Line-Rate Cybersecurity: Modern DPI and Encrypted Traffic Fingerprinting at 100 Gbps

Luca Deri, Alfredo Cardigliano
ntop
Italy
{deri, cardigliano}@ntop.org

Abstract

Modern network visibility and security depend critically on application-layer traffic understanding. However, the pervasive adoption of encryption and the emergence of purpose-built evasion protocols have substantially degraded the effectiveness of traditional inspection approaches. This paper presents recent advances in nDPI, an open-source Deep Packet Inspection (DPI) library, with a focus on three areas. First, we examine how modern DPI leverages cryptographic handshake metadata to classify encrypted traffic and identify malicious actors without access to plaintext payloads. Second, we expose structural limitations in the widely deployed JA3 and JA4 fingerprinting schemes when confronted with ephemeral TLS extensions, and describe nDPI's composite fingerprinting approach as a more robust alternative. Third, we present the practical integration of nDPI with the Linux Netfilter framework for kernel-enforced traffic policy, and report a comparative, measurement-based evaluation of hardware flow-table offload on two commercially available platforms: an FPGA-based SmartNIC (Napatech NT200 series) and an ARM-based SuperNIC (NVIDIA BlueField-3). The evaluation quantifies flow-table capacity, flow-creation throughput, and host CPU savings on each platform, and identifies the host (or DPU) memory subsystem as the shared scalability bottleneck that neither platform resolves.

Deep Packet Inspection in Modern Networking

The rapid growth of encrypted network traffic poses a fundamental challenge for network monitoring, security analytics, and traffic engineering. Traditional classification techniques based on well-known port numbers have long ceased to be reliable, as modern applications increasingly multiplex over standard ports (80/443), employ encryption, or tunnel their payloads through other protocols. Deep Packet Inspection (DPI) [17] addresses this gap by combining header analysis with payload examination and stateful flow tracking to identify protocols and applications with high accuracy.

nDPI [1, 2] is an open-source C library for high-speed deep packet inspection. Unlike simple port-based classification, nDPI performs stateful flow analysis to detect protocols through pattern matching, behavioral analysis, and protocol-specific dissection. It is designed as an embeddable library rather than a standalone tool: applications feed network packets to nDPI and receive protocol classification results. This design has made nDPI a widely adopted building block in network monitoring, intrusion detection, traffic shaping, and parental-control systems.

The Role of nDPI

nDPI represents each network conversation as a flow, uniquely identified by the standard 5-tuple (source/destination IP, source/destination port, transport protocol). Protocols are modelled as a two-level stack: a master protocol identifying the transport or tunnel layer, and an application protocol identifying the application-layer payload. As Internet traffic increasingly moves towards encrypted content carried over TLS and QUIC, nDPI implements Encrypted Traffic Analysis (ETA) to extract metadata from encrypted sessions and classify them without access to the plaintext. The TLS dissector operates across the handshake phase of a connection, parsing the sequence of handshake messages to accumulate the information required for both classification and metadata extraction. Although the widespread adoption of TLS, QUIC, and HTTP/3 has eliminated visibility into raw packet payloads, it has paradoxically enabled deterministic mechanisms for client identification, as described in the following section.

Fingerprinting Encrypted Traffic

In plaintext protocols, classification relied heavily on signature matching, string parsing, and regular expressions, techniques easily circumvented by minor protocol modifications. Encrypted protocols, by contrast, require a handshake phase during which clients must announce their capabilities prior to establishing a secure channel. This handshake exposes highly distinctive configuration parameters, including supported cipher suites, elliptic curve preferences, and protocol versions. Because these choices are deeply coupled to specific operating systems, cryptographic libraries (e.g., OpenSSL), and browser engines, they constitute a stable behavioral blueprint that is often more reliable than plaintext strings, enabling security devices to classify applications without inspecting private data.

Limitations of Industry-Standard Fingerprinting: JA3 and JA4

To standardize client identification from TLS handshake parameters, the industry adopted JA3 [13] and, more recently,

the JA4 fingerprinting suite [14]. Both methodologies exhibit structural limitations that make them unreliable in practice, even in the absence of adversarial intent.

JA3: Fragility by Design

JA3 encodes explicit TLS parameters (cipher suites, extensions, elliptic curves, and their ordering) into a single MD5 hash. Its reliance on the exact sequence of fields in the TLS ClientHello makes it inherently fragile. Adversaries learned to defeat it by exploiting the TLS GREASE mechanism [16], reordering cipher suites, or injecting arbitrary extensions into the ClientHello. Beyond deliberate evasion, even routine browser updates that alter extension ordering produce a different hash, necessitating continuous signature maintenance. JA3 is today broadly considered obsolete for these reasons.

JA4: An Improvement with a Residual Blind Spot

JA4 attempted to address JA3's weaknesses by introducing length-bucketed sorting and field truncation to produce a more stable, collision-resistant fingerprint string. For many traffic classes this works well. However, JA4 does not account for *ephemeral* TLS extensions, i.e. extensions whose values legitimately vary from session to session even within a single client.

A concrete illustration of this limitation was observed by the authors using a packet capture of Apple Safari (version 26.2) connecting to a production web server from a macOS workstation [30]. Despite all requests originating from the same browser and IP address, the JA4 fingerprint was not stable across sessions:

```
t13d1516h2_8daaf6152771_d8a2da3f94cd
t13d1516h2_8daaf6152771_d8a2da3f94cd
t13d1517h2_8daaf6152771_b6f405a00624
t13d1517h2_8daaf6152771_b6f405a00624
t13d1516h2_8daaf6152771_d8a2da3f94cd
```

Analysis of the capture revealed the cause: Safari uses the TLS Pre-Shared Key (PSK) extension (RFC 8446 [16]) to resume previously established sessions, and the PSK extension carries per-session material that varies with each resumption. Because JA4 incorporates the PSK extension into its hash computation, two connections from the same client, one performing a full handshake and one resuming via PSK, produce different fingerprint values. The same mechanism applies to the Session Ticket extension (RFC 9149) and the Padding extension (RFC 7685), all of which carry session-specific or length-dependent content that legitimately changes between connections.

