Journal of Network and Computer Applications 244 (2025) 104368

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Research paper

A protocol-independent in-network security service for cloud applications

Bin Song [a], Bin Sun [a], Qiang Fu [b], Hao Li [a],*

[a] Xi'an Jiaotong University, No. 28 , Xianning West Road, Beilin District, Xi'an, 710049, ShanXi, China
[b] RMIT University, 124 La Trobe Street, Melbourne, 3000, Victoria, Australia

ARTICLE INFO

Keywords:
Programmable data plane
Encryption
TLS

ABSTRACT

Cloud services are increasingly generating a large amount of Internet traffic. Much of it such as rich media and gaming traffic is not highly sensitive but prefers some protection. The traditional end-to-end encryption such as Transport Layer Security Protocol (TLS) is costly and has its own issues, such as increased latency, while the simple anonymity solutions cannot resist traffic analysis attacks. In this paper, we propose FlowShredder, a protocol-independent and in-network service to secure such traffic in the cloud. FlowShredder aims to break the association between the packets, the data flow, and the hosts by obfuscating the packet header (and some payload if needed). Without the context of the flow and the hosts, these packets are of little value to the adversary. The operation is carried out at cloud gateways, without encrypting the payload. Its simple logic can therefore be executed within a single pipeline of the Tofino programmable switch, to ensure wire-speed performance without the scalability issue. Being protocol-independent and operating in-network at wire speed make FlowShredder a practical and generic security service to protect the cloud traffic. In addition, FlowShredder can work with end-to-end encryption such as 0-RTT TLS (*e.g.*, Quick UDP Internet Connections Protocol, QUIC) for enhanced protection, ideal for web browsing or real-time communications. We implement FlowShredder in Programming Protocol-Independent Packet Processors Language (P4) switches. Experiments in synthetic and real scenarios show that FlowShredder can effectively resist the traffic analysis attack with supervised learning techniques, and realize the wire-speed performance over a 100Gbps network while incurring minor overhead.

1. Introduction

In the era of 5G, IoT and edge computing, a large amount of information is transferred in the cloud through the network. This trend has motivated various network optimization efforts, including energy-efficient flow forwarding schemes like EnFlow (Chaudhary and Kumar, 2021), resilient control plane designs such as PARC (Chaudhary and Kumar, 2020), and secure communication mechanisms. Much of the traffic is rich media with large and/or long-lived connections, *e.g.*, live video from the IoT cameras, online gaming and online conference streaming. One major threat to these applications is the traffic analysis attack. The adversary can sniff the traffic from a compromised switch, pick the packets of the end hosts of interest, and reassemble them into a complete flow to obtain the private data, *e.g.*, the video clips and the voice recordings.

Mature techniques, such as TLS (Transport Layer Security) (Anon, 2008) and traffic anonymity (Bromberg et al., 2022; Piotrowska et al., 2017), may overkill or are ill-suited for these scenarios. TLS provides strong protection of the individual packets, but requires extra RTTs (Round-Trip Times) to establish the session key causing increased latency, and consumes a large amount of resources at the end systems. This is particularly an issue for rich media and real-time applications over resource-constrained IoT and mobile devices, which have to establish a large number of sessions either in parallel or in sequence. 0-RTT TLS has been proposed to minimize the latency associated with establishing a secure connection by reusing the established sessions and session keys (Anon, 2018). However, this increases the risk of leaking the key, making the entire flow exposed to the adversary (Aviram et al., 2021). On the other hand, without content encryption, traffic anonymity can only break the association between flows and end hosts (Piotrowska et al., 2017; Berman et al., 2004). That is, the adversary does not know who is talking to whom, but can still reassemble the flow and obtain the full content. Furthermore, these solutions are usually designed for specific protocol stacks, *e.g.*, TCP (Datta et al., 2019), and cannot be easily extended to other transport protocols such as QUIC (Langley et al., 2017).

For applications that do not require strong protection but are sensitive to latency, it is not necessary to encrypt the payload. We argue that the flow is safe as long as its packets are not distinguishable because

* Corresponding author.
E-mail address: hao.li@mail.xjtu.edu.cn (H. Li).

https://doi.org/10.1016/j.jnca.2025.104368
Received 16 February 2025; Received in revised form 22 September 2025; Accepted 18 October 2025
Available online 22 October 2025
1084-8045/© 2025 Published by Elsevier Ltd.

the exposure of a single packet is of little value if the adversary cannot reassemble the flow. Imagine that there are thousands of flows in a network, together with the target flow f. If we stripped the headers of all packets, it would be extremely difficult to identify the packets belonging to f. In short, our insight is that *without the context of the flow and the hosts that the packets belong to, these packets are of little value to the adversary*. In such cases, having strong protection at the cost of latency as TLS does is not necessary. On the other hand, even if strong protection is needed, being able to prevent the adversary from reassembling the flow can significantly enhance the protection. This is particularly true for 0-RTT TLS and QUIC (Kumar and Dezfouli, 2024), commonly adopted for secure web browsing and real-time communications. The goal is therefore to break the association between the packets, the flow, and the hosts.

With this in mind, we propose FlowShredder, which only needs to obfuscate the IP addresses and the L4 header of every packet, without having to encrypt the payload. (In some cases, it may be necessary to encrypt some payload, which may contain metadata and thus reveal the identity of the flow.) Or, the encryption of the payload is left to an end-to-end scheme such as 0-RTT TLS. This makes it possible to take advantage of programmable data planes (Hauser et al., 2023) such as P4 switches and execute the obfuscation operations at wire speed while it has been proved effective in packet processing (Yazdinejad et al., 2020). As a result, FlowShredder is a *protocol-independent* and *in-network* approach operating at *wire speed*. Note that the wire speed is ensured due to the simplicity of the operations, which can be fitted into a single pipeline of the Tofino architecture. Therefore, there is no scalability issue in comparison with the end-host-based approach such as TLS.

FlowShredder does not aim to replace TLS or 0-RTT TLS. Instead, it provides an option for applications such as rich media, which do not require strong protection, in particular those real-time applications. FlowShredder can work with such end-host-based solutions for enhanced protection. Given its nature of being protocol-independent and operating in-network at wire speed, FlowShredder is a practical and generic security service for the cloud. A CSP (Cloud Service Provider) may provide FlowShredder as a service transparent to users among its gateways. Alternatively, cloud users may choose FlowShredder as a service based on the needs of their applications.

To summarize, the growing adoption of latency-sensitive cloud applications highlights critical limitations in existing security solutions. While TLS imposes significant overhead and latency, traditional anonymity approaches remain vulnerable to traffic analysis. This paper addresses these challenges by proposing FlowShredder, an in-network security service that achieves per-packet indistinguishability through protocol-independent header obfuscation. By breaking associations between packets, flows, and hosts without full payload encryption, our approach implemented in programmable data planes maintains wire-speed performance while resisting traffic analysis.

Our contributions are as follows.

- We propose FlowShredder for per-packet indistinguishability (Section 3). Without the context of the flow and the hosts, the packets are of little value to the adversary. The per-packet indistinguishability is achieved through packet header obfuscation using a per-packet random key and the lightweight `XOR` operation. This ensures the randomness of each packet, and the random key is only valid for a single packet. We take advantage of IPv6 to route the packet correctly.
- We leverage the programmable data plane and implement FlowShredder with hardware P4 switches FlowShredder's logic is programmed into a single pipeline of the Tofino switch to ensure its line-rate processing ability. The random key is encrypted by 2EM, a simple yet effective scheme.
- FlowShredder is evaluated with real and synthetic configurations (Section 6). The results suggest that (1) FlowShredder is effective against the traffic analysis attack even with supervised learning; (2) FlowShredder is transparent to the end hosts, and outperforms the existing in-network and end-to-end approaches; and (3) FlowShredder can achieve the wire-speed performance.

2. Threat model and theoretical base

In this section, we first demonstrate the threat model of the traffic analysis attack. We then discuss the security strength of per-packet indistinguishability.

