`ditto`: WAN Traffic Obfuscation at Line Rate

Roland Meier
ETH Zürich
meierrol@ethz.ch

Vincent Lenders
armasuisse
vincent.lenders@ar.admin.ch

Laurent Vanbever
ETH Zürich
lvanbever@ethz.ch

***Abstract*—Many large organizations operate dedicated wide area networks (WANs) distinct from the Internet to connect their data centers and remote sites through high-throughput links. While encryption generally protects these WANs well against content eavesdropping, they remain vulnerable to traffic analysis attacks that infer visited websites, watched videos or contents of VoIP calls from analysis of the traffic volume, packet sizes or timing information. Existing techniques to obfuscate Internet traffic are not well suited for WANs as they are either highly inefficient or require modifications to the communication protocols used by end hosts.**

This paper presents `ditto`, a traffic obfuscation system adapted to the requirements of WANs: achieving high-throughput traffic obfuscation at line rate without modifications of end hosts. `ditto` adds padding to packets and introduces chaff packets to make the resulting obfuscated traffic independent of production traffic with respect to packet sizes, timing and traffic volume.

We evaluate a full implementation of `ditto` running on programmable switches in the network data plane. Our results show that `ditto` runs at 100 Gbps line rate and performs with negligible performance overhead up to a realistic traffic load of 70 Gbps per WAN link.

I. Introduction

Many large organizations operate dedicated wide area networks (WANs) as a critical infrastructure distinct from the Internet to connect their data centers and remote sites. For example, cloud service providers such as Google [66], Amazon [23], and Microsoft [30] operate WANs to achieve low-latency, high-throughput inter data center communication. Public safety and security organizations rely on WANs to achieve secure and reliable communication between their sites (e.g., [7], [17], [20], [28], [65]). For large organizations, these WANs provide 100s of Gbps to Tbps of capacity over long distances and can cost 100s of millions of dollars per year [63].

WANs are an attractive target for eavesdropping attacks and mass surveillance because they are often used to transport large amounts of sensitive data. And because WANs spread over large geographical areas, it is impossible to secure the cables physically from wiretapping. Past revelations show that intercontinental fiber links were subject to tapping by governmental agencies [27], [59] or other entities [77] and many devices are available to tap on fiber links [11], [25], [47], [68], [85]. Indeed, major operators such as Amazon, Microsoft, and OVH acknowledge that WAN traffic is at risk and they use MACsec [21] to encrypt their traffic not only at the application layer, but also at the link layer [29], [35], [76], [82].

Network and Distributed Systems Security (NDSS) Symposium 2022
27 February - 3 March 2022, San Diego, CA, USA
ISBN 1-891562-74-6
https://dx.doi.org/10.14722/ndss.2022.24056
www.ndss-symposium.org

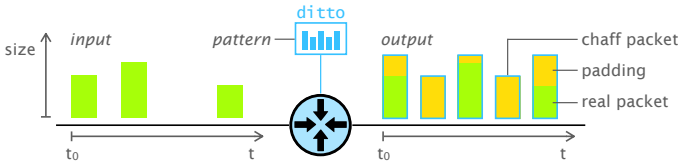


Fig. 1: `ditto` adds padding and chaff packets such that the outgoing traffic always follows a predefined pattern

However, it is well known that encryption alone is not sufficient to protect against traffic analysis attacks [53], [64]. Even if the network traffic is end-to-end encrypted, metadata such as the traffic volume, the packet sizes and the timing information reveals a lot about ongoing activities. As a result, eavesdroppers intercepting the WAN communication can still perform traffic analysis attacks. Such attacks are mostly known from Internet traffic, where past work shows that it is possible to infer the contents of VoIP calls [31], [46], streamed movies [50], [88]; visited websites [37], [61], [81], [91], [101], or the device identities [22], [24], [78], [83], [87], [93] without having to break the encryption. However, the same attacks can be applied to WAN traffic if the WAN carries the incoming and outgoing Internet traffic (which is typically the case if a company sends all Internet traffic via a central firewall). More generally, it has been shown many times (e.g., in [44], [49], [89]) that traffic classification also works for encrypted traffic.

Many techniques have been proposed to protect against traffic analysis attacks in the Internet. However, these techniques are not well adapted to the specific requirements of WAN traffic protection. Techniques such as BuFLO [22], CS-BuFLO [36], HORNET [42], or TARANET [43] add padding to obfuscate the size of individual packets and flows and require modifications on the software and protocols of the end hosts. For many organizations operating a WAN, it is impossible to adapt these protocols on all end hosts (e.g., because a cloud provider does not control the software that is running on its customer's instances). Other techniques such as Loopix [84], PriFi [32], or Wang et al. [98] impose strict transmission schedules and rates per flow, and thus severely limit the achievable throughput. As WAN traffic is high-throughput in nature, these solutions are not efficient enough to deal with high link traffic rates up to 100 Gbps.

This paper presents **`ditto`**, an in-network and hardware-based traffic obfuscation system specifically tailored to WANs. As illustrated in Fig. 1, `ditto` shapes traffic according to a predefined pattern (a periodic sequence of packet sizes at a fixed rate) using three operations: *(i)* packet padding; *(ii)* packet delaying; and *(iii)* chaff packet insertion. When there are "real" packets to transmit, `ditto` pads and transmits them. When there are no real packets, it transmits dummy "chaff" packets. Therefore, `ditto` only adds overhead (padding

or chaff packets) in a way that does not degrade the network performance for the real traffic. It fills the gaps when there is not enough real traffic to transmit and (slightly) delays real packets in order to make the resulting network behavior independent of the actual traffic being sent.

`ditto` runs on the gateway network devices (e.g., routers or switches) of the WAN sites and does not require any modification to the end hosts or protocols. Being network-based, it is further efficient and supports high-speed obfuscation even when the traffic is bursty and unpredictable. `ditto` devices can react locally to traffic changes in real time. This allows them to quickly adapt to different network loads and to add or remove chaff packets almost instantly depending on the actual link load. In contrast, application-based approaches that run on the end hosts lack the visibility of the network load and need to send chaff traffic independent of other applications, which creates a significant overhead that `ditto` does not have.

We implemented `ditto` using off-the-shelf programmable network hardware of the same type as major operators have already deployed [3], [4]. We show that `ditto` can obfuscate packet size, timing and volume information at line-rate. In the evaluation, we run interactive applications over `ditto` and we show that a `ditto`-enabled device can obfuscate up to 70 Gbps of production traffic (on a 100 Gbps link) without any significant impact on the network performance (in terms of throughput, packet loss, latency and jitter). This performance is enough for typical WANs since they usually run at (much) less than 60 % utilization [17], [28], [66]. Even in highly optimized WANs such as the ones of Google and Microsoft where the utilization is close to 100 % [63], [66], `ditto` could protect all the non-background traffic (which accounts to less than 50 % [63]). We further show that the efficient patterns computed by `ditto` result in a significant performance increase compared to simpler approaches in previous work while not compromising security properties against traffic analysis attacks.