nDPI's Mitigation: Ephemeral Extension Exclusion

The root cause in both the JA3 and JA4 cases is the same: extensions whose values are intentionally session-specific are treated as stable discriminating features. The fix is correspondingly straightforward, namely exclude known ephemeral extensions from the fingerprint computation.

nDPI implements this via a configurable exclusion list. When enabled, the extensions identified as ephemeral are omitted from the JA4 hash. With this option active, the same Safari capture produces a consistent fingerprint $(\mathrm{t13d1515h2}_8 daaf6152771_3 1ec0a762479) acrossallsessions.$

The exclusion list is intentionally open-ended: as new ephemeral extensions are standardized, they can be added without modifying the core fingerprinting logic. The option is disabled by default to preserve compatibility with existing JA4 deployments, but we recommend enabling it in any environment where fingerprint stability across sessions is required. The broader lesson is that any exact-match fingerprinting scheme applied to a handshake that carries session-specific material will exhibit this class of instability; the mitigation must be part of the fingerprinting specification, not an afterthought applied at the consumer level.

Unmasking Stealth VPNs: Practical Use Cases and Detection Heuristics

Advanced evasion techniques, censorship-circumvention tools, and unauthorized network access frequently rely on stealth VPN protocols. These protocols bypass traditional inspection by stripping structured packet indicators, mimicking standard web sessions, or generating intentionally randomized byte streams.

Representative Evasion Protocols

- **Shadowsocks and VMess/VLESS:** These tools use pre-shared keys or AEAD ciphers to encrypt the initial payload entirely, leaving no plaintext headers. By producing byte streams visually indistinguishable from unstructured TCP data, they defeat port-based and pattern-based classifiers.
- **Trojan Protocol:** This technique wraps proxy traffic inside a well-formed TLS connection directed at a plausible web server. It passes basic SNI and certificate validation checks, rendering standard TLS inspection ineffective.
- **Obfuscated OpenVPN and WireGuard:** These implementations apply custom XOR wrappers or padding layers to alter the standard magic bytes and packet lengths of the underlying protocols, invalidating static protocol signatures.

Encrypted Traffic Analysis and Behavioral Fingerprinting

The fragility of signature-based detection is well illustrated by production examples. The public Snort/Suricata rule for the SUNBURST backdoor [27] matches a literal two-byte TLS record header followed by a hard-coded certificate-subject substring at a fixed offset, a match condition defeated by a single-byte protocol or certificate modification. Also, although machine learning plays an important and growing role in network security, domain knowledge and feature engineering retain substantial value when the underlying signal is well characterized [23]. A strong, well-understood signal can be exploited effectively even with comparatively simple downstream processing. The mechanisms described in this section embody that philosophy. nDPI makes them tractable at line rate through established algorithms: Aho–Corasick automata [24] for substring search, radix trees for IP-prefix

matching, HyperLogLog [25] for approximate cardinality estimation, data binning for traffic clustering.

Metadata Extraction and Flow Risk Model

For HTTP, TLS, QUIC, and DNS traffic, which collectively carry the large majority of today's Internet flows, nDPI extracts protocol metadata even when the payload is opaque. Extracted fields include the TLS/QUIC Server Name Indication (SNI), certificate issuer, subject, validity interval, and SHA-1 fingerprint, the negotiated cipher suite and protocol version, the HTTP `Host` header and `User-Agent`, and the DNS query name. DNS names are additionally evaluated against algorithms for detecting Domain Generation Algorithm (DGA) activity [26], a technique widely used by botnet command-and-control infrastructure to evade static domain blocklists.

Each flow is then evaluated against a catalog of *flow risk* indicators, spanning conditions evaluable from plaintext payloads (e.g., XSS or SQL injection patterns), conditions evaluable from TLS handshake metadata alone (self-signed or expired certificates, obsolete protocol versions, weak cipher suites, a missing or suspicious SNI, a malicious JA3 fingerprint, or a blocklisted certificate hash), and conditions applicable regardless of encryption (a known protocol on a non-standard port, a flagged Autonomous System Number, or anomalously high payload entropy). Each risk indicator carries a severity level (Low, Medium, High, or Severe) and a numeric score, partitioned into client-attributable and server-attributable components.

Statistical Signals: Entropy and Packet-Length Distributions

When metadata alone is insufficient, nDPI falls back on statistical properties of the byte stream that are invariant under encryption. Shannon entropy of the payload bytes provides one such signal: because encryption approximates a uniform byte distribution, well-formed TLS records exhibit measurably higher and more consistent entropy than plaintext protocols. Table 1 reports the mean and standard deviation of payload entropy observed by the authors across three protocol classes. TLS payloads cluster tightly around 7.79 bits/byte - close to the 8-bit theoretical maximum for uniformly random bytes - while DNS and NetFlow cluster lower and with higher variance, reflecting the presence of compressible, structured plaintext fields. Anomalously low entropy on a flow classified as TLS is therefore a usable detection signal. The authors report exactly this pattern, combined with a self-signed certificate finding, in Trickbot command-and-control traffic [23].

Complementing entropy, nDPI bins packet lengths and inter-arrival times into histograms and compares them against known malware traffic profiles. This approach depends only on observable transport-layer metadata and is thus robust to payload encryption.