2.1. Threat model

In our model, a trusted CSP refers to a cloud provider whose infrastructure, including gateway switches and internal networks, is fully controlled by a trusted entity, such as colleges, companies or private households. Conversely, an untrusted CSP encompasses all external networks and devices where adversaries may operate, including third-party clouds or public internet segments.

Fig. 1 shows a typical scenario of traffic leaking. The end hosts are connected to a trusted CSP gateway (yellow switches) through a trusted CSP network (yellow cloud). The trusted networks are connected through the untrusted networks (gray cloud).

All devices not in the trusted CSP networks are prone to the attacks. We assume the adversaries work in a passive model, meaning that they cannot replay or modify the traffic to launch a replay-attack or Man-In-The-Middle attack.

The adversaries can compromise the untrusted switch (es), and sniff and record all traffic from it such that once the private key of the end host is lost in the future, they can use it to decipher the pre-collected traffic if encrypted with the same key. The adversaries can compromise some end hosts, *e.g.*, C_1 and C_2, along with the untrusted switch S, such that they can launch a chosen-plaintext attack by comparing the information sent from C_1 to C_2 with the information sniffed in S to identify the obfuscation scheme. This scenario is common for public clouds, where any customers (including adversaries) can buy virtual machines behind the trusted CSP network. The adversaries can then find a way to reassemble the data flow and discover useful information.

2.2. Security strength with per-packet indistinguishability

Existing work relies on content encryption for protection, while we argue that the connection is secure if per-packet indistinguishability is ensured. That is, the adversary cannot establish the association between the packets, the connection, and the hosts with the sniffed traffic. We now analyze why this argument may hold in practice.

First of all, the per-packet indistinguishability ensures the unobservability of the protected end hosts. That is, the adversary does not know whether the target end hosts are online, and cannot simply filter out the content of interest. Instead, the adversary has to examine *all* the sniffed traffic. Now consider there are P packets concurrently being transferred in the network, and the connection to be protected consists of p packets. If the adversary cannot associate any packets together, in the worst case, the adversary has to try A_P^p (A means arrangement) times to pick out and order the packets of the target connection. In practice, only a few packets are out-of-order, so the actual complexity approaches to C_P^p, which is still an unacceptable overhead for the adversary. For example, to distinguish 100 target packets from 100K packets, an adversary will have to query $C_{100,000}^{100} \approx 10^{340}$ times, which may take more than 40 years even with the current top-1 supercomputer (Anon, 2022a). In real networks, P and p are usually large enough to provide high-security strength. For p, a 10 s video clip could have tens of megabytes, creating thousands of packets. For P, in a 10Gbps network, there could be ~8M concurrent 1500B-packets at the peak time during the period.

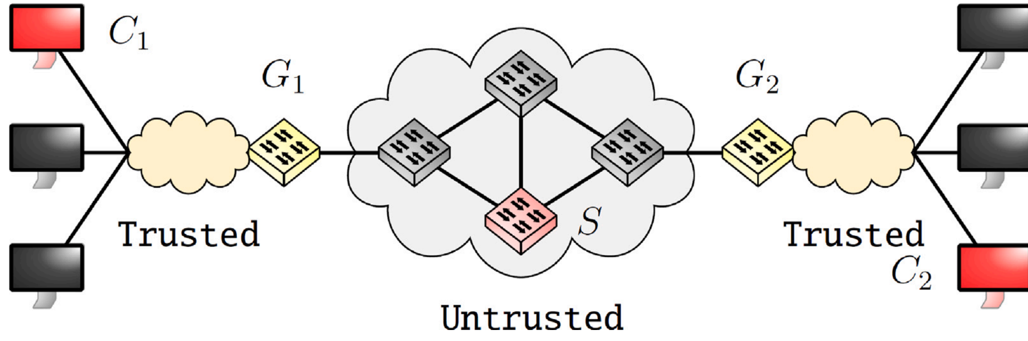


Fig. 1. Threat model. End hosts are firstly connected to a trusted CSP gateway through a trusted CSP network (yellow cloud and switches). Then the two trusted CSP networks are connected through a large untrusted network (gray cloud and switches). Adversaries can compromise the end hosts and untrusted switches to sniff the traffic (red hosts and switches).

3. Design of FlowShredder

An intuitive approach to realizing per-packet indistinguishability is to obfuscate the identifiers of the connection, *e.g.*, IP addresses and TCP sequence numbers. The difference between such obfuscation from the anonymity approach is that FlowShredder needs to break the association not only between the connection and the hosts but also between the packets and the connection, making *every packet* indistinguishable, that is, achieving per-packet indistinguishability. This prevents the adversary from reassembling the connection with sniffed packets. However, header obfuscation messes up the destination IP, leading to incorrect routing. We need to find a way to route the packet correctly. The procedure to construct a packet in FlowShredder is shown in Fig. 2.

3.1. Efficient obfuscation

As shown in Fig. 1, FlowShredder can be deployed in the gateway switch, *i.e.*, G_1 and G_2. It uses a per-packet random seed, *i.e.*, an obfuscation key, to XOR the header fields. In doing so, each packet is obfuscated in a different way using the lightweight XOR operation, and the random key is only valid for a particular packet. The leak of the key only affects a single packet. However, there are some issues. First, the obfuscation key used by G_1 must be sent along with the packets, otherwise G_2 cannot recover the packets from the obfuscation. This means that the key must be strongly encrypted, *e.g.*, with AES, which may consume significant resources (Chen, 2020). Second, even if we can assume that all packets are with IP protocol, we still have to obfuscate many other fields to support the L4 protocol independence. For example, a TCP packet requires obfuscating the TCP ports and sequence numbers, while a QUIC packet has to obfuscate the stream and connection IDs. There are also combined stacks like IP-in-IP that require complex obfuscation. As a result, the bytes to be obfuscated could be too long to fit into a single pipeline of the P4 switch. FlowShredder addresses these challenges as follows. First, it uses a 64-bit obfuscation key to XOR the IP addresses and *all headers hereafter*. This ensures that all L4 protocols can be supported by FlowShredder, and XOR is a lightweight operation. To secure the obfuscation key, instead of AES, FlowShredder uses a two-round Even-Mansour (2EM) scheme (Bogdanov et al., 2012) to encrypt the obfuscation key for its simplicity and tight security proofs. The cipher encrypts a n-bit text M by computing:

$$E(M) = P_2(P_1(M \oplus k_0) \oplus k_1) \oplus k_2 \quad (1)$$

where k_0, k_1 and k_2 are independent encryption keys and P_0, P_1 and P_2 are independent permutations (Wang et al., 2021).

Theoretically, 2EM is secure against about $2^{\frac{2n}{3}}$ queries with chosen-plaintext attacks (Barrera et al., 2010), *i.e.*, about 2.6M million queries for 32 bit message encryption (Wang et al., 2021). As we encrypt a 64-bit message, the number of queries will be up to 7 trillion, which should be sufficient to resist such attacks. Besides, FlowShredder rotates k_0–k_2 from time to time, making it even harder for an adversary to break. The source and destination IP and L4 headers are obfuscated using a per-packet random key, which is encrypted by a 2EM algorithm with a rotated key set. There is no need to share the random key between the trusted gateways because the gateway can recover the random key via the 2EM key set. Even though this requires global 2EM key management, the cost can be managed by manipulating the rotating period of the 2EM key set.

3.2. Facilitating routing procedure

The above random obfuscation may break the end-to-end routing procedure for two reasons. First, the obfuscated destination IPs are unknown to the routers in untrusted networks, so the packets cannot be correctly routed. Second, since the encrypted obfuscation key is sent along with the packet as a header field, we need to ensure such information will not impact the correct routing.