Our main contributions are:

- a strategy to determine packet sizes that allow an efficient mixing of real and chaff packets (§V);
- an architecture to obfuscate the traffic volume and timing at line rate in network switches (§VI);
- a full implementation (available as open source[1]) on off-the-shelf hardware (§VIII); and
- an evaluation on real Internet traffic and with interactive applications (§IX).

The remainder of this paper is organized as follows. In §II, we describe the network and attacker models as well as the security goals. In §III we summarize the key concepts of programmable network devices. In §IV, we provide an overview over `ditto` before we describe its main components in more detail (pattern computation in §V and traffic shaping in §VI). In §VII, we discuss `ditto`'s security properties and limitations. In §VIII we describe the hardware implementation and in §IX we evaluate it. Finally, we review related work in §X and conclude in §XI.

[1] https://github.com/nsg-ethz/ditto

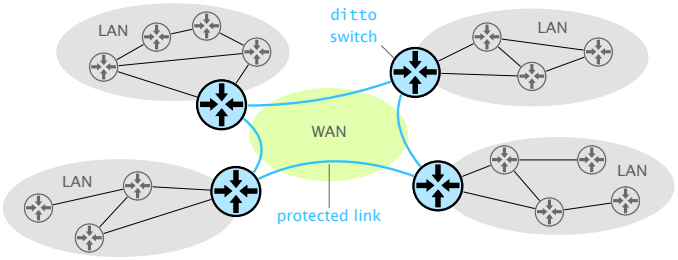


Fig. 2: Network model. `ditto` protects WAN links which interconnect different sites of an organization.

II. Model

In this section, we describe the network model (§II-A), the attacker model (§II-B), and `ditto`'s security goals (§II-C).

A. Network model

We consider a wide area network (WAN) which connects multiple sites of one organization over dedicated, encrypted tunnels as illustrated in Fig. 2. These tunnels can be created at layer 2 (e.g., leased fibers and MACsec encryption) or at layer 3 (e.g., IPsec tunnels). Each tunnel has a guaranteed bandwidth which the organization can fully utilize.

Each site is connected to the WAN with a *programmable switch*. These switches act as gateways between the local area network (LAN) in each site and the link(s) which interconnect the sites. The operator has full control over these switches and the LANs, but it does not control the WAN tunnels.

We note that such programmable switches are widely-available already *and* being deployed in large-scale infrastructure including AT&T, Deutsche Telekom, and Alibaba [3], [4].

B. Attacker model

We assume that the attacker has access to all devices and links in the WAN, but she does not have access to devices or links inside the LANs of the operator (including the gateways where `ditto` is running). The attacker can record timestamps and packet sizes but she cannot access the contents of packets since they are encrypted. We assume that the encryption happens at the same layer as the tunnel (e.g., MACsec [21] for a layer-2 tunnel, or IPsec [14] for a layer-3 tunnel). The attacker can also inject, modify, delay, or drop packets.

Our attacker model is realistic for typical organizations. As we elaborated in §I, several such wiretapping attacks happened in the past [27], [77] and major operators deploy link-layer encryption to mitigate them [29], [35], [76], [82].

C. Security goals

Similar to related work [43], [84], `ditto` shapes network traffic such that it satisfies the following security goals:

- *Volume anonymity:* The attacker cannot determine the real size of individual packets and flows which are sent over the WAN. This prevents attacks such as the one presented by Boshart and Rossow [34].

- *Timing anonymity:* The attacker cannot determine the timing between packets composing real traffic. This prevents attacks such as the ones presented by Wang et al. [99] and Feghhi and Leith [54].
- *Path anonymity:* The attacker cannot track packets across WAN tunnels. This prevents attacks such as the one presented by Wang et al. [99].

The key enabler for `ditto` to achieve these goals at line rate is that `ditto` operates in the network (on routers or switches) and not on end devices such as clients or servers. In the following section, we describe this deployment scenario.

III. Background on programmable switches

We base our design on programmable switches, as specified by the Portable Switch Architecture (PSA) [10] because these devices provide both high performance and the flexibility which we need to implement `ditto`. Such switches allow running custom programs (implemented in P4 [33]) in the data plane and can process traffic at terabits per second [8]. Programmable switches have been used for many security applications before [56], [69], including various types of obfuscation techniques [72], [74], [75], [94].

The PSA specifies five building blocks which each packet traverses: parser, ingress pipeline, traffic manager, egress pipeline, and deparser. Below, we provide more details about each of these building blocks.

The *parser* receives the incoming packet and extracts headers. The format of these headers is programmable (i.e., the parser can extract custom header formats). Only the parsed parts of the packet are accessible in the pipelines. The rest of the packet is considered as payload and cannot be modified.

The *ingress pipeline* receives the packet's headers together with metadata (e.g., its ingress port), which is all stored in the packet header vector (PHV). The pipeline consists of several stages in which match & action tables can match on data in the PHV and trigger actions to modify it.

The architecture and the focus on processing packets at line rate imposes three main limitations concerning the ingress (and egress) pipeline: *(i)* the number of pipeline stages limits the number of sequential actions that can be performed on each packet; *(ii)* the size of the PHV limits the size of the parsed headers and metadata (which can be seen as local variables); and *(iii)* operations which take a non-constant time per packet are not possible (e.g., loops, splitting or merging packets).

The *Traffic Manager (TM)* switches packets from ingress- to egress pipelines. If needed, the TM buffers packets in first-in first-out (FIFO) queues. When the egress pipeline can process the next packet, the TM selects a queue and sends its next packet to the egress pipeline. To determine the queue, the TM can use different strategies [90]: *(i)* the queue's priorities; *(ii)* weighted round-robin; *(iii)* a combination of both (round robin among queues with equal priorities).

The *egress pipeline* is identical to the ingress pipeline except that it is attached to an egress port. As a consequence, it is for example no longer possible to change a packet's egress port once the packet has passed the TM. Similarly, the *deparser* is the inverse of the parser: It takes the headers and the payload and assembles the final packet.

IV. `ditto` overview

In this section, we explain the high-level concepts behind `ditto` using a running example and Fig. 3.

Design goals The high-level goals of `ditto` are *(i)* to make the WAN traffic that an eavesdropper receives independent (in terms of packet sizes, inter-packet time and traffic volume) from the actual traffic that is exchanged over the network; *(ii)* to support high-throughput networks without degrading their performance; and *(iii)* to operate without requiring changes to end-devices (e.g., clients or servers).

Workflow `ditto` reaches these goals by running on programmable network devices (no changes to end-devices) and by shaping the incoming WAN traffic into a repeating sequence of packets with pre-defined sizes and timing. With `ditto`, the traffic actually flowing through the WAN is therefore perfectly independent from the real traffic entering it. A `ditto`-enabled switch shapes traffic by (possibly): *(i)* padding incoming packets, to regularize their sizes; *(ii)* buffering/delaying incoming packets, to regularize their timings and their relative order; and *(iii)* inserting chaff packets, to fill any possible gaps and ensure the consistency of the packet rates. Of course, enlarging packets and/or delaying them comes at a cost. `ditto` reduces this overhead by optimizing the shaping pattern.

