Rethinking the Role of Network Stacks for Website Fingerprinting Defenses

Elisaveta Lavrentieva, Marc Juarez*, and Michio Honda*
University of Edinburgh

Abstract

While encryption has become ubiquitous across the Internet, there is growing concern that website fingerprinting and other traffic analysis attacks could undermine the confidentiality guarantees encryption is meant to provide. Over the past decade, these attacks have become increasingly effective, highlighting the urgent need to deploy traffic obfuscation countermeasures. Although defenses have already been proposed in the literature, they remain inefficient partly because they are implemented at the application-level, which limits their control over packet sequences. This paper advocates for integrating packet sequence obfuscation support directly into host network stacks, where the fine-grained packet operations that defenses require can be effectively enforced.

CCS Concepts

• **Networks** → **Network security**; • **Software and its engineering** → **Operating systems**.

Keywords

Website fingerprinting, Network stacks

ACM Reference Format:
Elisaveta Lavrentieva, Marc Juarez*, and Michio Honda*, . 2025. Rethinking the Role of Network Stacks for Website Fingerprinting Defenses. In *The 24th ACM Workshop on Hot Topics in Networks (HotNets '25), November 17–18, 2025, College Park, MD, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3772356.3772428

1 Introduction

The layered architecture of the host network stack plays a critical role in enabling robust and scalable Internet communications. In addition to housing individual network protocols, the stack implements various subsystems that support system-wide and application-level policy enforcement (e.g., packet scheduling), hardware offload management in the network interface card (NIC), efficient data movement, and cross-layer event notification.

Today, the primary goals in data transmission, which consist of sending the application data to the network in packets, are: efficiency of CPU cycles and network bandwidth usage, mitigation of network congestion, and fairness guarantees between network flows. When the application posts data to the stack, before it reaches the network layer, the transport layer decides *when* to send the data, *over how many packets*, and *how to interleave* those packets over time. The decision is based on end-to-end path characteristics observed in the flow, such as MTU, RTT, and congestion level. It also takes into account the policies specified by the applications, such as priorities for low delay or high throughput.

This paper advocates for adding another objective in the data transmission process of the host network stack: *traffic obfuscation*. This objective has become critical due to the increased effectiveness and relevance of traffic analysis attacks, most notably through website fingerprinting (WF). WF is a class of traffic analysis techniques potentially used by passive eavesdroppers, such as censorship devices. By applying ML techniques on traffic metadata, which includes packet sizes and timing, WF techniques extract patterns that are unique to the webpage. These distinctive patterns reveal the website identity, even when the conntection has been encrypted or anonymized.

As WF can undermine encryption's confidentiality guarantees, it has become increasingly relevant in the era of *default-encrypted* Internet communication, attracting growing attention from the privacy and security community. In particular, the application of Deep Learning (DL) techniques to the design of WF attacks has led to breakthroughs that significantly boost their practical effectiveness [41, 48, 5], further heightening concerns about the threat WF poses to Internet users.

Several WF defenses that obfuscate the traffic features exploited by the attacks have been proposed in the literature. However, while most of these defenses are designed to operate on the traffic sequences observed by the attacker, they are instead implemented in the application layer and thus operate on application-level data that is yet to pass through

*Equal supervision role.

HotNets '25, College Park, MD, USA

ACM ISBN 979-8-4007-2280-6/2025/11
https://doi.org/10.1145/3772356.3772428

the network stack [13, 7, 30]. As a result, the intended modifications to packet-level features may not be reflected in the actual network traffic and may overlook the prohibitive performance penalties of a practical implementation. Only defenses operating at the network stack level can reliably and efficiently enforce these defensive transformations.

To address this limitation, we propose a traffic obfuscation framework implemented within the network stack, enabling existing and new WF defenses to operate directly on the final packet sequences, thus truly enforcing their obfuscation policies. To determine the feasibility of this approach, this paper addresses the following questions:

- Do WF attacks pose a significant threat? In particular, is the high accuracy of these attacks a cause for concern? And how likely are they to be applied in the wild? (§ 2)
- Is data transmission the right place to enable traffic obfuscation, and would it be effective in protecting against website fingerprinting? (§ 3)
- What could abstractions for enforcing obfuscation policies between the application and network stack look like? (§ 4)

In addition, we also discuss implications in the Internet architecture, transport protocol design, and service deployment model (§ 5).

System	Target	Strategy	Traffic Manipulation
ALPaCA [12]	Tor	Regul.	Padding
BuFLO [13, 7, 8] RegulaTor [23] Surakav [17] Palette [44]			Padding, and timing modification
WTF-PAD [26] FRONT [16] BLANKET [34]		Obfus.	Padding, and timing modification
Morphing [55] HTTPOS [30] Burst Defe. [32] Cactus [56]	TLS		Timing & packet size modification
Ad. FRONT [21]			Padding, and timing modification
QCSD [49] $\text{pad}_{\text{resour.}}$ [46]	QUIC		Padding, and timing & packet size modification
NetShaper [42]	TLS & QUIC		

Table 1: WF defense summary. Defenses for TLS/QUIC obfuscate traffic with padding, and timing or packet size modifications. For simplicity, *padding* encompasses countermeasure operations that aim to change web object sizes, regardless of which layer implements them. This differs from *packet size modification* which is packet-level padding with the sole objective to change the sizes of individual packets.