FlowShredder works natively with IPv6 but needs to transform IPv4 packets into IPv6 format in the entrance gateway and recover them back to IPv4 in the exit switch. The IPv6 format gives the flexibility that FlowShredder can encode the encrypted key into the 128-bit address fields. Specifically, FlowShredder constructs a new IPv6 packet containing a configured address field. The IPv6 source address field (128b) is partitioned into three parts, a random source address (32b); the obfuscation key in ciphertext (64b); and the encrypted original IPv4 source address (32b). The IPv6 destination address also comprises three parts: a destination address randomly selected from an address pool (32b), the encrypted original IPv4 destination address, and 64-bit reserved padding field for protocol extensions. FlowShredder announces the prefixes of the address pool to the routers in the untrusted networks. The addresses in the pool are randomly generated and rotated in a way to avoid frequent route convergence in the untrusted network. The L4 headers and the payload remain the same as the original IPv4 packets.

In doing so, the intermediate network can correctly route the packets from G_1 to G_2, since the destination address from the pool points to G_2. G_2 recovers the IPv4 packets by reversing the operations.

At the same time, such behavior could increase the transport cost and reduce the payload efficiency. We will analysis it in Section 6.4.2.

4. Improving the per-packet indistinguishability

The burst transferring breaks the indistinguishability even with per-packet random obfuscation, since the continuous packets could facilitate the connection reassembly. In this case, the indistinguishability is solely determined by such continuity. In the next, we first measure the continuity (Section 4.1), and then break such continuity to improve the indistinguishability (Section 4.2).

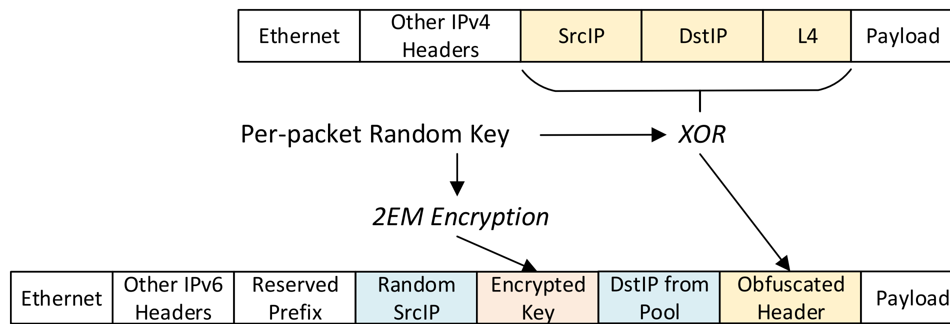


Fig. 2. The encryption of headers and the construction of packets in FlowShredder. The source and destination IP and L4 headers are obfuscated using a random per-packet key. Then, such a key is encrypted by the 2EM algorithm with a rotated key set. Finally, the obfuscated fields and the encrypted key are embedded into an IPv6 packet. The exit gateway performs the reverse process to recover the original packet.

4.1. Measuring the continuity

To quantify the continuity for a specific piece of traffic, we employ a simple metric, *i.e.*, coefficient of variation, to quantify the packet continuity, as defined below.

$$c_v(P) = \frac{\sigma(P)}{\mu(P)} \quad (2)$$

where σ and μ are standard deviation and average value of the indexes from the target connections in a continuous sampled traffic P. That is if the target packets in P have a larger dispersion, the indistinguishability is better.

Note that P is usually not the whole traffic, since the gap between bursts of the target connection could be very large. In that case, those two bursts can be viewed as irrelevant. In practice, we can slice the traffic containing the target connection with a fixed number, say 100, of packets, calculate c_v for each of the slices, and measure the continuity using the average c_v.

4.2. Reordering the packets to break continuity

We reorder the packets to break the connection continuity. Programmable switches often have very limited packet buffers, so it is not feasible to arbitrarily reorder the packets. Moreover, the random reordering may break the nature packet order of the same connection, resulting in lots of out-of-order packets for the TCP stack at the end host, which is an unacceptable overhead.

The basic idea that FlowShredder overcomes this problem is to exploit the existing traffic queue in the programmable switch. Specifically, FlowShredder sets multiple queues, each of which has the same priority. Incoming packets are pushed into different queues by hashing their 5-tuple (source IP, source port, destination IP, destination port, and Layer 4 protocol) at wire-speed. Thus, even if the target connection's packets arrive continuously, they must compete with other queues for transmission. As a result, the packets are reordered by cover traffic (i.e., non-encrypted traffic and other target connections) due to their stream-indistinguishable IPv6 headers, though the original packet sequence remains intact.

However, such reordering would only be effective when enough amount of concurrent packets are transferred in the network, *i.e.*, the switch must be somehow congested, such that the queues can have enough packets to compete for the out port. Otherwise, packets pushed to different queues would be directly sent out without queuing. FlowShredder deliberately creates the cover traffic in case the packets are not queued in the switch, but needs to address the following challenges.

First, FlowShredder is an in-network approach, meaning that there is no other end host we can use to generate the cover traffic. Second, the cover traffic must not be too large, or the original traffic would be dropped, and the cover traffic should not be too small, or it cannot generate enough congestion in the switch. Third, the cover traffic may slow down the transmission of original traffic, enlarging the flow completion time. FlowShredder adopts the following designs to address these challenges.

4.2.1. Generating and buffering the cover traffic

Instead of employing a dedicated traffic generator, FlowShredder generates the cover traffic from the original traffic. Specifically, FlowShredder samples the original traffic as it goes through the switch. However, the sampled traffic should not be directly sent out, because (1) the sampled traffic is largely the same as the original traffic, which may give the adversary clues on how to distinguish the cover traffic and original traffic; and (2) the sampled traffic may cause severe congestion in the switch, resulting in the packet loss of original traffic.

To this end, FlowShredder "buffers" the sampled cover traffic by looping it in a dedicated switch port, *i.e.*, the recirculation port. Those packets are tagged as cover traffic, while the tags are also encrypted in the IPv6 header. In the exit gateway, that traffic would be identified and directly dropped. Note that this would drain all the bandwidth of that port, but it would not impact the processing of the normal in ports and the out port.

4.2.2. Detecting the queue congestion

The next task is to detect the congestion condition of the queues, to determine when and where to send the cover traffic. The off-the-shelf switch, *e.g.*, P4 Tofino, provides a way of probing the congestion condition, but it can only detect the queue of the current packet.

To this end, we detect and relay the congestion condition by the following steps. First, we set a register to store the congestion state of all queues for the out port. Then, each packet sent to the out port would probe the congestion condition of its queue, and update such information into the register. Next, the recirculating packets, *i.e.*, the sampled cover traffic, would read and carry the register with its header. Finally, for each recirculating packet looping back to the switch, FlowShredder reads the information from its header, and if (1) all queues are already somehow congested, FlowShredder just recirculates the packet, so that the original traffic is not impacted; (2) some queues are not congested, FlowShredder would clone the current packet, and send it to one of the idle queues. Since we have "infinite" recirculating packets, all queues will eventually get congested.

Fig. 3 illustrates the above process. Consider two concurrent connections, yellow and green, transferring in the network. The packets of the yellow connection are continuous. The red packets are the pre-sampled cover traffic from other connections, looping in the recirculating port. The incoming packets are firstly classified into different queues by their connections, *i.e.*, $Q1$ for the yellow connection, and $Q2$ for the green connection. Currently, since $Q2$ has only one green packet, it cannot effectively break the continuity of the packets in $Q1$. In this case, the cover packets would read the congestion register and find that $Q2$ does not have enough packets. Then, the cover packet will be cloned and sent to the spare queue, *i.e.*, $Q2$.