In the paragraphs below, we rely on a simple example and Fig. 3 to explain how `ditto` determines the "shape" of the packet stream (we refer to this as the obfuscation pattern) and how `ditto` modifies traffic such that it follows this pattern.

Architecture `ditto` has two components: *(i)* a *pattern computation algorithm* to compute a secure and efficient traffic pattern based on the packet size distribution; and *(ii)* a *data-plane program* to shape traffic according to this pattern at line rate by padding packets and introducing chaff packets.

Simple example We consider a simplistic WAN composed of two `ditto` switches connected by one link. In this WAN, packets are of three sizes: 25 % of the packets are 500 B, 25 % are 1000 B and 50 % are 1500 B. The ordering of the packets follows an unknown distribution.

Pattern computation Given the packet size distribution as an input, `ditto` first computes an efficient obfuscation pattern. The *obfuscation pattern* specifies the order and sizes of packets traversing a link protected by `ditto`. We define it as an ordered list of packet sizes (the pattern states). `ditto` then repeats this pattern infinitely. For example, if the pattern is `[500,1000]`, `ditto` shapes the incoming traffic such that the outgoing packet sizes are `[500,1000,500,1000,500,1000,...]` at a fixed rate.

An efficient pattern minimizes the overhead in terms of padding (bytes added to a packet to make it larger) and chaff packets (dummy packets inserted to transmit at a constant rate). To minimize the amount of required padding, `ditto` computes the pattern such that it allows to distribute packets uniformly over all pattern states (this leads to minimal padding on average). To minimize chaff packets, `ditto` prefers short patterns (this reduces gaps between real packets). In §IX, we show that patterns of length 3 to 6 achieve good results.

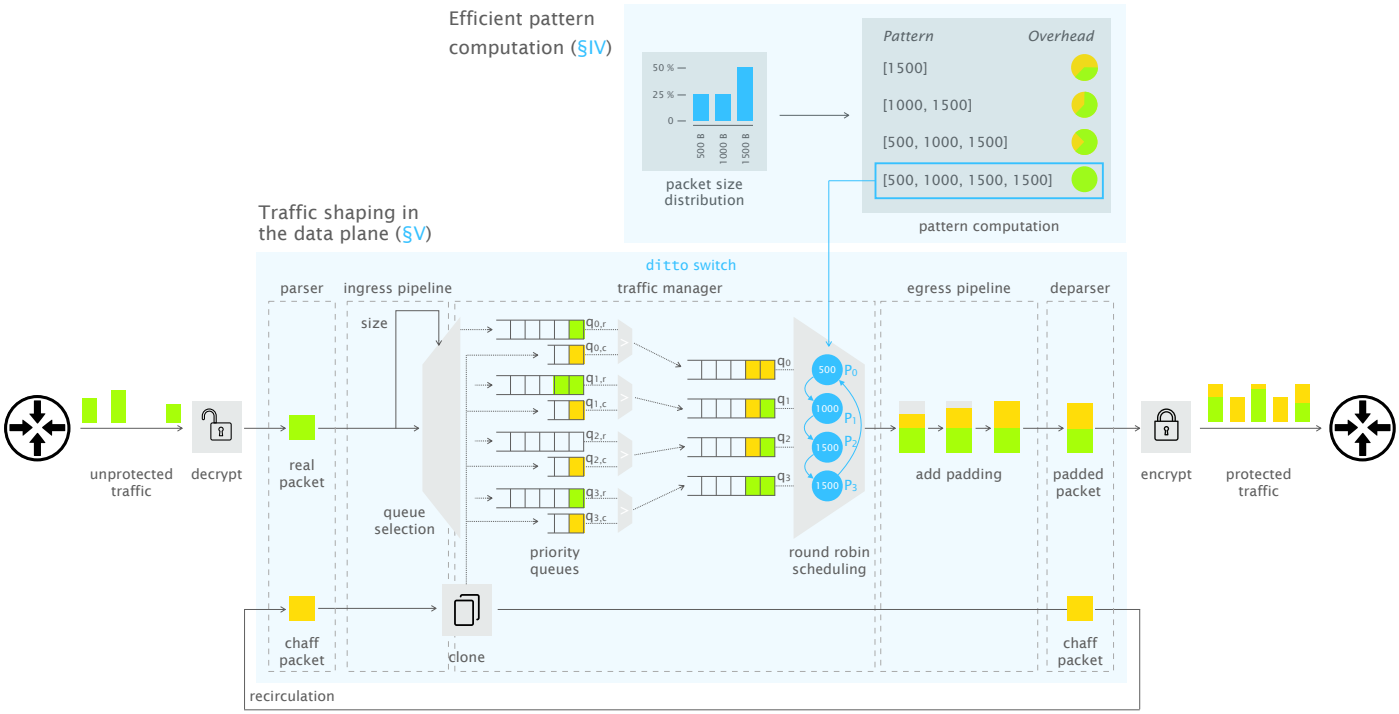


Fig. 3: `ditto` overview. The combination of priority queueing (real packets have higher priority and there is always a chaff packet ready to send with lower priority) and round-robin scheduling (one queue per pattern state) ensures that the outgoing traffic follows the predefined pattern regarding packet sizes and timing. The figure does not show the removal of padding on the other end of a protected link.

In the example from above, possible patterns include the ones listed below. For assigning packets to a pattern state, the objective is to minimize the amount of padding. Therefore, each packet is assigned to the next larger pattern state.

- `[1500]`: This would require to add 1000 B padding to 25 % of the packets (the ones of original size 500 B) and 500 B to another 25 % of the packets (the ones of 1000 B). But it minimizes the number of chaff packets since all packets can be padded to this size.
- `[1000,1500]`: Here, the 1500 B packets can be sent in the 1500 B state and the other packets in the 1000 B state. Therefore, `ditto` only needs to add 500 B of padding to 25 % of the packets. However, it needs to send chaff packets if multiple 1500 B packets arrive subsequently.
- `[500,1000,1500,1500]`: Here, the pattern equals the input distribution and `ditto` can send each packet without padding. However, it might need to send chaff packets depending on the order in which the real packets arrive.

For the continuation of the example, we use the pattern `[500,1000,1500,1500]`.

Traffic shaping The data-plane component of `ditto` merges incoming real packets with chaff packets such that the mix fits the pattern with minimal overhead. This is challenging because it needs to be performed in hardware to achieve high performance but typical networking hardware is not designed for this. `ditto` solves this challenges by combining switch queuing and scheduling to hierarchical queues with 2 levels:

When a packet arrives at a `ditto` switch, `ditto` first assigns it to a pattern state. A pattern state is one entry in the pattern; it defines the size which the packet has when it leaves the switch. Since `ditto` cannot split packets, it assigns a packet to the next-larger pattern state.[2] For example, a packet of size 800 B is assigned to the pattern state `1000` (P_1 in Fig. 3) such that `ditto` sends it next time it needs to send a 1000 B packet.