2 Website Fingerprinting

This section discusses the growing relevance of WF for passive network eavesdroppers, given both the widespread deployment of end-to-end encryption on the Internet and the increasing viability of state-of-the-art WF techniques.

2.1 Momentum

Passive network adversaries are widespresad across the Internet. They monitor network traffic for a variety of purposes, including censorship, content filtering, and targeted advertising. Their common objective is to identify the specific website that a client is visiting. When the Internet was mostly unencrypted [6], this could be achieved by simply inspecting the plaintext HTTP requests and DNS queries. However, with the growing adoption of TLS for both HTTP and DNS, these adversaries must resort to alternative methods. One key piece of metadata that remains exposed is the Server Name Indication (SNI) [28, 35], but this is starting to change. Since late 2024, Cloudflare, which hosts a significant portion of the web through its CDN infrastructure, has rolled out support for Encrypted Client Hello (ECH). ECH encrypts most of the TLS handshake, including the SNI, using a public key retrieved via DNS.

As the ability to read plaintext SNI declines, and other destination metadata remains encrypted, adversaries are likely to turn to traffic analysis techniques to infer web browsing activity. WF, one such a technique, has been shown effective in identifying the websites that a client visits from analyzing the timing, direction, and size of encrypted packets, thereby circumventing the protection encryption is meant to provide.

Recent advancements in the field of ML, particularly DL, have significantly improved the practical feasibility of these attacks on HTTPS and Tor traffic. Although some studies show that analysis of encrypted DNS traffic can aid the attacks [47], the vast majority of WF attacks are carried out on the connection path between the client and the web server, where more information to exploit is available [14, 50, 46]. Therefore, for the remainder of this paper, we focus exclusively on WF techniques applied to client-server traffic, rather than those targeting DNS traffic.

Given these trends, it is highly likely that passive eavesdroppers will increasingly adopt WF techniques. However, note that confirming this in practice is hard, as WF attacks are inherently passive and leave no traces. Detecting WF in the wild would require indirect evidence, such as side-effects or public disclosure of the practices.

2.2 Status Quo

The ongoing interest in WF is justified by the potentially severe consequences of a successful WF attack, the relatively low cost of deploying such attacks, and the high success rates reported in the academic literature. The study of WF techniques began in the context of the Tor overlay network, which was specifically designed for anonymous, encrypted

communication [48, 41, 5], and has only become relevant to the broader Internet with the recent widespread adoption of encryption (e.g., Google started default-HTTPS in 2010). Today, WF research increasingly considers direct Internet communications over TLS [10, 9, 32] and QUIC [58, 49, 46], where the adversary uses WF to identify a website within a CDN or a specific page within a website. Although WF studies still make simplified assumptions on the evaluation [36, 25] and the practical feasibility of attacks remains contested, there is increasing interest in defenses that protect users in high-stakes applications [37], as well as in techniques developed for Tor that influence the design of defenses for direct Internet communication.

Table 1 summarizes existing WF defenses based on their strategy, target and methodology. WF defense strategies divide into *regularization*, which attempts to make packet sequence across websites more similar, and *obfuscation*, which adds noise to the features exploited by the attacks. Defenses are also classified by the abstraction level in which they are specified. For Tor, several defenses are implemented at the browser or web server application for HTTP request/response manipulation and file object padding [12]. However, while there is incentive for Onion service operators to deploy such defenses, this is not the case yet for the broader Internet. Defenses for direct Internet communications are normally specified on the wire packet sequences, as they would be observed by the attacker. At that level of abstraction, defenses often adopt obfuscation, as there is no information for effective regularization.

Obfuscation-based defenses against WF typically rely on three primitives: packet size modification, timing modification, and padding. These primitives are typically combined to design complex defensive strategies to obscure distinctive web traffic patterns. Packet size modification involves changing packet boundaries of the transferring data (e.g., using smaller packets than MSS), making packet sizes less distinctive. Timing modification adds artificial delays to packet transmissions. Finally, padding refers to constructing and sending packets that do not carry real user data (also known as *dummy* packets), used to obscure higher-level characteristics such as total size of web objects and total volume of the communications.

The application of DL techniques for the development of WF has led to dramatic improvements in their accuracy. In particular, some attacks have achieved over 95% accuracy against Tor [48, 41, 4], which has been designed to protect against some types of traffic analysis. Although these results have been criticized for relying on simplified assumptions in the evaluation setting, there is growing concern among Tor developers and the broader privacy community regarding the threat that WF attacks pose [25]. For this reason, the community is increasingly open to the deployment of lightweight defenses that can raise the bar for WF attackers—particularly in deployment scenarios where the research assumptions hold [26, 37]. We believe the same motivation emerges in the open Internet.