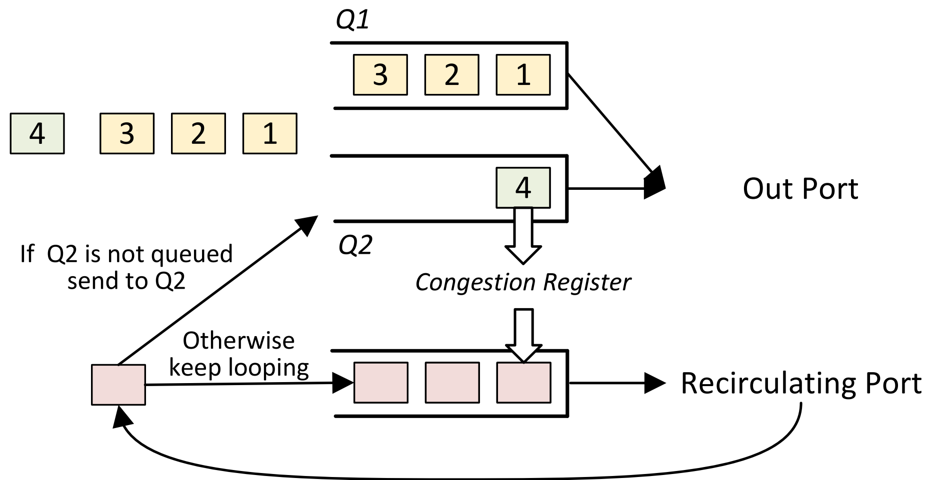


Fig. 3. Breaking the continuity by deliberately sending the cover traffic. The incoming packets are classified into different queues, and will probe the congestion condition and write to the congestion register. Then, the sampled cover traffic (red packets) will read such information, and if the corresponding queue is not sufficiently congested (*e.g.*, $Q2$ for the green connection), the cover packet will be cloned and sent to it. As a result, $Q1$ and $Q2$ will always have enough packets in the queue to compete with each other, such that packets from the same connection will not be continuous.

4.2.3. Trading off the latency for the indistinguishability

It is a hard trade-off between the indistinguishability and the latency, *i.e.*, flow completion time (FCT), when the queues are not congested. More cover traffic would largely split the burst, while it also burdens the untrusted network and enlarges the flow completion time of the target connections.

In practice, such trade-off can be controlled by the number of queues: The target connections would compete with more packets if more queues are set. Theoretically, given a system with K queues each with a congestion length N, a newly enqueued packet can expect to be transmitted after $K \times N$ forwarding cycles on average. As the number of queues K increases while the congestion queue length N remains constant, end-to-end latency scales linearly with K, establishing a direct proportional relationship. Therefore, if we set 32 queues, and all queues are congested, the packets of the target connections would be delayed for 32×.

However, we argue that the real scenario would not be that severe, because such a delay would only impact the FCT for the *last burst* of the connection. Consider a connection with N bursts, each of which has n packets, and the gap between each burst is g. Now assume we set q queues to create the congestion, then *for each burst*, its completion time would largely increase, *i.e.*, it needs $q \times n$ packets to finish the transfer of a single burst. However, g is usually quite larger than $q \times n$, considering there are many concurrent connections. In other words, although each burst is delayed, it will not delay the next burst. Only the last burst would impact the FCT, which would increase to $\frac{N-1+q}{N}$. In practice, N would easily exceed several hundred. For example, to transfer a 10MB file, it would use 200 (number of bursts) × 32 (number of packets in a burst) × 1500 bytes. Consider the worst case, *i.e.*, g equals to $q \times n$, the FCT would only increase 3.5% with 8 queues.

FlowShredder further mitigates this problem by dynamically adjusting the number of active queues. Initially, FlowShredder sets up all queues for the out port, while if the FCT is not satisfied by the user, FlowShredder can de-activate some queues by modifying the packet hashing process without halting the packet forwarding.

5. Implementation

We implement FlowShredder with 1500 lines of $P4_{16}$ code and deploy it on hardware P4 switches equipped with Barefoot Tofino chips. There are multiple pipelines in Tofino chips, and each pipeline is filled with 2 processes, Ingress and Egress. As Fig. 4 showed, when FlowShredder receives a packet, the packet first goes into the Ingress Parser where FlowShredder parses the packet headers, decrypts it if necessary in the Decryption Part in Ingress Control, routes the packet to correct egress port, then repacks the packet in the Ingress Deparser and sends it to traffic manager. The traffic manager would forward the packet to the correct Egress pipeline. In the pipeline of the corresponding egress port, FlowShredder will parse the packet first in the Egress Parser. In the Egress Control block, FlowShredder uses P4 random function to generate a 64-bit obfuscation key, replaces every byte of the obfuscation key with an 8-bit Substitution-box (S-box) and disrupts the output of the S-box by Permutation-box (P-box), then FlowShredder XORs the necessary original header information with the obfuscation key (Encryption block in the diagram). In the Egress Deparser, FlowShredder constructs a new IPv6 header to replace the original IPv4 header, with the configured address domain, then sends it to the forwarding queues. At the receiver, FlowShredder reverses the above packet construction steps and restores the original IPv4 packet.

In Tofino's design, Ingress and Egress have 12 stages each, and a stage will be assigned as an empty stage if no operation is assigned. An assigned stage costs more time than an empty stage, so Tofino running FlowShredder is less efficient than Tofino simply forwarding packets. But due to the pipeline design, Tofino running FlowShredder only increases the forwarding delay and matches the throughput of pure packet forwarding in Tofino, as long as the buffer can store all packets received during the processing of a single packet. In our case, we have used 10 stages in both Ingress and Egress to realize the function that we need, including encryption, decryption and routing. For storing, we only need to store the 2em cipher key which is 32 bits in most cases and 64 bits for higher security requirements. Other information, such as obfuscation key and original IPv4 addresses, are stored in the received packet's memory space.

2EM encryption. We use the Random external function to generate a 64-bit obfuscation key. To encrypt this obfuscation key via the 2EM scheme, we use 8-bit substitution boxes (S-boxes) to substitute every byte of the obfuscation key, and then use the permutation box (P-box) to shuffle the S-box output. As the definitions of these S-boxes are similar to each other, we use a macro definition to generate the S-boxes.

Operations fitting into a single pipeline. In Tofino architecture, a pipeline is a sequence of programmable operations that a packet has to go through. In a single pipeline, a packet will go through 6 different modules, including Ingress parser, Ingress match-action pipeline, Ingress deparser, Egress parser, Egress match-action pipeline, and Egress deparser. Programmers can define the operation logic in these modules. However, due to limited hardware resources, the P4

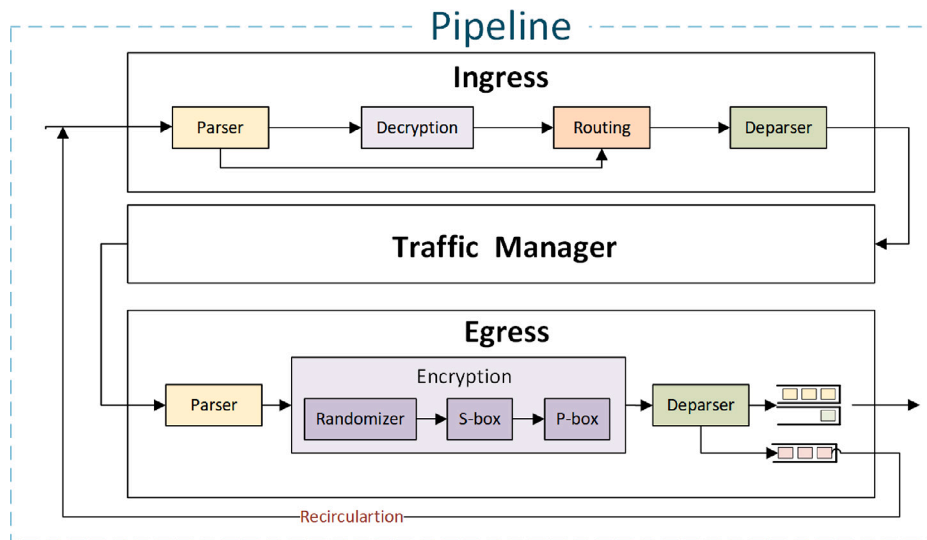


Fig. 4. FlowShredder Flowchart.

logic may not be able to fit into a single pipeline. In this case, the packet has to recirculate through the pipeline round by round until the logic is completed. Recirculating will increase the processing latency and decrease the throughput. FlowShredder performs a number of operations (*e.g.*, XOR and 2EM encryption) on the obfuscation key, which requires the limited Packet Header Vector (PHV) resources (Anon, 2021), obstructing its logic being fitted into a single pipeline. We address this problem by separating the encryption and decryption processes into the Ingress and Egress procedure. In the Ingress pipeline, FlowShredder processes the encrypted packets, *i.e.*, decrypting the packet and deciding how to route the packet. In this process, the Ingress deparser emits the decrypted packet to the Egress parser. In the Egress pipeline, FlowShredder decides whether the packet needs to be encrypted. If so, FlowShredder will generate a random key, encrypt it, and then emit the packet.