Each pattern state P_i has two first-in-first-out (FIFO) queues with priorities. A high-priority queue to which `ditto` assigns real packets ($q_{i,r}$) and a low-priority queue which `ditto` fills with chaff packets ($q_{i,c}$).

In the example, `ditto` assigns the 800 B packet to the high-priority queue $q_{1,r}$ belonging to the pattern state P_1.

Filling the low-priority queues with chaff packets requires a way to generate these packets. `ditto` achieves this by continuously recirculating chaff packets and cloning them into the low-priority queues. This does not require a dedicated traffic generator and it does not affect the switch performance (except that it requires one switch port to perform the recirculation).

`ditto` then feeds the output of each pair of priority queues $(q_{i,r}, q_{i,c})$ to a round-robin queue q_i and it configures their rates such that the output is $1/L$-th of the total rate (for a pattern of length L). As a result, each pair of priority queues will output packets at a constant rate and irrespective

[2] Fragmentation is often not available on switches or routers for performance reasons. The largest pattern state needs to correspond to the maximum size of any packet in the network (MTU). If multiple states have the same size, `ditto` distributes packets uniformly among them.

of whether there is a real packet or not (since there is always at least one chaff packet in each low-priority queue).

In the next phase, `ditto` performs round-robin scheduling among the round-robin queues (q_0, q_1, q_2 and q_3 in Fig. 3). Because the order of the round-robin queues corresponds to the obfuscation pattern, the scheduler then outputs packets according to the pattern.

After the queuing and scheduling in the traffic manager, `ditto` pads the packets in the egress pipeline. At this point, the ordering of the packets is already following the pattern and each packet is marked with its target size. `ditto` adds padding headers to compensate for the difference between the actual packet size and the target size. For example, it adds 200 B of padding headers for the 800 B packet from above.

Now that the packet has the right size, `ditto` sends it to the egress port. There, the packet needs to be encrypted (e.g., using MACsec, which can run at line rate [62]) before it leaves the switch. The encryption ensures that an attacker cannot see the padding and she cannot distinguish between real and chaff packets from content analysis.

V. Computing efficient traffic patterns

A naive pattern would be to make all packets equal size. However, this would be inefficient because network traffic contains a variety of different packet sizes and `ditto` would need to pad them all to the maximum size. For example, Internet backbone traffic is bi-modal (most packets are of size < 70 B or > 1400 B) [39].

In this section, we describe how `ditto` computes efficient patterns by minimizing the overhead for the expected traffic.

Definition An obfuscation pattern P for `ditto` is an ordered list of length L which specifies sizes of packets P_i:

$$P = [P_0, P_1, \ldots, P_{L-1}], \qquad P_i \in \mathbb{N} \tag{1}$$

Given such a pattern for a link protected by `ditto`, `ditto` orders and pads packets such that their size follows P. That is, the j^{th} packet is of size $P_{j \bmod L}$.

Every pattern is secure We first note that `ditto` achieves its security goals with every pattern because the pattern is static and therefore does not reveal information about the real traffic traversing a protected link.

Efficient patterns Obfuscation patterns differ in their overhead. Intuitively, a pattern is more efficient than another if it requires less padding and buffers/reorders real packets less. In the following paragraphs, we explain how we determine the pattern length L and its states P_i to obtain efficient patterns.

Selecting the pattern length The pattern length impacts the amount of: *(i)* padding required, longer patterns require less padding as they can better fit the original traffic distribution; *(ii)* chaff packets generated, shorter patterns generate less chaff packets because incoming packets are spread over fewer states; and *(iii)* packet reordering, longer patterns lead to more reordered packets because they require more queues.

In §IX, we show empirically that patterns of length 3–6 achieve good results in all dimensions and for realistic traffic.

Selecting the pattern states Since `ditto` iterates over the pattern and sends the same number of packets from each of the states over time, we compute the pattern such that each pattern state fits for $\frac{100}{L}$ % of the packets. This is the case if the pattern state P_i is equal to the $((i{+}1){\cdot}100/L)$-th percentile of the expected traffic distribution $\mathcal{D}$:

$$P_i = \text{percentile}_{(i+1)\cdot 100/L}\, \mathcal{D} \qquad i \in [0, 1, \ldots, L-1] \tag{2}$$

When to compute and update the pattern $\mathcal{D}$ models the distribution of real packet sizes expected on the protected link. Ideally, the operator computes it based on the real traffic (e.g., recorded prior to using `ditto`). Since this distribution only reveals information about the average traffic characteristics, it is usually not confidential. Otherwise, the operator can use publicly available data such as [38].

When $\mathcal{D}$ changes significantly, the operator can compute a new pattern and reconfigure `ditto` to use the new pattern without interruption. However, as we show in the evaluation (§IX), the same pattern can be used for many months of real Internet traffic with almost constant overhead.

VI. Traffic shaping in the data plane

In this section, we explain how `ditto` shapes traffic such that it follows the previously defined pattern.

Problem A switch running `ditto` receives packets with unpredictable size and at unpredictable times and it needs to ensure that the packets that leave the switch follow the predefined pattern (w.r.t. to packet sizes and inter-packet time). To achieve this, `ditto` needs to perform three operations using the capabilities of programmable switches: *(i)* add padding to real packets; *(ii)* buffer packets until they fit in the pattern; and *(iii)* insert chaff packets.

Architecture The architecture of a `ditto` switch is as follows (cf. illustration in Fig. 3). When a real packet arrives at the `ditto` switch, the parser first extracts information such as the IP header. Then, `ditto` determines the egress port depending on the packet's destination address and assigns it to one of the queues which belong to this egress port. For a pattern of length L, each egress port (these are the ports where obfuscated traffic leaves a `ditto` switch) has L queues and each queue corresponds to one state in the pattern (and therefore one packet size). The traffic manager then performs round-robin scheduling to send packets from the queues to the egress pipeline. There, `ditto` adds padding such that the packet's size eventually matches the target size determined by the pattern. Finally, the packet exits at the egress port.

Unfortunately, typical round-robin scheduling has a property that is not optimal for `ditto`: It skips a queue if it does not contain a packet. This is problematic for `ditto` because it leads to skipped states in the pattern. To avoid this, `ditto` makes sure that there is always at least one packet in each queue. If there is no "real" packet available, the switch sends a "chaff" packet.

Assigning packets to queues `ditto` selects the queue to which it assigns a packet such that the amount of padding is minimal.

Since ditto can only make packets larger, it selects the next-largest queue i for a packet of size s and pattern states P_i:

$$i = \arg\min_i (P_i - s \mid s \leq P_i) \tag{3}$$

If the packet fits into more than one state with the same amount of padding, ditto randomly selects one of them.

Round-robin scheduling to implement the pattern ditto configures all queues of an egress port with the same priority such that the traffic manager (TM) performs round-robin scheduling. The TM therefore iterates over all queues and sends one packet from each non-empty queue.