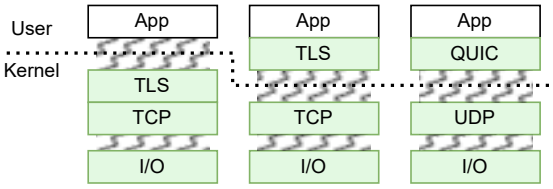

Figure 1: The stack model. The term "stack" refers to the shaded parts. Zigzag-connected parts are executed asynchronously.

2.3 Problems with Existing Defenses

Enforcing packet obfuscation algorithms on the actual packet sequence of ordinary applications is challenging, because network stacks lack abstractions for fine-grained control over packet sequences. Although WF studies prototype their proposed algorithms on top of the transport layer, the resulting packet sequence would match the expectation only if the application limits the bandwidth below the congestion window size (and thus enforces application-limited flows) and disables various batching or queuing mechanisms. Those assumptions are impractical and prohibitive in terms of performance penalty. However, without enforcing those conditions, application data posted to the stack—scheduled by the obfuscation algorithm running in the application layer—could be *deferred* for transmission or *coalesced* to the preceding data, as detailed next.

Figure 1 illustrates where such asynchronous data transmission occurs in common organizations of application and stack: TLS over TCP, in-kernel TLS (kTLS [27, 38]) over TCP[1] and QUIC over UDP. We use the term stack to refer to the layers between transport protocol implementation and NIC I/O, inclusive (colored blocks in the figure) hereafter.

The first asynchronous processing happens in the transport layer. In the `send()` syscall, application data is copied to the socket buffer, or the page pointer is linked to it if zero copy is opted. This data is sent to the network layer directly in the syscall context only when a sufficient receiver and congestion window is available; otherwise the syscall returns to the application and the data transmission is deferred until the stack receives the ACK packets that advance the window.

The other asynchronous I/O happens at the bottom of the stack. A packet pushed by the TCP or UDP implementation to the lower layer could be processed by another thread that dequeues the packet from the queuing discipline (e.g., for fair

[1]This model also applies to in-kernel encrypted transport protocols [15].

queueing between the flows) or enforces TCP small queue (in-host buffer bloat mitigation [11]). Furthermore, modern TCP implementations use TCP segmentation offload (TSO), which splits a large TCP segment to MSS-sized packets at the NIC to reduce the packetization overheads and the number of network layer traversals. Since TSO sends the packets that it generates at the line rate without interleaving them, the application cannot control the timing of those packets.

Application-level timing control of data transmission is not impossible, but it is extremely inefficient and impractical because it must ensure no data exceeding the current window is pushed to the stack and no other application is transmitting the data to the shared NIC, also disabling TSO.

Application-level traffic obfuscation also lacks packet size control unless the application ensures not to transmit data larger than MSS or advertised window. Since TCP provides the bytestream abstraction, packet size is determined by the TCP implementation based on the measured path MTU (PMTU) and advertised window if it is smaller than PMTU. Therefore, controlling packet size at the application layer is practical only for applications that send small messages, such as an interactive shell.

Controlling the packet sizes for large application data is hard with the abstraction currently available in commodity OSes. Furthermore, TSO creates fixed-sized packets, except for the last one. As a relevant inefficient example, HTTPOS [30] attempts to control packet size by advertizing very small window size and MSS. However, small MSS values apply for the connection lifetime and thus damages transmission efficiency; small advertised window prevents the server from sending the full congestion window of data, sacrificing bandwidth utilization and thus throughput.

This observation is based on TCP, but the same will apply to QUIC. Although it runs on top of UDP, since QUIC also provides stream abstractions, packet size is determined by QUIC based on its PMTU discovery. Datagram transmission to the UDP layer is also scheduled by QUIC based on its congestion control, rather than the application. Moreover, there exists a sign of hardware offload for segmentation and encryption for QUIC [19] as the latest Broadcom BCM57608 NICs, which is already available on the market. Such NICs and their abstractions lead to similar segmentation behavior to TLS/TCP.

Padding-based methods are harmful to the network, because it consumes extra network bandwidth. For example, FRONT [16] introduces 80% of bandwidth overhead [44] and QCSD introduces 309% of that [49]. Padding is worse than timing control, because it wastes network bandwidth in a non-work-conserving manner. Timing manipulation, such as delaying packets, leaves the idle resource for other flows. Using smaller packet sizes is not as harmful as padding, because although it decreases CPU efficiency, it does not consume additional network bandwidth apart from the overhead of TCP/IP headers. We do not argue that the padding-based methods must be avoided, but the current systems tend to rely on those methods excessively due to the lack of robust control of packet timing and sizing.

3 Stack-Viable Defenses Against WF-Enabled Censorship

In this section, we assess the potential of stack-level countermeasures to mitigate WF in a censorship scenario. Because censors must decide to block a website before the client downloads it, the attack must be able to detect the web page early in the connection. Censorship is a realistic WF attack, because it does not need large storage space or packet processing CPU cycles to record a large amount of traffic.