How To Compose Layers to Reduce Fingerprints False Positives

The JA3/JA4 instability described in Section 2 is compounded by a precision problem: even a fingerprint immune

Table 1: Payload Byte-Entropy Distribution by Protocol (data from [23])

Protocol	Mean Entropy	Std. Dev.
DNS	4.285 bits/byte	0.272
TLS	7.789 bits/byte	0.231
NetFlow	4.079 bits/byte	0.533

to per-session extension randomization can be a poor discriminator if unrelated clients converge on the same value. The nDPI maintainers report a concrete instance: a JA4 fingerprint associated with an Android VPN application also matched unrelated macOS web traffic, making it an unreliable search predicate against a traffic archive [28].

nDPI addresses this by composing, rather than replacing, existing fingerprint layers into a single unified value. The *nDPI Traffic Fingerprint* concatenates three independent components and hashes the result with SHA-256, truncated to 16 bytes (128 bits) and rendered as a hexadecimal string (e.g., `7b8ae0e0763cd4dc7ece67c6231d18a7`): a *TCP fingerprint* derived from Layer-4 connection characteristics (independently usable for client/OS identification, e.g., tagged `/Android` in nDPI's example output), the client-side JA4 string, and a server-side component (the TLS certificate's SHA-1 hash when a certificate is observable, falling back to JA3S when it is not). The fallback is not incidental: TLS 1.3 encrypts the Certificate message under the negotiated handshake keys [16], so the server certificate is unavailable to a passive monitor in the common case, whereas JA3S is derived entirely from the unencrypted ServerHello and remains observable regardless of TLS version.

Applying the TCP fingerprint as an additional constraint to the Android VPN example eliminates the cross-platform false matches, since the unrelated macOS traffic carries a different Layer-4 signature despite sharing the same JA4 string. The format is configurable per deployment via `protos.txt` entries (`ndpifp:<fingerprint>@<label>=<port>`). This is an open, patent-free, and extensible mechanism, rather than a fixed standard.

Note that no single fingerprint value is authoritative: independent traffic sources can converge on identical values, so any fingerprint should corroborate other signals rather than serve as a standalone verdict. An exception applies where the client is unusually distinctive: nDPI reportedly identifies mass-scanning tools such as zMap and Masscan reliably from their fingerprint alone, since minimal, non-randomized network stacks provide little opportunity for convergence with unrelated traffic. This is consistent with the flow-risk model described above: nDPI corroborates multiple independent, partially redundant signals - protocol metadata, payload entropy statistics, and a composite Layer-4/Layer-7 fingerprint - before attributing risk to a flow.

When SNIs Cannot Be Trusted: Detecting Unresolved Hostnames

The TLS/QUIC Server Name Indication (SNI) and the equivalent HTTP `Host` header are widely treated as reliable indicators of a connection's intended destination, but this assump-

tion is eroding on two fronts. Encrypted Client Hello (ECH), discussed further in Section 9, hides the SNI and other ClientHello fields from on-path observers entirely. Even where the SNI remains visible, it is not authenticated end-to-end: a client is free to present an arbitrary hostname while connecting to an unrelated IP address. Evasive applications exploit exactly this gap, presenting a popular, benign hostname in the SNI while routing traffic to a different, potentially malicious destination, in order to blend into flows permitted by SNI-based filtering policies [29].

nDPI mitigates this by passively constructing a DNS cache of (IP address, hostname) pairs from DNS responses observed at the same vantage point, implemented as a bloom-filter-like hash structure to bound lookup cost. Each subsequent TLS/QUIC SNI or HTTP `Host` value is matched against this cache: a flow whose destination IP was never previously observed resolving from the claimed hostname raises the `NDPI_UNRESOLVED_HOSTNAME` flow risk, flagging the SNI as unverified rather than definitively malicious. An important limitation applies: when DNS resolution is itself encrypted - via DNS-over-HTTPS, DNS-over-TLS, or DNS-over-QUIC - nDPI cannot observe the resolving query and may generate false positives due to lack of visibility [29]. Despite this caveat, ntop reports using the mismatch signal operationally within nEdge to build a dynamic blocklist, illustrating the same design principle as the flow-risk model: an individually imperfect signal corroborating other evidence rather than serving as an independent verdict.

nDPI Packet Processing Performance

To contextualize the detection capabilities described above against the throughput requirements discussed throughout this paper, Table 2 reports single-core packet-processing performance measured by the authors on a real traffic capture on a macOS system based on AppleSilicon M1.

Table 2: nDPI Single-Core Performance

Metric	Value
nDPI static memory	3.69 KB
Memory per flow	1.28 KB
Throughput	4.24 Mpps / 10.97 Gbps

On a single core with real traffic, nDPI sustains over 4 million packets per second, equivalent to more than 10 Gbit, with a memory footprint of approximately 1 KB per concurrently dissected flow. Reaching the 100 Gbps target addressed throughout this paper therefore requires distributing traffic across at least 10–16 cores, even before accounting for the random-payload worst case used in the SmartNIC and SuperNIC evaluations presented below. This is precisely the role that RSS-based multi-queue distribution and hardware flow-table offload are designed to play: neither replaces per-flow DPI, but both reduce the number of flows and packets that must be delivered to a core running the DPI engine.

Linux Netfilter Integration

Integrating deep packet analysis into the Linux kernel networking pipeline poses strict stability and safety requirements: nDPI must not introduce memory unsafety, excessive per-packet latency, or kernel-space exposure to the complex dissection logic required for application-layer classification. Rather than executing DPI within a kernel module, nDPI employs a decoupled architecture leveraging the `NF_QUEUE` target and the connection tracking (`conntrack`) infrastructure provided by Netfilter [3].