The main challenge to ensure that the sent packets always follow the pattern is therefore to make sure that a queue is never empty when the TM tries to send a packet from it. Ideally, the hardware would allow to inject a "chaff" packet when the TM attempts to send a packet from an empty queue. While this would be a small extension in hardware, there was no need for such a feature so far and thus it does not exist. Below, we describe how we combine priority queueing and round-robin scheduling to overcome this limitation.

Priority queuing to mix real and chaff packets To ensure that round-robin queues are never empty, we implement *hierarchical queueing* with 2 levels. The idea is to combine priority queues with round-robin scheduling. For each pattern state, there is a pair of priority queues: A high-priority queue ($q_{i,r}$) which receives the real packets belonging to this state, and a low-priority queue ($q_{i,c}$) which is flooded with chaff packets for this state. Each pair of priority queues produces a constant stream of packets while prioritizing real packets. These outputs are then fed into the round-robin scheduling which produces the pattern. We detail the implementation in §VIII.

Custom headers to add padding to packets After the TM ensured that packets reach the egress pipeline in the right order, ditto adds padding such that they have the right size. ditto pads packets by adding additional headers after the Ethernet header (adding and removing custom headers is something that programmable switches are designed to do at line rate). Because the structure of headers needs to be defined at compile time, ditto uses a combination of multiple headers of different sizes (32, 16, 8, 4, 2 and 1 Byte) to add the right amount of padding to every packet. To allow the receiver to identify padded packets and to remove the padding, ditto marks padded packets in the Ethernet header using the EtherType field. Since a device running ditto encrypts the traffic over the WAN links, ditto can add padding with arbitrary contents and include information about the packet's original size such that the endpoint knows how much padding to remove.

Removing padding from packets The receiving switch recognizes padded packets based on their value in the EtherType field and can thus remove the padding without additional information or overhead. To remove the padding from packets before they are sent over a link which is not protected (e.g., to an end host), ditto removes all the padding headers and restores the original EtherType.

VII. Security analysis and limitations

In this section, we explain why ditto achieves the security goals from §II-C and we discuss ditto's limitations.

A. Security goals

Volume anonymity The obfuscation pattern together with the link bandwidth defines the traffic volume (in terms of bytes and packets) that is transmitted over every protected link. Since this volume is static and independent from the real traffic, an attacker cannot learn anything from it other than the maximum number of bytes and packets sent over the link if the link was fully utilized. Naturally, an attacker with access to multiple links and additional background information can derive an upper bound for the total traffic volume (cf. §VII-B).

Timing anonymity Because ditto always sends traffic according to the same pattern and at the same rate, it does not leak timing information. Since packets are encrypted such that the ciphertext changes for each WAN link, attackers cannot distinguish real and chaff packets and they cannot determine which packets belong to the same host, application or flow.

Path anonymity Even if an attacker can eavesdrop on all links connected to a ditto switch, she cannot link incoming and outgoing packets because *(i)* the pattern of incoming and outgoing packets is always the same and ditto would rather drop packet than violate the pattern; *(ii)* she does not know which packets contain real traffic; and *(iii)* packets are encrypted such that the ciphertext changes for every WAN link.

Summary ditto ensures that the traffic seen on each protected link is independent of the real traffic crossing this link. From an attacker's perspective, the traffic always looks the same: packets whose size follows a repeating pattern, with constant inter-packet time and random contents (because of the encryption). Assuming a bug-free implementation and properly working hardware, there is therefore no difference between the observed traffic when there is real traffic and when there is only chaff traffic. Thus, traffic analysis attacks would produce the same result in both situations and do therefore not work. This also extends to other properties than timing and size, such as packet directions and to multiple colluding attackers.

B. Limitations

While ditto achieves its security goals and prevents traffic analysis attacks, there are some limitations and potential attack vectors outside of our threat model. We discuss them below.

Malicious insider An attacker who has compromised multiple hosts (e.g., servers in two datacenters connected through a ditto-protected link) can *(i)* try to estimate the real traffic volume by measuring the performance of her own traffic; and *(ii)* exploit ditto for a DoS attack.

To *estimate the real traffic volume*, the malicious insider can leverage the fact that ditto enforces a predefined traffic pattern and that it does not split packets. We illustrate this with a simple example: Assuming the pattern is [500,1500], the protected link runs at 100 packets per second (pps), the attacker knows that the link can carry at most 50 packets of size

between 501 B and 1500 B per second. If the attacker can send and receive 50 packets of size 1500 B per second herself, she can use the observed delays or losses to roughly estimate the amount of other real traffic of this size because otherwise there would have been congestion or losses. Since the traffic created by this attacker is likely untypical for the compromised servers, it could be detected using anomaly detection techniques.

To *exploit* `ditto` *for a DoS attack*, the malicious insider can follow a similar strategy as above and she can create her traffic such that it requires a maximum amount of padding. For example, if the pattern is `[500,1500]`, she can repeatedly send packets of size 64 B (the smallest IP packet size) and 501 B. `ditto` would pad these packets to 500 B and 1500 B respectively and thereby help the attacker to amplify her volume. Since the attacker's packets compete with benign traffic for "transmission slots", the attacker can partially prevent benign traffic from being sent if her rate is high enough. We see two possible approaches for mitigating this attack: *(i)* Within each pattern state, packets could be prioritized based on the amount of padding which is required (lower priority for packets which require more padding). Then, packets from such an attacker would be dropped first but this would also impact benign traffic that requires a lot of padding. *(ii)* The rate at which each user can send packets of a certain size could be limited (e.g., 10 packets with 101 B per second). Packets exceeding this limit could be handled with lower priority such that the measure only has an impact when the network is congested.

Attacker with insider knowledge If the attacker has additional knowledge of the network topology, she can potentially also use this to *estimate the real traffic volume*. The best case (for the attacker) is a linear topology where the traffic crosses multiple links and each link uses a different pattern. An extreme example is a linear topology with two links where the first one used a pattern that sends only MTU-sized packets (i.e., 1500 B) and the second one uses a pattern that sends minimal-sized packets of 64 B. If both of these links run at 100 Gbps, the first one transmits around 8 Mpps (million packets per second) and the second one 195 Mpps. If the attacker knows that all traffic crosses both links, she can derive an upper bound of 8 Mpps (because `ditto` does not concatenate or split packets) and 8 Mpps $\times$ 64 B ≈ 4 Gbps throughput (because `ditto` only makes packets larger). However, this weakness is not harmful in practice because *(i)* both links would use the same pattern if the pattern was computed as described in §V; and *(ii)* it does not break the volume anonymity property.

Compromised or faulty hardware `ditto` runs on network switches and requires them to be trusted and to function properly. If this is not the case, it allows various attacks.

If an attacker has administrative access to a `ditto` switch, she can easily break `ditto`'s security properties (e.g., by simply disabling `ditto`). While this is not in our threat model, there are existing techniques to mitigate such attacks [45].

If the used hardware leaks information (e.g., because the scheduling is not working properly or the encryption scheme is weak), this can naturally also weaken `ditto`'s security.