In this experiment, we have collected web traffic data and emulated packet modifications that can be implemented in a kernel-level defense. To evaluate the censorship setting, we apply the attack on only the first few packets of a network trace.

To collect data for this evaluation, we use `tcpdump` to capture traffic for 9 popular websites, selected from the Tranco top 1M websites [39]. These were: `bing.com`, `github.com`, `instagram.com`, `netflix.com`, `office.com`, `spotify.com`, `whatsapp.net`, `wikipedia.org`, and `youtube.com`. For each website, we collected 100 samples—all collected within 3 hours—and extracted packet timestamps and directions to train k-FP, a WF attack that is still commonly used in benchmarks in the literature [22].

After sanitizing the data by checking for connection errors and removing outliers outside of the interquartile range of total download size, we were left with 74 traces for each site. We used the resulting data to train k-FP in a closed-world setting, meaning that the test traces can only visit the 9 selected sites instead of any other site available on the internet, greatly limiting the scope for k-FP. This corresponds to the most favorable conditions for the attacker, therefore our results represent an upper bound on attack success.

With these unmodified traces, we emulate two potential packet sequence modifications that can be implemented in the kernel: packet splitting and delaying. We emulate splitting by dividing packets of size larger than 1200 bytes into two individual packets of half the size of the original packet. To implement packet delaying, we increment the inter-arrival time between the original packet and the one before by 10–30%, where the percentage is drawn uniformly at random. These countermeasures are only applied on incoming traffic from the server, emulating a deployment of the defense at the server-side.

While packet splitting and delaying can be implemented in the stack, we selected conservative parameters to avoid

aggressive traffic modifications. For example, we set this packet splitting threshold to prevent creating packets that are smaller than the minimum TCP MSS of 536 bytes [53], which would not occur in real-world connections. Similarly, we use small random delays because larger delays could trigger retransmission timeouts. Our choice of parameters could thus provide a conservative estimate of the effect of the countermeasures.

We apply these countermeasures individually and in combination on the entire unmodified traffic, generating three protected datasets. To evaluate the censorship scenario, where blocking decisions occur early in the connection, we also apply the countermeasures on the first 15, 30, and 45 packets only, resulting in a total of 16 datasets.

Table 2: k-FP Random Forest accuracy rates. k-FP is distinguishing between 9 sites in a closed-world setting. *N* indicates the number of packets at the beginning of the trace on which the attack is applied.

N	Original	Split	Delayed	Combined
15	0.798 ± 0.017	0.825 ± 0.024	0.825 ± 0.030	0.795 ± 0.031
30	0.884 ± 0.007	0.860 ± 0.013	0.855 ± 0.030	0.850 ± 0.062
45	0.938 ± 0.016	0.897 ± 0.030	0.913 ± 0.021	0.904 ± 0.004
All	0.963 ± 0.002	0.980 ± 0.008	0.980 ± 0.014	0.992 ± 0.009

Table 2 shows that k-FP achieves high accuracy in this closed-world setting, with accuracy increasing as more packets are observed—an expected result. Remarkably, the countermeasures actually increase attack accuracy on the 'All' dataset containing complete traffic traces. This counterintuitive result is possibly due to the added noise being insufficient to fool the classifier, while the randomness might have introduced unique patterns that facilitate fingerprinting. On the other hand, the rate at which k-FP's accuracy increases over *N* is slower when either defense is applied compared to no defense, indicating that countermeasures delay confident detection in the censorship setting.

Although this preliminary result offers modest promise, implementing these and more sophisticated countermeasures at the kernel level is likely to enable a broader range of tunable parameters and thus a greater effectiveness. As a next step, we will implement a traffic obfuscation framework in the network stack that enables the development of more advanced defensive strategies. In turn, this will allow a more accurate assessment of the potential of these countermeasures in mitigating the attacks.

It is important to note that splitting packets also inherently adds a delay to the connection, as more packets must be sent out to complete the connection. It may be that a combination of delay and packet size would have compound effects in the features and the overheads. An evaluation of the effects of combinations of these variables and more complex defensive strategies is our ongoing work.

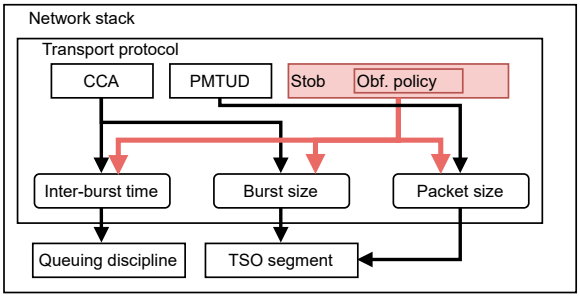

Figure 2: The Stob architecture.

4 Stack Support for WF Defense

Based on the potential effect of stack-level packet sequence control, we now design **Stob**, **s**tack-level **t**raffic **ob**fuscation.