Asynchronous Flow Inspection Pipeline

The integration exploits the Linux connection tracking (`conntrack`) framework's per-flow *marker*, a 32-bit integer stored inside the kernel's `conntrack` entry for each active flow. Newly established sessions carry a marker value of zero; classified sessions carry a non-zero value equal to the numeric nDPI protocol identifier assigned at classification time. This single field drives the entire dispatch logic. The pipeline operates as follows:

1. **Selective Queuing via Zero-Marker Matching.** Netfilter rules (configured via `iptables` or `nftables`) inspect the `conntrack` marker of each arriving packet. Only packets whose marker is zero, i.e., packets belonging to a not-yet-classified flow, are redirected to a user-space queue via the `NF_QUEUE` target. Packets belonging to already-classified flows carry a non-zero marker and are forwarded or dropped directly in the kernel without ever leaving it.
2. **User-Space Inspection by the nDPI Daemon.** A dedicated daemon receives queued packets via `libnetfilter_queue` and maintains an in-memory hash table keyed on the 5-tuple. Each entry accumulates per-flow state across successive packets of the same session. The daemon feeds each packet into nDPI's dissection engine, which evaluates protocol patterns, cryptographic handshake metadata, and statistical signals as described in the preceding sections.
3. **Incremental Classification and Marker Update.** nDPI classifies most protocols within the first four to eight packets of a session, but may require more for ambiguous or deliberately evasive traffic. While classification is still in progress, the daemon issues an `NF_ACCEPT` verdict without modifying the marker, causing the next packet of the same flow to be re-queued and inspected again. Once a definitive protocol identification is reached, the daemon issues an `NF_ACCEPT` or `NF_DROP` verdict and simultaneously writes the corresponding non-zero nDPI protocol number into the flow's `conntrack` marker via `libnetfilter_conntrack`. Flow that cannot be classified have the dummy Unknown protocol set, to avoid future packets to be processed by nDPI.
4. **Kernel-Enforced Fast Path.** From the next packet onward, the Netfilter rule matching on a zero marker no longer fires for that flow: the non-zero marker causes the queuing rule to be skipped, and the kernel applies the preconfigured policy for that protocol identifier directly, without any further user-space involvement. The nDPI daemon is therefore invoked only for the small initial window of each session; the steady-state cost of a classified flow is effectively zero.

This design provides two important safety properties. First, a crash or restart of the user-space daemon does not affect already-classified flows, whose forwarding or drop policy is enforced entirely inside the kernel. Second, the overhead introduced by nDPI is strictly bounded: only unclassified packets traverse the kernel/user-space boundary, so at steady state, i.e. when the majority of traffic belongs to long-lived, already-classified flows, the queuing path carries a negligible fraction of total traffic volume. In our tests, the overhead introduced by nDPI relative to standard Netfilter is in the range of 2–5%, depending on traffic characteristics. For long-lived sessions such as TLS, the overhead is negligible: because a session typically spans tens to hundreds of packets, the classification cost is amortised across the entire flow and only the first few packets are ever queued. The overhead is proportionally higher for short-lived UDP-based flows such as DNS, where a flow may consist of only one or two packets, meaning the queuing cost is incurred with little or no amortisation benefit.

High-Performance Architecture: 100 Gbps Monitoring and Hardware Flow Offload

Scaling DPI to line-rate 100 Gbit/s connectivity introduces CPU cache and PCIe bus bottlenecks that the standard Linux network stack cannot absorb. Kernel-bypass frameworks such as PF_RING [6] and DPDK [7] address the first-order bottleneck by removing the operating system from the packet reception path: with PF_RING ZC, packets move via DMA directly from the NIC to user-space memory buffers, eliminating per-packet kernel copies.

This does not, however, address the second-order bottleneck that dominates at scale: once the kernel is bypassed, the primary cost for a software-based probe is maintaining per-flow state for every active flow, a cost that scales with the number of concurrent flows rather than with packet rate alone. To achieve deterministic processing at 100 Gbps, flow-state management is offloaded to a SmartNIC, as summarized in Table 3. Once a flow has been classified by nDPI, a bypass rule is programmed into the SmartNIC's flow table. Subsequent packets matching that 5-tuple are handled (forwarded or dropped) entirely in hardware, freeing host CPU cycles completely.

Table 3: SmartNIC Responsibilities in the Offload Pipeline

Component	Responsibility / Function
SmartNIC HW layer	Computes 5-tuple hashes, manages the flow-tracking table in hardware.
First-packet diversion	Unclassified flows are passed via PF_RING ZC to the user-space nDPI engine on the host CPU.
Inline classification	nDPI processes the first 4 to 8 packets to derive application IDs and fingerprint metadata.
Bypass injection	nDPI programs a bypass rule into the SmartNIC, subsequent packets are handled directly in hardware.

From NIC to SmartNIC to SuperNIC

Vendor literature uses the term "SmartNIC" inconsistently. Following the taxonomy proposed in [5], we distinguish three classes of network adapter:

- **NIC:** a standard adapter providing DMA-based reception and transmission, Receive Side Scaling (RSS), and limited static packet filtering, with no per-flow stateful offload capability.
- **SmartNIC:** a NIC augmented with a fixed-function or FPGA-based offload engine capable of stateful, per-flow hardware actions (forward, drop, count), with constrained programmability.
- **SuperNIC:** a SmartNIC augmented with a general-purpose, fully programmable on-board compute subsystem (typically a multi-core ARM or RISC-V processor) and, on some models, direct GPU connectivity for on-path AI/ML acceleration.

Under this taxonomy, the Napatech NT200 series evaluated in Case Study I is a SmartNIC, whereas the NVIDIA BlueField-3 evaluated in Case Study II is a SuperNIC.

Generic Hardware Flow-Offload Pipeline

Despite differing compute models, the flow-offload pattern evaluated on both platforms follows the same sequence of steps, summarized below:

1. Capture the packet via zero-copy DMA (PF_RING ZC or equivalent).
2. Extract the canonical flow key (typically the 5-tuple, optionally extended to inner/outer headers for encapsulated traffic).
3. On a flow-table miss, forward the packet to the software DPI engine for application-layer classification.
4. Once the flow has been classified, program a hardware flow-table entry specifying the associated action.
5. Periodically poll hardware counters and process flow-expiration events to keep the software and hardware flow tables synchronized.

Once a flow entry has been installed, subsequent packets matching its key are processed entirely on the adapter (forwarded, dropped, or counted) without host CPU or PCIe bus involvement beyond periodic, batched statistics polling. The remainder of this section reports a comparative, measurement-based evaluation of this pattern on two commercially available platforms: an FPGA-based SmartNIC (Napatech NT200 series, Case Study I) and an ARM-based SuperNIC (NVIDIA BlueField-3, Case Study II).