VIII. Implementation

We fully implemented `ditto` in P4 (traffic shaping) and Python (pattern computation). Since implementing the pattern computation is relatively straightforward, we focus on the P4 implementation, which is technically challenging (cf. §VI).

The source code of our implementation is available GitHub: https://github.com/nsg-ethz/ditto.

Hardware target Our data-plane implementation runs on Intel's Tofino chipset [8], which powers several off-the-shelf switches [2], [79], [80] and is used by large operators (e.g., AT&T, Deutsche Telekom, and Alibaba [3], [4]).

Architecture The architecture of our implementation follows the description in §VI. In the ingress pipeline, we *(i)* determine the egress port for the incoming packet; *(ii)* we check if the egress port is one that we want to obfuscate and whether it is a real packet (as opposed to a chaff packet); if so, we *(iv)* assign the packet to the right queue; and *(v)* check if the switch can add enough padding in one pipeline pass or if the packet needs to be sent through the pipeline multiple times.

Approximating hierarchical queueing As explained in §VI, one challenge behind `ditto` is that the switch needs to send a packet from each round-robin queue *even if the queue is empty*. This is not possible in existing switches. For `ditto`, we implemented an approximation of hierarchical queueing, where a packet traverses two queueing stages instead of just one. We achieve this by sending each packet through the switch data plane twice. As illustrated in Fig. 4, the first queueing stage consists of one pair of priority queues for each pattern state. The queue with the higher priority receives all real packets belonging to the respective pattern state, while the queue with the lower priority is flooded with chaff packets.

We set the output rate of each priority-queue pair such that the sum of all pairs equals the total sending rate of the switch. For example, if the switch sends 10 Mpps and $L = 4$, each queue pair needs to transmit 2.5 Mpps. The outputs from the priority queues are fed back to the switch via loopback ports.

When the packets arrive at the switch for the second pass, `ditto` sends them to the round-robin scheduler attached to the actual egress port. Its output then follows the pattern.

The main cost of sending each packet through the switch twice is that the loopback modules occupy ports which cannot be used for interconnections with other switches. Since each physical 100 Gbps QSFP port can be split into 2 or 4 sub-ports (with 50 Gbps or 25 Gbps throughput each, respectively), one loopback port can be used for up to 4 pattern states. The bandwidth of one sub-port needs to be at least $100/L$ Gbps where L is the length of the pattern. Since the bandwidth can only be 100, 50, or 25 Gbps, `ditto` configures it as follows:

$$bw = \begin{cases} 100, & L = 1 \\ 50, & L = 2 \text{ or } L = 3 \\ 25, & L \geq 4 \end{cases} \tag{4}$$

And the number of required loopback ports n computes to

$$n = \left\lceil \frac{L \cdot bw}{100} \right\rceil \tag{5}$$

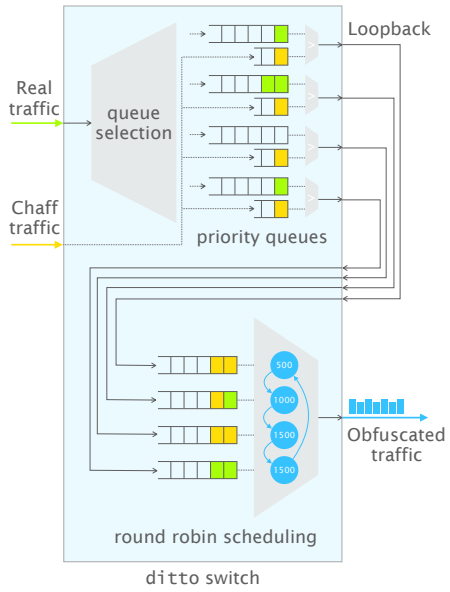


Fig. 4: `ditto` implements hierarchical queueing by sending each packet through the switch twice.

For example, patterns of length 3 and 6 require 2 loopback ports for each obfuscated port. In the typical use-case where an organization uses a `ditto` switch as its gateway to the WAN, switch ports are not scarce (e.g., a typical switch has 64 ports [8] and could therefore support around 20 WAN links).

Adding padding In the egress pipeline, `ditto` adds padding to the packet until it has the required size. `ditto` adds padding in the form of additional headers of different sizes (between 32 B and 1 B). `ditto` adds the padding headers in decreasing order in separate stages to reduce the number of match & action table entries: First, it adds as many 32 B padding headers as possible (and needed). Then, it tries to fill the remaining space with 16 B headers and so on until there is no more padding needed. For example, `ditto` adds 2×32 B, 1×4 B and 1×2 B padding headers to increase a packet's size by 70 B.

Recirculate packets if needed If the required amount of padding is larger than what can be added in one pipeline pass, `ditto` *recirculates* the packet. Then, the packet traverses the switch multiple times (i.e., from the end of the egress pipeline, it is sent to the beginning of the ingress pipeline). With each pass, `ditto` can add additional 254 B of padding. By default, `ditto` uses two 100 Gbps port for the recirculations. Depending on the pattern and the traffic distribution, this can result in a bottleneck especially if `ditto` protects traffic on multiple ports. In this case, `ditto` can load-balance the recirculations over more ports (thereby reducing the number of available switch ports for other interconnections).

Removing padding Removing padding is straightforward: `ditto` removes all the padding headers and restores the original EtherType. Again, the limited size of the PHV (Packet Header Vector, cf. §III) can require recirculating the packet in order to remove all padding.

Results from hardware measurements	
Performance and efficiency	§IX-B
• Longer patterns reduce the padding and chaff overhead.	
• `ditto`'s padding strategy leads to less overhead compared to related work.	
• Non-interactive traffic (replayed traces): no performance loss for link loads between 60 and 70 Gbps (depending on the dataset).	
• Interactive traffic (iPerf, VoIP and web browsing): no performance loss for link loads between 70 and 80 Gbps (depending on the dataset).	
Security	§IX-C
• Packet sizes and timing do not allow conclusions about real traffic.	
Results from simulations	
Performance and efficiency	§IX-D
• Longer patterns reduce the padding and chaff overhead.	
• 1 MB of buffer space is enough to obfuscate a traffic volume of up to 99 % of the link bandwidth.	
• The same pattern can be used for months without sacrificing efficiency.	
• 92 % of the packets remain in the correct order for highest load and the longest pattern.	
Security	
• The pattern produced by `ditto` is secure by design.	

TABLE I: Evaluation summary

Resource usage The main bottleneck regarding resource usage of `ditto` is the amount of padding that can be added in one pipeline pass. Because `ditto` adds padding in the form of additional headers, the amount is limited by the size of the PHV and the deparser. In our current implementation, `ditto` can add up to 254 B of padding in one pipeline pass. Regarding other types of resources (e.g., SRAM and TCAM memory), our implementation uses only a small fraction of the switch's resources (less than 10 % on average over all stages).

IX. Evaluation

We first describe our methodology in §IX-A. Then, we evaluate our implementation with respect to performance (§IX-B) and security (§IX-C). In addition, we outline the potential of future hardware optimized for `ditto` through simulations (§IX-D). Table I summarizes our main results.