In this implementation, FlowShredder only obfuscates the connection identifier (L3/4). In some cases, it is necessary to obfuscate some payload that may reveal the identity of the flow. One example is some special metadata present in rich media traffic such as sequence numbers and timestamps in the RTP protocol.

FlowShredder is currently implemented on Tofino switches. It can also be implemented on other P4 switches or PISA/PSA switches in general. The switch operations required include packet header parsing, XOR operation, 2EM encryption (including flow table matching and shifting operation) and simple forwarding operations. These operations are commonly supported among PISA/PSA switches (for example, P4 DPDK and BMv2).

6. Evaluation

In this section, we evaluate the effectiveness and performance of FlowShredder. We are particularly interested in answering the following questions:

- Is FlowShredder safe enough such that the adversaries cannot obtain the valuable content of the target connections? Section 6.2 shows that (1) by obfuscating the packet headers, the adversaries cannot reassemble the original content, even using some clustering technique, and (2) by reordering the packets, the continuity of the connection is largely broken, and the adversaries cannot reassemble the connection by simply concatenating the continuous packets.
- Is FlowShredder fast enough as an in-network approach, that is, realizing the wire-speed encryption without end host intervention? Section 6.3 deploys FlowShredder in the local testbed and public cloud. The results demonstrates that FlowShredder is 5.5× faster than existing in-network encryption approach (P4-AES), and 1.1× more CPU-efficient than the de facto end-to-end encryption approach (TLS). Moreover, FlowShredder can always catch up with the wire speed, *i.e.*, the highest throughput the traffic generator or end host can achieve.
- Does FlowShredder trade off too much for the security, *i.e.*, will the packet reordering and cover traffic sending impact the performance of the target connections? Section 6.4 shows that FlowShredder only brings 2.2% extra flow completion time under real scenarios, while TLS would encounter more than 40% of such overhead.

6.1. Testbed setup

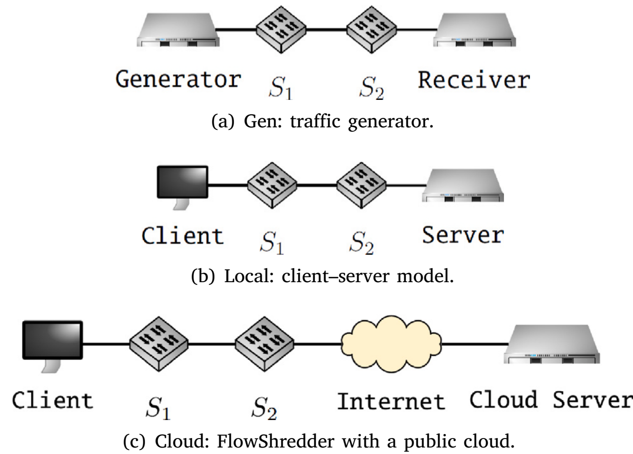
We deploy FlowShredder in hardware P4 switches along with the local end hosts and the public clouds. Each P4 switch is a Wedge 100BF-32X switch with a Tofino programmable chip (Anon, 2022). Without otherwise specified, FlowShredder uses 8 queues to break the continuity. We build three types of testbeds, as shown below.

Traffic generator (Gen). As shown in Fig. 5(a), Gen uses a traffic generator to generate the traffic, all of which should be protected. Those traffic will traverse the entrance and exit gateway sequentially and be received by the end host. The generator and the receiver are equipped with 100Gbps NIC, in order to drain the link and test the upper bound of the throughput.

Local testbed (Local). As shown in Fig. 5(b), Local sets a pair of end hosts between the gateway switches. We use Nginx to build a simple HTTP/HTTPs server, such that this testbed can test whether FlowShredder performs well on the real connections with congestion control schemes. The bandwidth between the client and the server is 10Gbps.

Public cloud (Cloud). As shown in Fig. 5(c), we also rent five cloud severs (C1–C5) from all over the world.[1] By communicating with the

[1] We omit the locations of the servers as they may reveal the country of the authors.



(a) Gen: traffic generator.

(b) Local: client–server model.

(c) Cloud: FlowShredder with a public cloud.

Fig. 5. Three types of testbeds for evaluating FlowShredder.

cloud servers, we can use Cloud to test the performance of FlowShredder under complex public environments. The bandwidth of the cloud server is 100Mbps.

The local machines are with Intel 16-core 2.5 GHz CPU and 16G memory, and the cloud servers are with 2-core 2.5 GHz vCPU and 1G memory. All servers run CentOS 8 OS.

6.2. Security

We evaluate the security level of FlowShredder from two perspectives, *i.e.*, whether the per-packet indistinguishability is effective to resist the traffic analysis attack, and whether the packet reordering technique proposed by FlowShredder can break the packet continuity.

6.2.1. Security from per-packet indistinguishability

The major insight of FlowShredder is that the per-packet indistinguishability could be sufficient to secure the large rich media files, since the content of a single packet is of minor value to the privacy.

We leverage the classic convolution neural network (CNN) and ResNet, to emulate the chosen-plaintext attack. We assume that the adversary knows the file format of the target connection, and is able to rent the servers in the trusted network. The adversaries can then generate their own connections, ideally along with the target connections, and train the CNN and ResNet model with supervised learning. To mimic this process, we prepare 100 different files of the same format (`.mp4`) with various sizes. We use Gen testbed, and transfer all 100 files. The test set and the training set are derived from transfer sampling of the same files. We take ∼90% of the packets in the sampled data as the training set, and the rest as the test set. The input of the models is the packet header with some payload. As the input length grows, more of the payload is added to the input. This is to see how the payload affects the detection accuracy.

Fig. 6 shows the testing accuracy in CNN and ResNET, when the test set and the training set are derived from transfer sampling of the same files. The share of the background and target traffic is 1:1. That is, the accuracy of a random guess is 50%. Both CNN and ResNet have a high accuracy (nearly ∼100%), when dealing with the raw traffic, indicating that the traffic analysis with supervised learning is quite effective. The only exception is when the training input length is 25B, as it only covers the first two header fields in Fig. 2, which have no valid information for detection. The accuracy is essentially a random guess, ∼50%. It is a different story for FlowShredder. The accuracy is slightly better than a random guess. This indicates that the knowledge learned from the sample traffic is not quite meaningful for the testing traffic, as the headers of all packets are completely random. ResNet outperforms CNN when the payload is used for training, indicating that ResNet seems to have learned something from the payload.

This experiment shows that packet header obfuscation is an effective way against the traffic analysis attack, by achieving per-packet indistinguishability. Otherwise, even without knowing the syntax or semantics of the header, the supervised learning can easily reassemble the connections.

6.2.2. Breaking the packet continuity

With the per-packet obfuscation, the adversary is not capable to cluster the packets for the same connections by checking the header and content. However, the burst nature of connection could facilitate the connection reassembly, *i.e.*, the adversary can simply capture a sequence of packets, and the majority of the packets belong to a single connection. FlowShredder reorders the packets to break such continuity.

Specifically, we use Local testbed, and retrieve a large file with different rates. We capture the traffic and measure the continuity with the coefficient of variation (*i.e.*, c_v) mentioned in Section 4.1. The results in Fig. 7 depict two trends of c_v. First, more queues generally result in better indistinguishability, *i.e.*, large coefficient of variation. For example, 8 queues can increase the average c_v by 3.9× while the rate of 4 queues is around 2×. However, when the number of queues increase to 16, c_v becomes stable. This is because such case requires too many cover traffic to congest the queues. In practice, 8 queues are sufficient to make enough disordered packets.

