast, simple packet creation / parsing," https://github.com/kbandla/dpkt, accessed: 2026-01-16.

[22] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[23] D. Eastlake, "Transport Layer Security (TLS) Extensions: Extension Definitions," RFC 6066, Jan. 2011. [Online]. Available: https://www.rfc-editor.org/info/rfc6066

[24] E. Kinnear, C. Wood, P. E. McManus, and T. Pauly, "TLS Encrypted Client Hello," RFC 9337, Mar. 2023. [Online]. Available: https://www.rfc-editor.org/info/rfc9337

[25] "M-Lab: Measurement Lab Platform," https://www.measurementlab.net/, accessed: 2026-01-16.

[26] M. Z. N. Saavedra, "A comparison between text, parquet, and PCAP formats for use in distributed network flow analysis on hadoop," *Journal of Advances in Computer Networks*, 2017.