Case Study I: FPGA-Based Flow Offload on the Napatech SmartNIC

This case study summarizes the architecture, integration effort, and measured results reported in [4], where hardware

flow-offload support was integrated, via the PF_RING framework, into nProbe Cento, a commercial NetFlow/IPFIX [18] probe that already implements zero-copy packet capture and nDPI-based application classification.

Hardware and Firmware Architecture

The evaluated adapter is a Napatech NT200A02, a PCIe Gen3 card supporting 100 Gbps reception and 100 Gbps transmission, equipped with a Xilinx XCVU5P FPGA and 12 GB of on-board DDR4 memory. The firmware implements a stateful Flow Manager [11] that decodes every received frame in hardware and performs flow classification and lookup against a hardware flow table structured as a cuckoo hash [10]. Up to 10.5 GB of on-board memory is dedicated to the flow table, providing capacity for approximately 140 million concurrent flow entries and approximately 130 ms of ingress traffic buffering at 100 Gbps. The cuckoo-hash structure supports a peak insertion rate of approximately 3.5 million flows/sec below 90% table occupancy, degrading sharply beyond that threshold. The vendor-validated sustained rate used for capacity planning is up to 1 million flows/sec over a single DMA stream, rising to 3 million flows/sec when flow programming is distributed across multiple parallel streams.

The application identifies a flow to the adapter via a 64-bit *flowId*. A `flowId` of zero indicates an unclassified (first-seen) flow. Once the host creates the corresponding software flow-table entry, it programs the `flowId` as the memory address of that entry, which the adapter echoes verbatim on every subsequent packet belonging to the flow and on the flow-expiration event used to reclaim the entry. This design eliminates the need for a flow-table lookup on the host fast path until expiration.

Software Integration

Cento's existing software flow table - one private, per-RSS-queue cache, preserving memory locality and avoiding cross-core locking - is retained even with hardware offload enabled, for three reasons identified in [4]: (i) the hardware table has a hard capacity limit, so flows exceeding that limit must fall back to software processing; (ii) only the software table can hold custom per-flow DPI dissection state (approximately 1 KB per flow during active classification); and (iii) the hardware flow key cannot be extended to incorporate application-layer fields that the NIC does not natively parse, such as a DNS transaction ID. Extending Cento to support hardware flow-table offload required fewer than 200 lines of code, limited to synchronizing the `flowId` between the software and hardware tables and consuming flow-expiration events from the adapter. Programming a single hardware offload rule costs under 10 µs. Cento defers this operation to a dedicated, lightly loaded thread to avoid stalling the packet-processing thread during periods of high flow-creation rate.

Evaluation Methodology

The testbed comprises two servers connected directly via a DAC cable, each equipped with a Napatech 100 Gbps adapter. The device-under-test (Intel Xeon Gold 6526Y, 128 GB DDR5) runs Cento with an NT200A02, The traffic generator (Intel Xeon E-2136, DDR4) runs the open-source `pfsend` tool with an NT100E3. `pfsend` independently controls the number of active flows, the new-flow arrival rate, packet size, and bit rate. On this machine it sustains 80 Gbps at 970-byte packets (10 Mpps) or 60 Gbps at 60-byte packets (89 Mpps). Flows are not pre-allocated, so flow-table insertion overhead is included in the measurements, unlike pre-allocating engines such as Suricata [12]. Traffic carries random payload bytes, representing the worst case for nDPI since every dissector must be evaluated rather than terminating on the first signature match typical of real-world traffic. Following the authors' experience with campus-network and Internet-link captures, the test matrix spans a flow density of 5,000–10,000 flows/Gbps (up to 10 million active flows at 100 Gbps) and a 10% flow birth rate, swept across five operating points from 10,000 to 20 million active flows.

Results

Table 4 presents passive-mode CPU load and packet-drop measurements at the generator's maximum bit rate (10 Mpps, 80 Gbps at 970-byte packets), comparing all combinations of nDPI and hardware offload enabled or disabled.

At 80 Gbps and 10 Mpps, packet loss is zero in every configuration, while hardware offload substantially reduces CPU load, from 100% to 95% at 10 million active flows with nDPI enabled, and far more at lower flow counts (1% versus 35% at 10,000 flows). At the generator's maximum packet rate (89 Mpps, 60 Gbps at 60-byte packets), hardware offload eliminates packet loss entirely for up to 1 million distinct flows and reduces loss by approximately 25 percentage points at 10 million flows. In the inline (bump-in-the-wire) configuration, the PCIe Gen3 bus saturates at the adapter's full 100 Gbps in / 100 Gbps out rate without offload, imposing a hard performance ceiling regardless of CPU headroom. With offload enabled, the majority of forwarded traffic never crosses the PCIe bus, which allows the adapter to scale to bridging and multi-port configurations. Across all configurations, enabling nDPI measurably increases CPU load and, at high flow counts, packet drop, due to the unavoidable cost of per-flow classification.

A key finding from [4] is that hardware flow-table offload does not resolve the worst-case scaling problem for a DPI-enabled probe. Because DPI cannot be performed in hardware, every flow (including those eventually offloaded) still incurs a software flow-table insertion, and the host's memory subsystem (cache and TLB pressure from the flow hash table) remains the bottleneck once flow-creation rate, rather than packet rate, becomes the limiting factor. We return to this open problem in Section 9.

Case Study II: ARM-Based Flow Offload on the NVIDIA BlueField-3 SuperNIC

This case study summarizes the architecture and measurements reported in [5], evaluating whether the flow-offload pattern validated on an FPGA SmartNIC (Case Study I) can be replicated on a SuperNIC using the vendor's programmable SDK rather than a fixed-function hardware control API.