The second trend from Fig. 7 is that when the input rate increases, the effect of reordering decreases. This is reasonable, because Local only sends *a single* connection as the target connection, and if the target connection dominates the traffic, the cover traffic could not make enough obfuscation. In our experiment, the wire speed of Local is 10Gbps, meaning that when the input throughput is 4Gbps, 4 out of 10 packets are from target connection, which explains the low increase rate of c_v (∼ 1.3) in such case. We again emphasize that this only happens when *a single* target connection dominates the traffic. In real scenarios, a single target connection is usually quite small, and multiple target connections can cover each other.

6.3. Performance

We evaluate the performance of FlowShredder from two perspectives: whether FlowShredder can outperform the existing in-network approaches and whether FlowShredder can catch up with the wire speed.

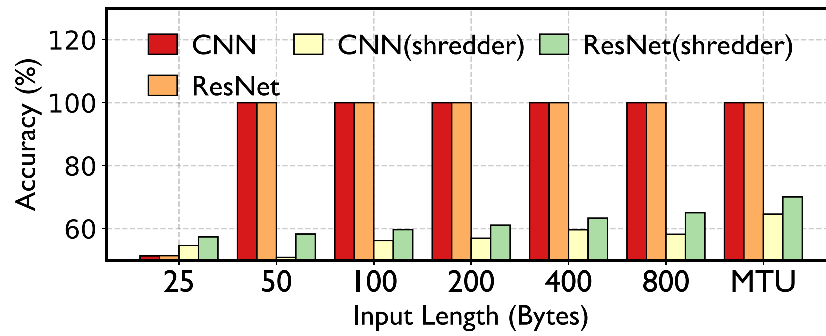


Fig. 6. Known cover traffic can improve accuracy for original packets, especially when input size become longer. However, known cover traffic cannot improve enough accuracy of FlowShredder (The maximum accuracy for FlowShredder is ~70%).

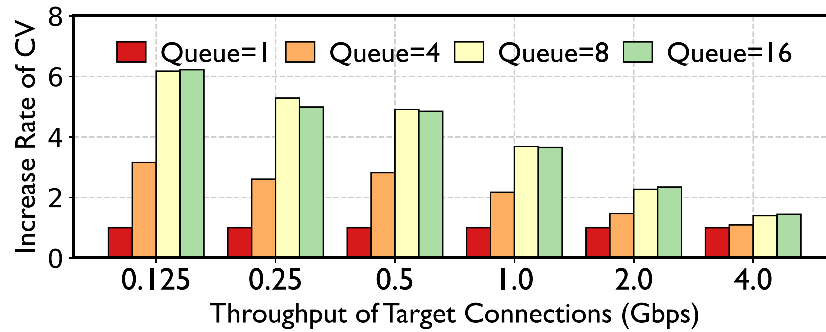


Fig. 7. The coefficient of variation of FlowShredder with different number of queues and input rates. More queues generally lead to better indistinguishability, which however is also impacted by the input rates.

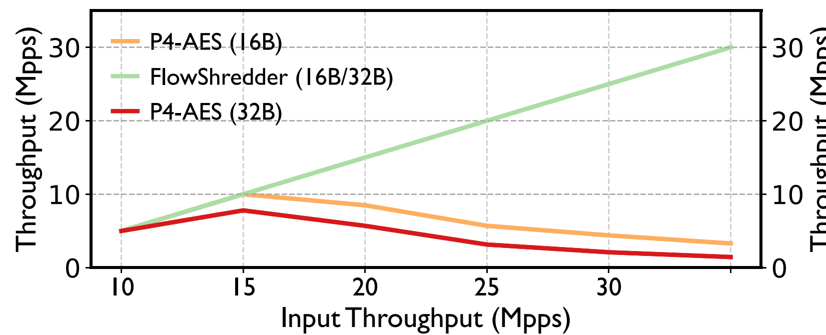


Fig. 8. Comparison with P4-AES over different size of packets. Curves of FlowShredder (16B) and FlowShredder (32B) are completely overlapped, since the performance of FlowShredder is irrelevant to the packet size.

6.3.1. Comparing with P4-AES

P4-AES (Chen, 2020) is an in-network encryption approach that leverages Tofino chip to encrypt the packet header or payload. This approach can be used to offload the encryption task at the end hosts, so as to reduce the end-to-end overhead. We compare FlowShredder with P4-AES to show the benefit of the encryption from per-packet indistinguishability.

We use Gen testbed, where the generator sends fix-sized UDP packets to the receiver, and the P4 switches will encrypt and decrypt every packet in sequential: for FlowShredder, S_1 obfuscates the headers and reorder the packets by creating cover traffic; for P4-AES, S_1 encrypts the fixed-sized payload. S_2 will perform the reverse operations accordingly.

Fig. 8 measures the throughput of FlowShredder and P4-AES by transferring UDP packets with 16- and 32-byte payload. It can be seen that P4-AES suffers from a significant penalty as the packet size and/or input rate increases. For example, it can only achieve 3.3Mpps for 16-byte payload and 1.45Mpps for 32-byte payload, at the input rate of 30Mpps. The root cause is that P4-AES will recirculate the packets for many rounds to encrypt the payload, *i.e.*, 10 rounds for 16-byte encryption and 20 rounds for 32-byte. Larger packets lead to more recirculation rounds, and higher input rate increases the possibility to drop the packet in the recirculation. In the worst case, the packet is eventually dropped before finishing the recirculation, and the output throughput would drop to zero. In contrast, the performance of FlowShredder is irrelevant to the packet size, which always catches up with the input rate. The reason is that FlowShredder only obfuscates the packet header, and the cover traffic would only generate proper congestion in the queues, and will not drop the original packets.

6.3.2. Comparing with TLS

HTTPS/TLS is the de facto end-to-end encryption approach, which however might consume many more CPU cycles for the task of asymmetric (*e.g.*, ECDHE) and symmetric (*e.g.*, AES) encryption. We compare FlowShredder with HTTPS/TLS approach to reveal the benefit of the in-network approach.

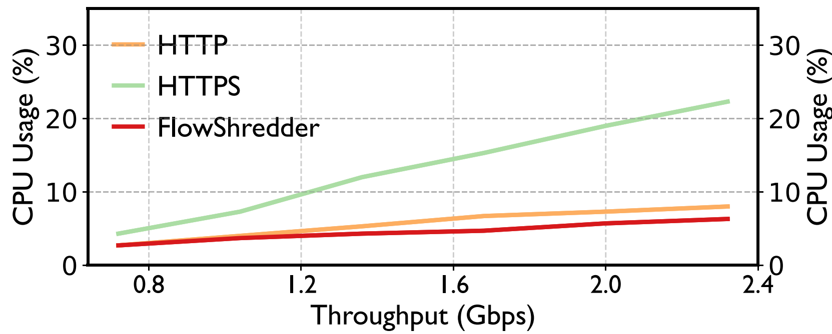


Fig. 9. The CPU usage of the server side. HTTPS/TLS approach consumes averagely 1.1× more CPU cycles than non-encryption case, while FlowShredder saves 15% CPU because of the larger FCT.

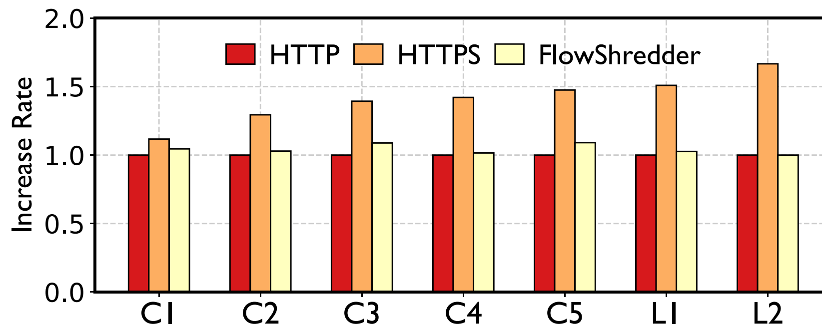


Fig. 10. The increase rate of flow completion time (FCT). FlowShredder only adds ∼2% FCT while HTTPS/TLS increases FCT by 40%.