4.1 Architecture

The design goal of Stob is to enable fine-grained control over the packet sequence sent by the host, so as to accommodate a range of packet obfuscation algorithms and trade-offs between protection strength and cost. To this end, Stob directly operates in the network stack, spanning across the transport and packet I/O layers where the relevant decisions, including packet size and departure time, are ultimately made. Figure 2 illustrates the Stob architecture.

The packet obfuscation policies are determined by the application or system administrator. Since the packet departure time and size applied to data units can be represented as relatively compact distribution functions like histograms (§ 2.2) and their instances can be shared between flows in some cases (e.g., same destination), those policies could be maintained in the shared memory between the application and stack.

4.2 Packet Sequence Control

Stob affects the decision being made by the transport protocol about *when to send the application data—over how many packets or in what packet size—by how long interleaving them.*

The first relevant decision is the TSO size—the size of the transport-level segment that is passed to the NIC, which splits it to MSS-sized or smaller packets (§ 2.3). TCP ideally wishes to create a largest-possible (i.e., 65 K B) segment for CPU efficiency, but it often does not, because TSO creates a *micro burst*, a series of line-rate packets, which increase the amount of data queued in the network router. Since the packets in the same TSO segment cannot be interleaved, the Linux TCP implementation chooses a small TSO size at

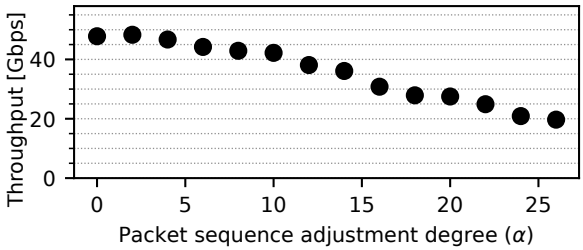

Figure 3: Packet and TSO size adjustment. The horizontal axis indicates the maximum reduction degree of packet size and TSO size. Packet size reduces from 1500 (default) by α up to $1500 - \alpha \times 10$ bytes (reset to the default then and repeat) and TSO size does so from 44 (default) by $\alpha/4$ up to $44 - \alpha/4 \times 8$ or 1.

the expense of CPU efficiency when the remote endpoint is distant (i.e., long RTT) to perform fine-grained pacing of data and corresponding ACK packets. Another relevant feature is pacing. The Linux stack provides a pacing queue primarily for modern congestion control algorithms like BBR, which allows the congestion control algorithm (CCA) to schedule the departure of TSO segment in a nanosecond granularity.

Based on these observations, Stob interacts with TSO sizing and pacing decisions usually controlled by congestion control and packet sizing decision usually made by the advertised MSS and PMTU. For example, even if the CCA decides a large TSO size, the obfuscation action may apply smaller one to apply for fine-grained pacing. However, Stob must ensure that it does not generate more aggressive traffic to the network (e.g., higher pacing rate than what CCA desired).

We leave padding policy decision to the application, although its implementation could be done in TLS record padding, because some application protocols have built-in padding features. For example, Jain et. al. [24] proposes web object specific padding scheme. Further, Siby et. al. [46] shows that padding without application-specific knowledge is not effective for WF defenses. Stob's strategy is to allow the application to apply arbitrary packet sequence obfuscation methods using their own padding method and fine-grained packet sequence control, which is impossible with existing systems (§ 2.3).

To see the impact of packet sequence control on performance, we implemented a simple strategy that incrementally reduces TSO size and packet size (reset to the default value once reaching the maximum reduction) over data transmissions in the TCP connection. Figure 3 plots the throughput of a single connection (thus a single CPU core is used in the stack) with `iperf3` over a 100 Gb/s link[2]. As expected, throughput decreases when the packet size or TSO size changes in a wider range (i.e., reduces more aggressively from the default), but it still preserves 19.7 Gb/s or higher. Those results imply relatively low overheads for connections over the Internet access links that are typically at most a few Gbps and shared by many connections.

[2]Between two servers each equipped with two Intel Xeon Gold 5418N CPUs and NVIDIA ConnectX-6 NIC.

5 Discussion

In addition to the stack design challenges that have been discussed, Stob has several implications for the wider Internet architecture.

5.1 Obfuscation Algorithm Designs

Although packet sequence control without increasing the sending rate that the CCA has decided ensures harmless impact to the network, its decision still may conflict with the CCA. For example, BBR uses pacing as a method of measuring queue build up in the network based on resulting ACK intervals. Therefore, packet sequence control may confuse the estimation of the queuing status. Similarly, Copa [3] paces out packets over the window.

To address this challenge, we believe we ultimately need to co-design CCA and packet sequence control. It may be enough to provide interfaces to packet sequence control not to perform any action in certain phases (e.g., BBR's slow start where pacing plays a crucial role), or it may need radical re-design of CCA. It should be noted that CCA specifications usually do not specify exact packet sequence; for example, [57] reports the difference of traffic characteristics between stack implementations that adopt the same CCA. Therefore, some degree of packet sequence adjustment for obfuscation could be possible without violating the CCA specification. Since existing WF defenses based on packet sequence obfuscation have never considered their interplay with congestion control, designing an effective algorithm in this space remains an open question.