A. Datasets and methodology

The distribution of traffic processed by `ditto` depends on the type of WAN in which it is deployed. As a typical use-case, we envision an organization with several sites which uses its WAN to connect them and use one site as a gateway through which it sends all outgoing and incoming traffic. In this case, WAN traffic has similar characteristics as Internet traffic. In addition, we consider two extreme cases: *(i)* the best case for `ditto` where all packets have the same size (e.g., if there was heavy traffic shaping); and *(ii)* the worst case for `ditto` where the packet sizes are uniformly distributed.

Datasets We use real Internet traffic and synthetic traffic where the packet sizes follow given distributions. For the real Internet traces, we use the publicly available CAIDA anonymized Internet traces dataset [38] (CAIDA). Even though this dataset was collected on Internet links, we believe it is representative for a WAN where one site is used as a gateway for all incoming and outgoing traffic (e.g., for central compliance monitoring). We use the most recent dataset

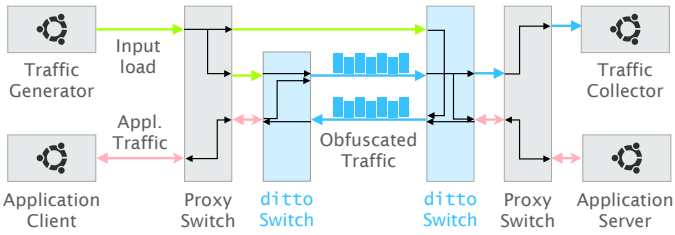


Fig. 5: Evaluation setup. We use two ditto switches, two switches which act as proxies to perform measurements (e.g., timestamps), and two servers to send and receive traffic.

(captured in January 2019) and we preprocess it in two steps: *(i)* we remove all non-IPv4 packets because the current implementation of ditto can only handle IPv4 traffic;[3] and *(ii)* we extract the first 100 M packets to speed up our simulations.

In addition, we generate two synthetic traffic traces using Scapy [16]: one where all packets have size 1480 B (CONSTANT) and one with uniformly distributed packet sizes between 60 B and 1480 B (UNIFORM). While these traffic distributions are unlikely to occur in practice, we use them to represent two extreme cases: CONSTANT is the best case for ditto because it already follows a constant pattern while UNIFORM represents the worst case because fitting a pattern to a uniform distribution creates the highest overhead.

Many WANs will have different traffic distributions than the traces that we use in the evaluation, but we argue that our datasets are useful to show ditto's performance on a wide range of different traffic characteristics. It is also important to note that the traffic distribution does not have an impact on ditto's security properties.

All datasets mentioned above represent non-interactive traffic. That is, the traffic does not change depending on the network behavior. While this allows us to test ditto with high traffic volumes (up to 100 Gbps), it is not fully representative for real-world behavior (e.g., a dropped TCP packet would be retransmitted). To address this, we run *interactive applications* on top of the replayed traffic and we measure the performance achieved by these applications. We describe this in §IX-B.

Methodology We evaluate ditto on off-the-shelf devices and we simulate how it would perform on ideal future hardware.

The *hardware prototype* is our implementation as described in §VIII. It runs on two Intel Tofino switches with 32×100 Gbps ports. Between the switches, ditto obfuscates traffic on one link at 100 Gbps. We use a traffic generator (Moongen [52]) to inject traffic from one server, and to record traffic on another server (cf. Fig. 5).

We also implemented a *simulator* for optimal hardware in Python. It receives a packet sequence as input (the real traffic) and produces a packet sequence as output (the obfuscated traffic). The simulator operates in discrete time steps: in each iteration, it receives and sends one packet. To shape traffic according to the pattern, it uses round-robin scheduling and when there is no real packet to send it sends a chaff packet.

[3]This is only an implementation detail. The same approach would also work for IPv6 packets.

B. Performance and efficiency in today's hardware

In the following paragraphs, we show the performance of ditto with respect to these three aspects:

- *Throughput:* The ratio between the incoming and the outgoing real traffic
- *Recirculations:* The number of recirculations of each packet
- *Application performance:* The performance of interactive applications

Throughput Fig. 6 shows the ratio between incoming real traffic and outgoing real traffic. To obtain these results, we send traffic from a server to a first switch where we add additional information to packets that we need for our measurements (e.g., we add a number to each packet). Then we send it to the switch which runs ditto. Afterwards, we record the obfuscated traffic on another server (see Fig. 5).

The total outgoing traffic is always 100 Gbps (not shown in the plot). If there was no performance loss, the amount of incoming traffic would equal the amount of outgoing traffic. But since ditto makes packets larger and fits them into a predefined pattern, it creates overhead and therefore reduced the usable throughput. However, as Fig. 6 shows, the hardware prototype operates almost without loss until 90 % (CONSTANT), 70 % (UNIFORM) or 60 % (CAIDA) load.

The reasons for this sub-optimal performance include:

- ditto relies on precisely controlled output rates of priority queues such that the output does not fluctuate. However, today's switches are typically not designed for that (they offer traffic shaping, but the rate is only correct "on average"). In our case, bursts of too much traffic lead to dropped packets.
- ditto adds padding in the form of additional headers. Treating padding like packet headers is expensive with respect to the required resources in the pipeline and the deparsing time, and a switch that could add padding without using these resources could perform better.
- ditto uses a maximum bandwidth of 100 Gbps for recirculation. Therefore, if packets need to be recirculated multiple times, this bandwidth is not enough and packets get lost.

Comparison with related work In Fig. 6 we also show an upper bound for the performance of three systems that rely on end-host protocols to obfuscate traffic: HORNET [42], TARANET [43] and BuFLO [22]. To compute the performance of these systems, we only simulate their padding strategy (i.e., we neglect computational overhead and overhead due to mixing with chaff packets), hence the real results of these systems would be strictly worse than the plotted upper bound.

Our results show that ditto outperforms all of these approaches even if their computational overhead is ignored (the results for ditto are measurements on actual hardware and therefore include all forms of overhead). ditto outperforms these approaches because *(i)* ditto adds padding according to an efficient pattern, which produces less overhead than padding all packets to the same size; and *(ii)* operates on a per-link basis as opposed to obfuscating each flow separately.

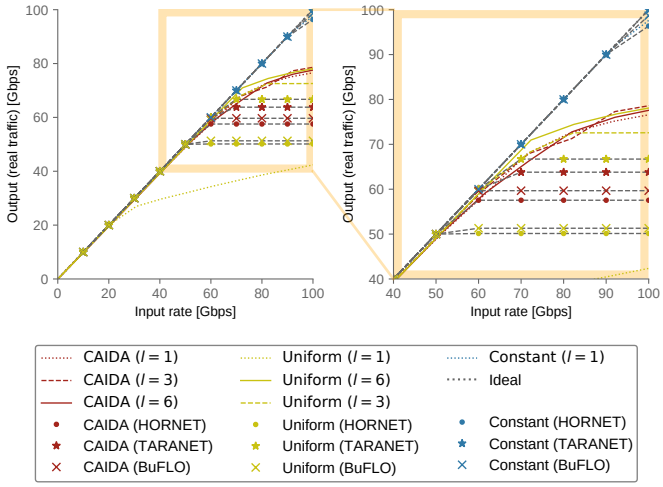


Fig. 6: Input vs. output rate of real traffic. Longer patterns and constant size traffic lead to higher goodput. `ditto` outperforms related work even their computational overhead is ignored.