We use Local testbed, where the client uses 8 processes to retrieve the 1MB file from the server at various rates. We measure the average CPU usage of the server during the transmission. Fig. 9 shows that HTTPS/TLS consumes 1.1× more CPU cycles than the original HTTP transfer, largely because the server is busy encrypting and decrypting the key and the payload with its CPU. In contrast, FlowShredder saves averagely 15% CPU cycles compared to the non-encryption case. This is because FlowShredder slightly enlarges the flow completion time, which means in a fixed period, fewer packets would arrive at the server, and hence the lower CPU usage. (See Section 6.4 for the details of such overhead.)

6.4. Overhead

In this part, we evaluate the overhead of FlowShredder for two perspectives, computation overhead and communication cost.

6.4.1. Computation overhead: Flow completion time

One major trade-off FlowShredder has made is that it may slightly increase the flow completion time (FCT) of the target connections, because they have to be queued and compete with the cover traffic. We quantify such overhead by comparing the FCT of FlowShredder with TLS.

We use Cloud testbed, and let the client use 8 processes to keep retrieving a 1MB file from the cloud servers (C1–C5) without any rate limit. Due to the various network condition, we cannot drain the 100Mbps bandwidth provided by the cloud server. We further use Local testbed, and let the client retrieve a small file (1MB, L1) and a large file (5 GB, L2) from the server. Each experiment is conducted for 10 times and we report the average FCT with TLS or FlowShredder.

Fig. 10 shows that TLS averagely adds 41% of FCT while FlowShredder only increases such rate by 2.2%. The key reason to this result is that TLS would cost extra RTT to exchange keys, enlarging the FCT, especially for the small- and median-sized connections. Transferring larger files might hide such overhead, but the encryption and decryption of the large payload would in turn increase the FCT. In contrast, FlowShredder's overhead comes from the packet reordering, which might delay the *last* burst of the connection. As briefly discussed in Section 4.2, such overhead is usually minor, and would become negligible for larger files.

Table 1
Transmission cost.

Scenario	TCP		UDP	
	L6 payload (Byte)	Scale	L6 payload (Byte)	Scale
Normal IPv4	1460	97.33	1472	98.13
IPv6 tunneling	1412	94.67	1424	95.47
NAT64	1432	96.00	1452	96.80

6.4.2. Transmission overhead: payload efficiency

FlowShredder receives an IPv4 packet then forwards an IPv6 packet, such behavior could increase the transport cost and reduce the payload efficiency. In rich media applications, the packet size is usually large, reaching the maximum transmission unit (MTU) of 1500 bytes, which means 1460 bytes of Layer 4 payload. Due to the obfuscation, FlowShredder changes an IPv4 header into an IPv6 header while keeping the Layer 4 payload unchanged. This allows FlowShredder to use a smaller Layer 4 payload to avoid MTU-induced packet splitting. If the end system does not support NAT64, the IPv6 packets will be tunneled over IPv4, which also reduces the Layer 4 payload. The payload efficiency here is defined as the ratio of the Layer 4 payload to that of the normal IPv4 packet (1500B). However, the transport cost is not a big issue in practice, since rich media applications usually have large packets and long-lived connections, making the obfuscation overhead negligible. Table 1 presents the transport costs and performance differences between TCP and UDP when used with IPv6 tunneling and NAT64 transition technologies.

Table 1 demonstrates that IPv6 tunneling and NAT64 slightly reduce the effective L6 payload compared to normal IPv4, with a maximum

degradation of less than 4%. Notably, NAT64 has better result than IPv6 because of the smaller header but requires extra operation. UDP benefits slightly more than TCP due to its smaller base header size, resulting in a marginally higher effective payload under the same conditions. Although small, this reduction could impact applications with strict MTU constraints or high packet-per-second rates. However, for most applications, especially those transmitting larger payloads or using path MTU discovery, such cost is negligible.

7. Related work

End-to-end encryption. TLS is the de facto end-to-end encryption system widely deployed in real networks. While it is highly secure, TLS consumes enormous resources at the end systems and requires extra RTTs to establish the session keys leading to increased latency. In a trusted CSP network, it makes sense to remove the computation burden to the gateways.

TLS uses AES for content encryption, which may potentially limit its throughput performance. For example, even with hardware acceleration in AES-NI, the throughput is limited to under 10Gbps with a dedicated x86 core and under 2Gbps in an ARM platform (AbdAllah et al., 2020). Considering the increasingly popular IoT scenarios and real-time applications, TLS may not always be the best choice. That said, FlowShredder and AES encryption are not mutually exclusive. The payload can be encrypted by AES while FlowShredder obfuscates connection identifiers, providing an anonymous effect and thus strengthening the protection. It is also possible for AES to reuse the rotated key generated by FlowShredder to optimize the overhead of TLS key exchange. In particular, FlowShredder may work with 0-RTT TLS for enhanced protection and reduced latency.

Anonymity. Anonymity systems, such as LAP (Hsiao et al., 2012), Dovetail (Nikolić et al., 2015), HORNET (Chen et al., 2015), and TARANET (Chen et al., 2018), obfuscate the IP addresses to break the association between the connection and the hosts. However, the association between the packets and the connection can still be discovered.

In addition, these systems usually require host intervention.

Some systems are based on the programmable data plane, *e.g.*, ONTAS (Kim and Gupta, 2019), PANEL (Moghaddam and Mosenia, 2019), and MIMIQ (Govil et al., 2020). These efforts suffer the same problem of traditional anonymity approaches while having to handle the new challenge of limited resources in the switch.

Encryption with programmable data plane. P4-AES (Chen, 2020) is an in-network encryption engine and thus relieves the end hosts from the encryption burden. It can be considered a P4 version of TLS and inherits its limit of requiring powerful computation capability. Moreover, the AES encryption in P4 cannot be completed within a single pipeline, even for a 16B message. As a result, P4-AES cannot achieve wire-speed encryption. PINOT (Wang et al., 2021) is a DNS-privacy approach that obfuscates the IP addresses of the DNS requests. However, it does not aim to protect the content. SPINE (Datta et al., 2019) aims to protect IP addresses but avoid the high overhead of IPSec. It obfuscates the IP address and TCP sequence number for each packet. SPINE does break the association between the data flow and the end hosts but relies on TLS for payload encryption. Some studies leverage the programmable data plane to obfuscate the topology (Meier et al., 2018) or adjust end-to-end encryption schemes (Liu et al., 2020). These studies are orthogonal and complementary to FlowShredder. Compared with our earlier short version published in ICSOC 2024 (Song et al., 2024), this paper extends the related work section with deeper technical and experimental analysis. In particular, we introduce a new queue-based packet reordering and cover-traffic generation mechanism to further enhance the resistance against traffic analysis, which was not included in the short version. We also broaden the experimental comparison by adding public-cloud evaluations and latency–security trade-off measurements, providing more comprehensive evidence of FlowShredder's effectiveness under real-world conditions.

8. Conclusion

We proposed FlowShredder, a protocol independent and in-network security service to protect the cloud traffic, based on the idea that without the context of the connection and the hosts the packets are of little value to the adversary. The packets appear to be unrelated. FlowShredder breaks the association between the packets, the connection, and the hosts, and thus achieves per-packet indistinguishability by obfuscating packet headers. FlowShredder's logic is implemented within a single pipeline in the hardware P4 switch for wire-speed performance. To route the packet correctly, FlowShredder takes advantage of IPv6 and uses randomized IP addresses. The service is particularly suitable for applications that do not require strong protection but are sensitive to latency, for example, some of the rich media and real-time applications. In case strong protection is required, FlowShredder may work with end-to-end encryption schemes such as 0-RTT TLS for enhanced protection and reduced latency. Future research may focus on integrating FlowShredder with emerging network paradigms, such as edge computing and 6G architectures, which impose stringent low-latency and high-throughput requirements. Such integration could reveal new design challenges and optimization opportunities. For example, to establish secure connections among more than two trusted networks, we will need to deploy our switch at the edge of each trusted network, and PARC (Chaudhary and Kumar, 2020) would serve as an important reference. Moreover, combining FlowShredder with advanced privacy-enhancing technologies, including differential privacy and homomorphic encryption, holds promise for enhancing resilience against increasingly sophisticated traffic analysis attacks. These hybrid approaches may offer a more robust privacy-preserving framework in dynamic and adversarial network environments.