5.2 Other Traffic Analysis

Packet sequences are used for other purposes than WF by passive eavesdroppers, either maliciously or not. For example, CCA identification of the flow is a popular network measurement task. Although the previous systems relied on the active probing of the server [33, 18], the state-of-the-art method, CCAnalyzer [54], passively identifies the CCA by classifying the flows based on the bottleneck queue occupancy behavior. Some users may wish to prevent their CCA from being identified, because it potentially reveals other information, such as the OS kernel and application identity.

As other examples, FOAP [29] and AppScanner [52] identify the mobile application from the HTTP/TLS traffic based

on packet sizes and burst behavior. Some monitoring tasks also infer the flow characteristics and thus applications [2].

5.3 Architectural Implications

Schmitt et. al. [43] discusses the *decoupling principles* in the Internet applications. Some of recent systems attempt to generalize or make WF protection deployable, but they create a single locus of observation based on that principle. For example, NetShaper [42] places a middlebox at both client- and server-side to multiplex and demultiplex the obfuscated traffic. In addition to the lack of robust obfuscation (§ 2.3), it creates a single point of observation. In contrast, Stob follows the decoupling principle in terms of not creating such a point by natively supporting WF defense in the host stack.

Utilizing the application-level information in the transport protocol is not a serious violation of the layered Internet architecture principle. The host stack already adjusts packet transmission behavior based on the application-informed policies through `setsockopt`, including `TCP_NODELAY`, which enables immediate transmission without creating a full-sized packet, and `TCP_CORK`, which defers the transmission of data.

5.4 Deployment

Who is participating in defense is an important consideration when designing a WF defense. Most WF defenses designed for Tor assume the collaboration between the client and the middle relay in the Tor circuits constructed by the client [26, 40, 37]. Some WF defenses are designed as client-only solutions [12, 30, 49, 56], but at the expense of signifiant performance penalty, because they need to proactively schedule the sender (i.e., typically the server) using small MSS and advertised window (§ 2).

On the other hand, at the server-side, the defender has more access to the web objects whose features are identified; a defense deployed here is able to perform operations that are not available at the client-side and is thus better placed to offer protection at low overheads. Further, given the widespread use of CDNs, adoption of a server-side protection in one CDN operator would have a significant coverage. Our proposed stack-level support for WF defense is inspired by this observation.

Today's TLS deployment has been accelerated by the server-side activation (e.g., default TLS connection by Google in 2020 and SEO prioritization for TLS-enabled websites), including large operators and small ones with the help of accessible certificate authorities (e.g., Let's Encrypt). Support for WF defense as a part of the data transmission process of the stack is aligned with facilitating such a deployment path.

Operators may be concerned about stack software overheads of having to use small packet sizes or batch sizes and padding, because CDN operators have to serve a large number of clients in a limited space footprint [1]. In addition, existing defenses must incur high bandwidth and latency overheads to be effective, discouraging their widespread adoption [31]. Efficient stack-level packet sequence enforcement (Figure 3) and minimum application-specific padding could address this concern. Furthermore, although the current deployment of WF defense is limited with the exception of Tor [37], we believe Stob-based defense is viable, because the scientific community agrees that even modest defenses can raise the bar for attackers and thus deter censors and eavesdroppers from applying the attacks [31].

5.5 Hardware Offload

Stob relies on a custom packet queuing mechanism, which may hinder its adoption in existing systems that already rely on hardware-based scheduler in commodity NICs. However, we believe emerging NICs are flexible enough to accommodate Stob's need. For example, PIEO [45] implemented in FPGA enables dequeuing an arbitrary packet based on the policy. Other hardware-based packet scheduler designs (e.g., Loom [51]), also trivially implement support for Stob. Packet scheduling in the SoC cores embedded in SmartNICs or DPUs is also common today (e.g., FairNIC [20]).

Stob calls for slightly higher flexibility in TSO. Currently all the packets split by TSO have the same length except for the last one, and the software interface for TSO is designed based on this assumption. If the NIC supports more flexible policies and allows the software to specify arbitrary packet length for each packet, efficiency of Stob would be improved. Since TSO is used by many applications and available in a range of NICs, including those for wireless networks, such flexible TSO behavior could benefit a number of users.

6 Conclusion

Given the growing threat of WF particularly over the client-server encrypted traffic caused by DNS and TLS encryption, we believe it is time to consider practical approaches to protect users from WF. We experimentally demonstrated feasibility of the stack-level WF protection and discussed pathways of extending today's commodity OS to enable it.

Acknowledgments

We are grateful to the anonymous reviewers and Francesco Bronzino, our shepherd, for valuable comments.

References

[1] Joao Taveira Araujo, Lorenzo Saino, Lennert Buytenhek, and Raul Landa. 2018. Balancing on the edge: transport affinity without network state. *USENIX NSDI*.

[2] João Taveira Araújo, Raúl Landa, Richard G Clegg, George Pavlou, and Kensuke Fukuda. 2014. A longitudinal analysis of internet rate limitations. *IEEE Infocom*.