Table 4: Napatech NT200A02: CPU Load and Packet Drop vs. Active Flows, Passive Mode, 10 Mpps / 80 Gbps (data from [4])

Metric	nDPI	Offload	10K	100K	1M	10M / 20M
CPU load	On	Off	35%	58%	100%	100% / 100%
CPU load	Off	Off	31%	36%	87%	100% / 100%
CPU load	On	On	1%	23%	82%	95% / 100%
CPU load	Off	On	1%	17%	46%	98% / 100%
Pkt drop	On	Off	0%	0%	24%	72% / 93%
Pkt drop	Off	Off	0%	0%	0%	40% / 63%
Pkt drop	On	On	0%	0%	0%	46% / 82%
Pkt drop	Off	On	0%	0%	0%	11% / 37%

Hardware and Operating Modes

The NVIDIA BlueField-3 SuperNIC integrates a 16-core ARM Cortex-A78 subsystem, 16 GB of on-board DDR5 memory, PCIe Gen 5.0 host connectivity, and ConnectX-7 network ports supporting up to 400 Gbps. The evaluation in [5] exercises the adapter at up to 100 Gbps, consistent with the scope of this paper. The adapter supports two operating modes relevant to the placement of monitoring applications. In *DPU mode*, the embedded ARM subsystem and on-board eSwitch own the physical network ports, and the x86 host sees only representor ports behind the eSwitch. In *NIC mode*, the adapter operates as a transparent passthrough, behaving as a standard ConnectX NIC directly attached to the host.

Both modes are programmed through DOCA, NVIDIA's SDK for hardware-accelerated packet processing. The *DOCA Flow* library exposes a pipeline abstraction in which *pipes* match, monitor, modify, and forward packets, chained from a root pipe to output ports or to the host application on a miss. The *DOCA Flow CT* (Connection Tracking) pipe specializes this abstraction for stateful 5-tuple flow tracking, providing APIs for entry insertion, removal, and update, per-entry statistics, and flow aging. DOCA Flow CT is available on both the embedded ARM cores and the host side, including on ConnectX adapters without an embedded DPU.

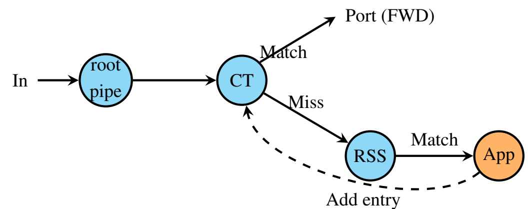

Figure 1: Kryptonite DOCA Flow CT pipeline (adapted from [5]). Packets enter via a root pipe and are dispatched to the CT pipe. Known flows match and are forwarded at line rate (fast path); new flows miss and are steered via RSS to the application, which classifies the flow and installs a CT entry (dashed arrow) to offload subsequent packets.

Proof-of-Concept Implementation

To evaluate DOCA Flow CT under live traffic, the authors implemented *Kryptonite*, an open-source DOCA Flow application (source code available at `github.com/ntop/bluefield-kryptonite`) that realizes the generic offload pipeline of Table **??** on top of DOCA Flow CT. Kryptonite routes incoming packets through a root pipe into the CT pipe: packets matching an existing entry are forwarded at line rate (fast path), packets missing the CT table are steered via RSS to the application for classification, after which a new CT entry is installed to offload the flow going forward (Figure 1). Kryptonite maintains a software shadow flow table mirroring the hardware CT entries to preserve per-flow DPI metadata that the hardware key cannot encode, for the same reasons that motivated the dual software/hardware table design in Case Study I.

Evaluation Methodology

Two configurations were evaluated, both driven by the `pfsend` traffic generator: (i) Kryptonite running on the BlueField-3's embedded ARM cores in DPU mode, and (ii) Kryptonite running on the x86 host (Intel Xeon Gold 6526Y) with the BlueField-3 configured in NIC mode as a ConnectX passthrough. In both configurations, the generator sustained either 100 Gbps at 200-byte packets (55.8 Mpps) or 40 Gbps at 60-byte packets (60 Mpps), with up to 2 million concurrent flows, the maximum flow count reliably configurable on the BlueField-3 CT table in this evaluation.

Results

The measured peak flow-table entry creation rate was 2.7 million flows/sec when Kryptonite ran on the x86 host in NIC mode. When running on the embedded ARM cores in DPU mode, the sustained creation rate was lower at 1.3–1.5 million flows/sec, reflecting the reduced per-core throughput of the ARM subsystem relative to the x86 host, a gap consistent with independent evaluations of earlier BlueField generations [19, 20]. In both configurations, fast-path traffic matching an already-offloaded flow was forwarded at full line rate with zero loss. Packet loss was observed exclusively on the slow path and only once the new-flow arrival rate exceeded the measured creation-rate ceiling for that configuration.

Comparative Analysis: SmartNIC versus SuperNIC Flow Offload

Table 5 summarizes the two evaluations side by side.

Both platforms validate the same architectural thesis that associating per-flow actions with a hardware flow table meaningfully reduces host CPU load and PCIe bus traffic once

Table 5: Napatech SmartNIC vs. BlueField-3 SuperNIC: Measured Flow-Offload Characteristics

Characteristic	Napatech NT200A02 (SmartNIC)	BlueField-3 (SuperNIC)
Offload mechanism	Fixed-function FPGA Flow Manager	Programmable DOCA Flow CT pipe
Measured flow-table capacity	∼140 million flows	2 million flows (configuration tested)
Flow-creation rate	1–3.5 million flows/sec	1.3–2.7 million flows/sec
Action model	Vendor-defined actions (forward/drop/count)	User-programmable pipeline actions
Application placement	Host x86 only	Embedded ARM (DPU) *or* host x86
Host CPU benefit	Substantial reduction at low-to-mid flow counts (Table 4)	Eliminates host involvement for fast-path traffic
Observed limitation	Host memory subsystem bottleneck at high flow-creation rate	Slow-path bottleneck, particularly on embedded ARM cores

a flow has been classified, but they make materially different trade-offs. The Napatech SmartNIC offers an order of magnitude greater flow-table capacity and a mature, narrowly scoped control API, at the cost of a fixed, firmware-defined action set. The BlueField-3 SuperNIC offers full pipeline programmability and the option of running the entire monitoring application on the adapter's embedded compute subsystem, untethered from the host x86, at the cost of an order-of-magnitude smaller flow-table capacity in the tested configuration and a less mature software toolchain. Neither platform eliminates the fundamental scaling problem identified in Case Study I: both require a software DPI pass on every new flow, and in both cases the system performing that pass (the x86 host for Napatech, and either the x86 host or the embedded ARM cores for BlueField-3) remains bottlenecked by its own memory subsystem once flow-creation rate, rather than packet rate, dominates. We regard this as the central open problem for hardware-accelerated DPI at 100 Gbps and beyond.