CRediT authorship contribution statement

Bin Song: Writing – review & editing, Software, Methodology, Investigation, Formal analysis. **Bin Sun:** Writing – original draft, Visualization, Data curation. **Qiang Fu:** Resources, Project administration, Methodology, Investigation, Data curation. **Hao Li:** Supervision, Resources, Project administration, Methodology.

Funding

This work is supported by the National Key Research and Development Program of China (2022YFB2901403) and NSFC, China (62572831).

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data will be made available on request.

References

AbdAllah, E.G., Kuang, Y.R., Huang, C., 2020. Advanced encryption standard new instructions (AES-NI) analysis: Security, performance, and power consumption. In: International Conference on Computer and Automation Engineering.

Anon, 2008. The transport layer security (TLS) protocol version 1.2. https://datatracker.ietf.org/doc/html/rfc5246.

Anon, 2018. The transport layer security (TLS) protocol version 1.3. https://datatracker.ietf.org/doc/html/rfc8446.

Anon, 2021. $P4_{16}$ intel tofino native architecture - public version. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf.

Anon, 2022. Intel tofino. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html.

Anon, 2022a. Frontier - Hpe Cray Ex235a, Amd Optimized 3rd generation Epyc 64c 2ghz, Amd Instinct Mi250x, Slingshot-11. https://www.top500.org/system/180047/.

Aviram, N., Gellert, K., Jager, T., 2021. Session resumption protocols and efficient forward security for TLS 1.3 0-RTT. J. Cryptology 34 (3), 20.

Barrera, J.F., Vargas, C., Tebaldi, M., Torroba, R., 2010. Chosen-plaintext attack on a joint transform correlator encrypting system. Opt. Commun. 283 (20), 3917–3921.

Berman, R., Fiat, A., Ta-Shma, A., 2004. Provable unlinkability against traffic analysis. In: International Conference on Financial Cryptography.

Bogdanov, A., Knudsen, L.R., Leander, G., Standaert, F.-X., Steinberger, J., Tischhauser, E., 2012. Key-alternating ciphers in a provable setting: Encryption using a small number of public permutations. In: International Conference on the Theory and Applications of Cryptographic Techniques.

Bromberg, Y.-D., Dufour, Q., Frey, D., Rivière, É., 2022. Donar: Anonymous VoIP over tor. In: Usenix NSDI.

Chaudhary, R., Kumar, N., 2020. PARC: Placement availability resilient controller scheme for software-defined datacenters. IEEE Trans. Veh. Technol. 69 (8), 8985–9001. http://dx.doi.org/10.1109/TVT.2020.2999072.

Chaudhary, R., Kumar, N., 2021. Enflow: An energy-efficient fast flow forwarding scheme for software-defined networks. IEEE Trans. Intell. Transp. Syst. 22 (8), 5293–5309. http://dx.doi.org/10.1109/TITS.2020.2999134.

Chen, X., 2020. Implementing AES encryption on programmable switches via scrambled lookup tables. In: ACM SIGCOMM Workshop on Secure Programmable Network Infrastructure.

Chen, C., Asoni, D.E., Barrera, D., Danezis, G., Perrig, A., 2015. HORNET: High-speed onion routing at the network layer. In: ACM SIGSAC Conference on Computer and Communications Security.

Chen, C., Asoni, D.E., Perrig, A., Barrera, D., Danezis, G., Troncoso, C., 2018. TARANET: Traffic-analysis resistant anonymity at the network layer. In: IEEE European Symposium on Security and Privacy.

Datta, T., Feamster, N., Rexford, J., Wang, L., 2019. SPINE: Surveillance protection in the network elements. In: USENIX Workshop on Free and Open Communications on the Internet.

Govil, Y., Wang, L., Rexford, J., 2020. $\{MIMIQ\}$: Masking $\{IPs\}$ with migration in $\{QUIC\}$. In: USENIX Workshop on Free and Open Communications on the Internet.

Hauser, F., Häberle, M., Merling, D., Lindner, S., Gurevich, V., Zeiger, F., Frank, R., Menth, M., 2023. A survey on data plane programming with P4: Fundamentals, advances, and applied research. J. Netw. Comput. Appl. 212, 103561. http://dx.doi.org/10.1016/j.jnca.2022.103561, URL https://www.sciencedirect.com/science/article/pii/S1084804522002028.

Hsiao, H.C., Kim, T.H.J., Perrig, A., Yamada, A., Nelson, S.C., Gruteser, M., Meng, W., 2012. LAP: Lightweight anonymity and privacy. In: IEEE Symposium on Security and Privacy.

Kim, H., Gupta, A., 2019. ONTAS: Flexible and scalable online network traffic anonymization system. In: Proceedings of the 2019 Workshop on Network Meets AI & ML. pp. 15–21.

Kumar, P., Dezfouli, B., 2024. QuicSDN: Transitioning from TCP to QUIC for southbound communication in software-defined networks. J. Netw. Comput. Appl. 222, 103780. http://dx.doi.org/10.1016/j.jnca.2023.103780, URL https://www.sciencedirect.com/science/article/pii/S1084804523001996.

Langley, A., Riddoch, A., Wilk, A., Vicente, A., Krasic, C., Zhang, D., Yang, F., Kouranov, F., Swett, I., Iyengar, J., Bailey, J., Dorfman, J., Roskind, J., Kulik, J., Westin, P., Tenneti, R., Shade, R., Hamilton, R., Vasiliev, V., Chang, W.T., Shi, Z., 2017. The QUIC transport protocol: Design and internet-scale deployment. In: ACM SIGCOMM.

Liu, G., Quan, W., Cheng, N., Lu, N., Zhang, H., Shen, X., 2020. P4NIS: Improving network immunity against eavesdropping with programmable data planes. In: IEEE INFOCOM Workshops.

Meier, R., Tsankov, P., Lenders, V., Vanbever, L., Vechev, M., 2018. $\{NetHide\}$: Secure and practical network topology obfuscation. In: 27th USENIX Security Symposium (USENIX Security 18). pp. 693–709.

Moghaddam, H.M., Mosenia, A., 2019. Anonymizing masses: practical light-weight anonymity at the network level. arXiv preprint arXiv:1911.09642.

Nikolić, I., Wang, L., Wu, S., 2015. Cryptanalysis of round-reduced LED. Cryptology ePrint Archive, Paper 2015/429.

Piotrowska, A.M., Hayes, J., Elahi, T., Meiser, S., Danezis, G., 2017. The loopix anonymity system. In: USENIX Security Symposium.

Song, B., Sun, B., Fu, Q., Li, H., 2024. Flowshredder: a protocol-independent in-network security service in the cloud. In: International Conference on Service-Oriented Computing. Springer, pp. 327–334.

Wang, L., Kim, H., Mittal, P., Rexford, J., 2021. Programmable in-network obfuscation of DNS traffic. In: NDSS Workshop on DNS Privacy.

Yazdinejad, A., Parizi, R.M., Bohlooli, A., Dehghantanha, A., Choo, K.-K.R., 2020. A high-performance framework for a network programmable packet processor using P4 and FPGA. J. Netw. Comput. Appl. 156, 102564. http://dx.doi.org/10.1016/j.jnca.2020.102564, URL https://www.sciencedirect.com/science/article/pii/S1084804520300382.