[3] Venkat Arun and Hari Balakrishnan. 2018. Copa: practical Delay-Based congestion control for the internet. *USENIX NSDI*.

[4] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. 2019. Var-cnn: a data-efficient website fingerprinting attack based on deep learning. *PETS*.

[5] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. 2018. Var-cnn: a data-efficient website fingerprinting attack based on deep learning.

[6] Andrea Bittau, Michael Hamburg, Mark Handley, David Mazieres, and Dan Boneh. 2010. The case for ubiquitous transport-level encryption. *USENIX Security*.

[7] Xiang Cai, Rishab Nithyanand, and Rob Johnson. 2014. CS-BuFLO: A congestion sensitive website fingerprinting defense.

[8] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. 2014. A systematic approach to developing and evaluating website fingerprinting defenses. *ACM CCS*.

[9] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. 2010. Side-channel leaks in web applications: a reality today, a challenge tomorrow. *IEEE S&P*.

[10] Heyning Cheng and Ron Avnur. 1998. Traffic Analysis of SSL Encrypted Web Browsing. *Project paper, University of Berkeley*.

[11] Yuchung Cheng and Neal Cardwell. 2016. Making linux tcp fast. *Netdev conference*.

[12] Giovanni Cherubin, Jamie Hayes, and Marc Juárez. 2017. Website fingerprinting defenses at the application layer. *PETS*.

[13] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. 2012. Peek-a-Boo, I still see you: Why efficient traffic analysis countermeasures fail. *IEEE S&P*.

[14] Ellis Fenske and Aaron Johnson. 2024. Bytes to schlep? use a fep: hiding protocol metadata with fully encrypted protocols. *arXiv preprint arXiv:2405.13310*.

[15] Tianyi Gao, Xinshu Ma, Suhas Narreddy, Eugenio Luo, Steven W. D. Chien, and Michio Honda. 2026. Designing transport-level encryption for datacenter networks. *IEEE S&P*.

[16] Jiajun Gong and Tao Wang. 2020. Zero-delay lightweight defenses against website fingerprinting. *USENIX Security*.

[17] Jiajun Gong, Wuqi Zhang, Charles Zhang, and Tao Wang. 2022. Surakav: generating realistic traces for a strong website fingerprinting defense. *IEEE S&P*. IEEE.

[18] Sishuai Gong, Usama Naseer, and Theophilus A Benson. 2020. Inspector gadget: a framework for inferring tcp congestion control algorithms and protocol configurations. *Network Traffic Measurement and Analysis Conference*.

[19] Andy Gospodarek. 2023. Offloading quic encryption to enabled nics. Linux Plumbers Conference, https://lpc.events/event/17/contributions/1592/. (2023).

[20] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. 2020. Smartnic performance isolation with fairnic: programmable networking for the cloud. *ACM SIGCOMM*.

[21] David Hasselquist, Ethan Witwer, August Carlson, Niklas Johansson, and Niklas Carlsson. 2024. Raising the bar: improved fingerprinting attacks and defenses for video streaming traffic. *PETS*.

[22] Jamie Hayes and George Danezis. [n. d.] K-fingerprinting: A Robust Scalable Website Fingerprinting Technique. en.

[23] James K Holland and Nicholas Hopper. 2022. Regulator: a straightforward website fingerprinting defense. *PETS*.

[24] Pranay Jain, Andrew C Reed, and Michael K Reiter. 2024. Near-optimal constrained padding for object retrievals with dependencies. *USENIX Security*.

[25] Marc Juarez, Sadia Afroz, Gunes Acar, Claudia Diaz, and Rachel Greenstadt. 2014. A critical evaluation of website fingerprinting attacks.

[26] Marc Juarez, Mohsen Imani, Mike Perry, Claudia Diaz, and Matthew Wright. 2016. Toward an efficient website fingerprinting defense.

[27] [n. d.] Kernel tls offload. https://www.kernel.org/doc/html/latest/networking/tls-offload.html. ().

[28] Platon Kotzias, Abbas Razaghpanah, Johanna Amann, Kenneth G Paterson, Narseo Vallina-Rodriguez, and Juan Caballero. 2018. Coming of age: a longitudinal study of tls deployment. *ACM IMC*.

[29] Jianfeng Li, Hao Zhou, Shuohan Wu, Xiapu Luo, Ting Wang, Xian Zhan, and Xiaobo Ma. 2022. FOAP: Fine-Grained Open-World android app fingerprinting. *USENIX Security*.

[30] Xiapu Luo, Peng Zhou, Edmond WW Chan, Wenke Lee, Rocky KC Chang, Roberto Perdisci, et al. 2011. Httpos: sealing information leaks with browser-side obfuscation of encrypted flows. *NDSS*.

[31] Nate Mathews, James K Holland, Se Eun Oh, Mohammad Saidur Rahman, Nicholas Hopper, and Matthew Wright. 2023. Sok: a critical evaluation of efficient website fingerprinting defenses. *IEEE S&P*.