A methodological caveat applies: the two evaluations were not conducted on identical testbeds, traffic patterns, or firmware/SDK versions, and the flow-table capacities reported are the maxima the authors configured and verified, not necessarily the architectural maxima of each platform. In particular, the 2-million-flow figure for the BlueField-3 reflects the configuration validated in [5] and not a documented hardware limit. The comparison should therefore be read as indicative of architectural trade-offs rather than as a controlled, head-to-head benchmark. A standardized methodology for such a comparison is proposed in Section 9.

Future Research Directions

The case studies above identify several open problems that we consider promising directions for future work, each grounded in an empirically observed limitation rather than a theoretical hypothesis.

Host and DPU Memory Subsystem Scalability

Both evaluations independently identify the same bottleneck: at sufficiently high active-flow counts or flow-creation rates, performance is constrained not by the NIC but by cache and TLB pressure on the software flow hash table maintained by the host (or, on a SuperNIC, by the embedded DPU cores) [4]. Hardware flow-table offload defers this bottleneck but does not eliminate it, since every flow still requires at least one software-side DPI classification pass. This finding is consistent with the broader observation that the software slow path, rather than the hardware fast path, is increasingly the binding constraint in modern NIC accelerator design [22]. Concrete next steps include cache-conscious flow-table layouts, NUMA- and core-aware sharding beyond per-RSS-queue partitioning, and adaptive eviction policies under adversarial flow churn such as DDoS-induced flow-table exhaustion.

A Vendor-Neutral Flow-Offload Abstraction

The comparative evaluation exposes a fragmented programming landscape: Napatech's Flow Manager is controlled through a vendor-specific, fixed-function API, while NVIDIA's DOCA Flow CT is a more expressive but equally vendor-specific SDK with its own object model and release cadence. Neither abstraction is portable to the other vendor's hardware, nor to other SmartNIC or SuperNIC platforms. Controller-driven flow-table programming is not a new concept, OpenFlow [21] proposed it for switches over a decade ago, though adoption for monitoring and security probes was limited by controller-introduced latency and typical table sizes far below the flow counts reported in Table 5. P4 [8] and eBPF/XDP [9] represent more recent and established vendor-neutral abstractions for programmable packet processing. An open question is whether a flow-offload-specific subset of either could express the stateful insert/match/age/forward primitives common to both evaluated platforms, enabling DPI engines such as nDPI to target a single portable offload abstraction rather than per-vendor integration code.

Erosion of Observable Handshake Metadata

Emerging protocol mechanisms are progressively reducing the metadata on which cryptographic fingerprinting depends. Encrypted Client Hello (ECH) conceals the SNI and other ClientHello fields from on-path observers. QUIC's [15] connection-ID rotation undermines the flow-correlation assumptions that both fingerprinting and flow-table offload rely upon. The unresolved-hostname detection mechanism described in Section 4 is itself only a partial mitigation, since it depends on the SNI being visible and on DNS resolution being passively observable. ECH and encrypted DNS (DoH, DoT, DoQ) erode both preconditions simultane-

ously. As these mechanisms see wider deployment alongside TLS 1.3 [16], DPI engines will require fallback classification based on packet timing and size distributions rather than handshake content, a substantially more challenging and currently less mature classification problem.

Conclusion

This paper reports advances in nDPI across three areas: cryptographic fingerprinting robustness, kernel-integrated traffic enforcement, and hardware-accelerated flow offload at 100 Gbps. The principal technical findings are as follows.

1. JA4's length-bucketed hashing is more resilient than JA3's rigid parameter concatenation, but both remain structurally vulnerable to ephemeral TLS extensions, since any exact-match fingerprinting scheme is sensitive to deliberate or incidental per-session parameter randomization [13, 14].
2. Production detection heuristics for stealth proxy protocols (Shadowsocks, VMess/VLESS, Trojan, obfuscated OpenVPN/WireGuard) rely on behavioral and statistical signals rather than static signatures, because these protocols are explicitly engineered to defeat the latter.
3. A decoupled `NF_QUEUE`/`conntrack` integration [3] allows nDPI to classify the initial packets of a flow in user space while the kernel enforces the resulting verdict natively for all subsequent packets, bounding DPI's exposure to kernel-space failure modes.
4. Hardware flow-table offload has been empirically validated on two different platforms: an FPGA-based SmartNIC (Napatech NT200A02, ∼140 million flow entries, 1–3.5 million flows/sec programming rate) and an ARM-based SuperNIC (NVIDIA BlueField-3, 2 million flow entries in the tested configuration, 1.3–2.7 million flows/sec) [4, 5]. Both platforms reduce host CPU load. On the FPGA platform, offload eliminates packet loss for up to 1 million concurrent flows at the generator's maximum packet rate.
5. Neither offload platform resolves the underlying scalability limit: software DPI classification is still required for every new flow, and the host's (or DPU's) memory subsystem (not the NIC) becomes the bottleneck once flow-creation rate dominates packet rate. We regard this as the central open problem for line-rate DPI and discuss directions for addressing it in Section 9.