[32] Brad Miller et al. 2014. I know why you went to the clinic: risks and realization of https traffic analysis. *PETS*.

[33] Ayush Mishra, Xiangpeng Sun, Atishya Jain, Sameer Pande, Raj Joshi, and Ben Leong. 2019. The great internet tcp congestion control census. *Proc. ACM Meas. Anal. Comput. Syst.*

[34] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. 2021. Defeating DNN-Based traffic analysis systems in Real-Time with blind adversarial perturbations. *USENIX Security*.

[35] Niklas Niere, Felix Lange, Robert Merget, and Juraj Somorovsky. 2025. Transport layer obscurity: circumventing sni censorship on the tls-layer. *IEEE S&P*.

[36] Mike Perry. 2011. Experimental defense for website traffic fingerprinting. Tor Project Blog. https://blog.torproject.org/blog/experimental-defense-website-traffic-fingerprinting. (2011).

[37] Mike Perry. 2015. Padding negotiation. Tor Protocol Specification Proposal. https://github.com/torproject/torspec/blob/main/proposals/254-padding-negotiation.txt. (2015).

[38] Boris Pismenny, Haggai Eran, Aviad Yehezkel, Liran Liss, Adam Morrison, and Dan Tsafrir. 2021. Autonomous nic offloads. *ACM ASPLOS*.

[39] Victor Le Pochat, Tom Van Goethem, Samaneh Tajalizadehkhoob, Maciej Korczyński, and Wouter Joosen. 2018. Tranco: a research-oriented top sites ranking hardened against manipulation. *arXiv preprint arXiv:1806.01156*.

[40] Tobias Pulls and Ethan Witwer. 2023. Maybenot: a framework for traffic analysis defenses. *Proceedings of the 22nd Workshop on Privacy in the Electronic Society*.

[41] Vera Rimmer, Davy Preuveners, Marc Juarez, et al. 2018. Automated website fingerprinting through deep learning.

[42] Amir Sabzi, Rut Vora, Swati Goswami, Margo Seltzer, Mathias Lécuyer, and Aastha Mehta. 2024. NetShaper: a differentially private network Side-Channel mitigation system. *USENIX Security*.

[43] Paul Schmitt, Jana Iyengar, Christopher Wood, and Barath Raghavan. 2022. The decoupling principle: a practical privacy framework. *ACM HotNets*.

[44] Meng Shen, Kexin Ji, Jinhe Wu, Qi Li, Xiangdong Kong, Ke Xu, and Liehuang Zhu. 2024. Real-time website fingerprinting defense via traffic cluster anonymization. *IEEE S&P*.

[45] Vishal Shrivastav. 2019. Fast, scalable, and programmable packet scheduler in hardware. *ACM SIGCOMM*.

[46] Sandra Siby, Ludovic Barman, Christopher Wood, Marwan Fayed, Nick Sullivan, and Carmela Troncoso. 2023. Evaluating practical quic website fingerprinting defenses for the masses. *PETS*.

[47] Sandra Siby, Marc Juarez, Claudia Diaz, Narseo Vallina-Rodriguez, and Carmela Troncoso. 2020. Encrypted dns⇒ privacy? a traffic analysis perspective. *NDSS*.

[48] Payap Sirinam, Mohsen Imani, Marc Juarez, et al. 2018. Deep fingerprinting: undermining website fingerprinting defenses with deep learning.

[49] J Smith, P Mittal, and A Perrig. 2021. Website fingerprinting in the age of quic. *PETS*.

[50] Jean-Pierre Smith, Luca Dolfi, Prateek Mittal, and Adrian Perrig. 2022. Qcsd: a quic client-side website-fingerprinting defence framework. *USENIX Security*.

[51] Brent Stephens, Aditya Akella, and Michael Swift. 2019. Loom: flexible and efficient nic packet scheduling. *USENIX NSDI*.

[52] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. 2016. Appscanner: automatic fingerprinting of smartphone apps from encrypted network traffic. *IEEE EuroSP*.

[53] 1983. The TCP Maximum Segment Size and Related Topics. Request for Comments RFC 879. Retrieved 10/22/2025 from.

[54] Ranysha Ware, Adithya Abraham Philip, Nicholas Hungria, Yash Kothari, Justine Sherry, and Srinivasan Seshan. 2024. Ccanalyzer: an efficient and nearly-passive congestion control classifier. *ACM SIGCOMM*.

[55] Charles V Wright, Scott E Coull, and Fabian Monrose. 2009. Traffic morphing: an efficient defense against statistical traffic analysis. *NDSS*.

[56] Renjie Xie et al. 2024. Cactus: obfuscating bidirectional encrypted tcp traffic at client side. *IEEE Transactions on Information Forensics and Security*.

[57] Gina Yuan, Thea Rossman, and Keith Winstein. 2025. Internet connection splitting: what's old is new again. *USENIX ATC*.

[58] Pengwei Zhan, Liming Wang, and Yi Tang. 2021. Website fingerprinting on early quic traffic. *Computer Networks*.