These results should be interpreted as a measurement-based progress report rather than a closed result: the comparative figures derive from two independently conducted evaluations rather than a single controlled benchmark, and the fingerprinting and offload techniques discussed throughout this paper are evaluated against evasion and protocol-evolution trends that will continue to evolve.

Authors' Biographies

Luca Deri is the founder of ntop, a company specializing in open-source, high-performance network traffic monitoring and cybersecurity. He created the original ntop network monitoring tool and has since led the development of various tools adopted in both academia and industry for deep packet inspection and traffic analysis. Luca holds a Ph.D. in Computer Science from the University of Berne, Switzerland, and has authored numerous papers and tools on network traffic analysis. He is a lecturer in the Department of Computer Science at the University of Pisa, Italy, and regularly presents his work at international networking and security conferences.

Alfredo Cardigliano is a principal engineer at ntop, where he has contributed for over a decade to the development of high-performance packet-processing technologies, including PF_RING. His work focuses on kernel/user-space integration for line-rate network monitoring, traffic classification, and traffic recording. He contributes to ntop's open-source projects on GitHub, supporting both research and production deployments of network visibility tools, and has contributed with original device drivers for SmartNICs to the DPDK project.

References

[1] L. Deri, M. Martinelli, T. Bujlow, and A. Cardigliano. nDPI: Quick and Extensible Deep Packet Inspection for Practical Network Architecture and Management. In *Proceedings of the 2014 International Conference on Communications and Communication Security*, 2014. Open source repository: `github.com`.

[2] ntop. *nDPI source code*, 2026. `https://github.com/ntop/`

[3] Netfilter Project. *Infrastructure for Network Packet Filtering Framework in Linux*. User-space library and queue parameter specifications: `netfilter.org`.

[4] L. Deri, A. Cardigliano, and F. Fusco. Advancements in Traffic Processing Using Programmable Hardware Flow Offload. 2024. `https://luca.ntop.org/NetAccel-AI_2024.pdf`

[5] A. Cardigliano. Unleashing SuperNIC's Superpowers. FOSDEM 2025, Brussels, Belgium, February 2025. `https://archive.fosdem.org/2025/events/attachments/fosdem-2025-5559-unleashing-supernic-s-superpowers/slides/238432/SuperNICs_UMPmQUf.pdf`

[6] F. Fusco and L. Deri. High Speed Network Traffic Analysis with Commodity Multi-Core Systems. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC)*, 2010.

[7] Intel Corporation. *DPDK: Data Plane Development Kit*, 2014. `https://www.dpdk.org/`

[8] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker. P4: Programming Protocol-Independent Packet Processors. *ACM SIGCOMM Computer Communication Review*, 44(3):87–95, 2014.

[9] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, and D. Miller. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th ACM/IFIP/USENIX International Conference*

on Emerging Networking Experiments and Technologies (CoNEXT), 2018.

[10] R. Pagh and F. F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[11] Napatech A/S. *Stateful Flow Management*, 2024. https://docs.napatech.com/r/Stateful-Flow-Management

[12] Open Information Security Foundation. *Suricata IDS/IPS/NSM Engine*. https://suricata.io

[13] J. Althouse, J. Atkinson, and J. Atkins. TLS Fingerprinting with JA3 and JA3S. *Salesforce Engineering Blog*, 2017.

[14] J. Althouse. JA4+ Network Fingerprinting. FoxIO, 2023. https://github.com/FoxIO-LLC/ja4

[15] J. Iyengar and M. Thomson (eds.). *QUIC: A UDP-Based Multiplexed and Secure Transport*. IETF RFC 9000, 2021.

[16] E. Rescorla. *The Transport Layer Security (TLS) Protocol Version 1.3*. IETF RFC 8446, 2018.

[17] T. Bujlow, V. Carela-Español, and P. Barlet-Ros. Independent Comparison of Popular DPI Tools for Traffic Classification. *Computer Networks*, 76:75–89, 2015.

[18] B. Trammell and E. Boschi. An Introduction to IP Flow Information Export (IPFIX). *IEEE Communications Magazine*, 49(4):89–95, 2011.

[19] J. Liu, C. Maltzahn, C. Ulmer, and M. L. Curry. Performance Characteristics of the BlueField-2 SmartNIC. *arXiv preprint arXiv:2105.06619*, 2021.

[20] S. Karamati, J. Young, T. Conte, K. S. Hemmert, R. Grant, C. Hughes, and R. Vuduc. Computational Offload with BlueField Smart NICs. Sandia Report SAND2021-13031, 2021.

[21] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[22] A. Zulfiqar et al. The Slow Path Needs an Accelerator Too. *ACM SIGCOMM Computer Communication Review*, 53(1):38–47, 2023.

[23] L. Deri. Using Deep Packet Inspection for Monitoring and Security. ntop presentation, 2022.

[24] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, 1975.

[25] P. Flajolet, É. Fusy, O. Gandouet, and F. Meunier. HyperLogLog: The Analysis of a Near-Optimal Cardinality Estimation Algorithm. In *Proceedings of the International Conference on Analysis of Algorithms (AofA)*, 2007.

[26] M. Antonakakis, R. Perdisci, Y. Nadji, N. Vasiloglou, S. Abu-Nimeh, W. Lee, and D. Dagon. From Throw-Away Traffic to Bots: Detecting the Rise of DGA-Based Malware. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[27] Mandiant. *SUNBURST Countermeasures*. GitHub repository, 2020. https://github.com/mandiant/sunburst_countermeasures

[28] ntop. Beyond JA3/JA4: Introducing nDPI Traffic Fingerprint. ntop.org blog, August 2025. https://www.ntop.org/beyond-ja3-ja4-introducing-ndpi-traffic-fingerprint/

[29] ntop. When SNIs Cannot Be Trusted. ntop.org blog, October 2025. https://www.ntop.org/when-snis-cannot-be-trusted/

[30] ntop. Is JA4 Now Obsolete? ntop.org blog, January 2026. https://www.ntop.org/is-ja4-now-obsolete/