Below, we provide more details about the simulated parameters.

HORNET pads all packets to the same size s. We set s to the size of the largest packet in the respective dataset, which minimizes the padding. Further, HORNET adds headers to the original packets. The size of these headers depends on the AS path length and a sample length. We set the AS path length to 1 and the sample length to 16 (as in the paper [42]).

TARANET shapes flow(lets) such that they transmit at a fixed rate (constant size and inter-packet time) and it adds chaff packets to obfuscate the real length of a flow(let). In contrast to `ditto`, TARANET can also split packets to make them smaller. For our simulations, we do not consider chaff packets (which makes the results only better) and we set the packet size to the average packet size of the respective dataset.

BuFLO's padding strategy is to pad all packets to the MTU (1500 B in our case) without modifying the timing. Again, we only simulate the padding overhead without considering chaff packets. Ideally, this approach can work without additional headers which is why we assume there is no such overhead.

Recirculations Since our hardware prototype can only add 254 B of padding per pipeline pass, some packets need to traverse the switch multiple times. For this, we use a technique called recirculation which sends a packet from the end of the egress pipeline back to the beginning of the ingress pipeline.

Recirculating packets increases reordering and packet delay and (because the bandwidth for recirculation is limited) can lead to packet loss. Therefore, it is better to keep the number of recirculations small. The best strategy to reduce recirculations is to make patterns long enough such that the difference between each pattern state and the next smaller or larger one is less than the number of padding bytes per pipeline pass.

CONSTANT does not require recirculations. For CAIDA and UNIFORM, the number of recirculations decreases with an increasing pattern length. A pattern of length 1 requires 1.59 (CAIDA) or 0.99 (UNIFORM) recirculations on average. This numbers decrease to 0.23 and 0.40 for a pattern of length 3 and 0.18 and 0.03 for length 6.

Application performance In addition to the raw throughput measurements above, where we replayed static traffic traces, we now measure the performance of interactive applications.

We measure the real-world performance of three typical types of applications: *(i)* high-bandwidth TCP and UDP traffic (using iPerf [9]); *(ii)* web browsing traffic (using WprGo [40]); and *(iii)* VoIP traffic (using pjsip [12]). Fig. 5 illustrates the setup of this experiment. We run Docker containers with clients and servers for the respective application and we measure the performance of these applications when the traffic is sent via a `ditto`-enabled link versus the performance when the traffic is forwarded directly (we use this as the baseline).

Fig. 7 shows the results of these measurements. We obtain the results by running 50 measurements for each input rate (0–100 Gbps) and we perform each measurement with and without `ditto`. We highlight that we perform the measurements *in addition* to the replayed traffic. Therefore, a TCP throughput of 7 Gbps at an input rate of 80 Gbps means that `ditto` is handling 87 Gbps in total. The measurements without `ditto` represent a baseline where traffic is directly forwarded.

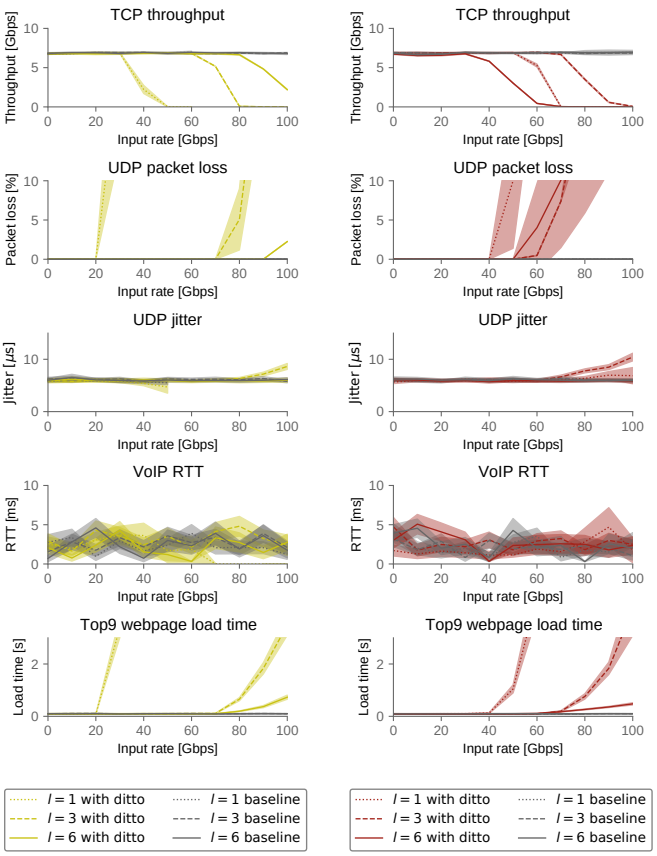
We measure the following metrics:

- *TCP throughput:* Important for applications which demand high bandwidth and reliable throughput (e.g., file transfer). We measure TCP throughput by creating 30 s TCP flows with maximum throughput using iPerf.
- *Packet loss:* Important for applications using unreliable transport (e.g., video streaming over UDP) and reliable transport (e.g., TCP). We measure packet loss by creating 30 s UDP flows with 1 Gbps throughput using iPerf.
- *Jitter:* Important for real-time applications and streaming (e.g., video calls). We measure jitter by creating 30 s UDP flows with 1 Gbps throughput using iPerf.
- *Round-trip time (RTT):* Important for real-time applications (e.g., VoIP). We measure the RTT by creating 8 × 30 s VoIP calls using PJSIP.
- *Website load time:* Important QoE metric. We measure the load time for the 9 most popular websites according to Alexa [1].[4]

The results in Fig. 7 show that – as expected – `ditto` does not degrade the network performance up to a certain point. Depending on the distribution of the traffic and the length of the pattern, `ditto` does not have significant impact on the network performance with an input rate of up to 80 Gbps (TCP throughput, UNIFORM, pattern of length 6). The main causes for the degraded performance are dropped packets (due to the same reasons as discussed above). If packets are not dropped, we highlight that `ditto` has no significant effect on timing-related metrics such as jitter and Round-Trip Time (RTT).

In general, longer patterns are more efficient because they produce less overhead. Most of our results confirm this hypothesis. However, there are two exceptions in the CAIDA dataset: TCP throughput and packet loss. The reason for this

[4] amazon.com, facebook.com, netflix.com, reddit.com, youtube.com, zoom.us, bing.com, google.com, and wikipedia.org



(a) UNIFORM (b) CAIDA

Fig. 7: `ditto` compared to the baseline. Lines show mean values and colored areas indicate the 95 % confidence interval. `ditto` affects the application performance after a certain link load, depending on the dataset, pattern length and metric.

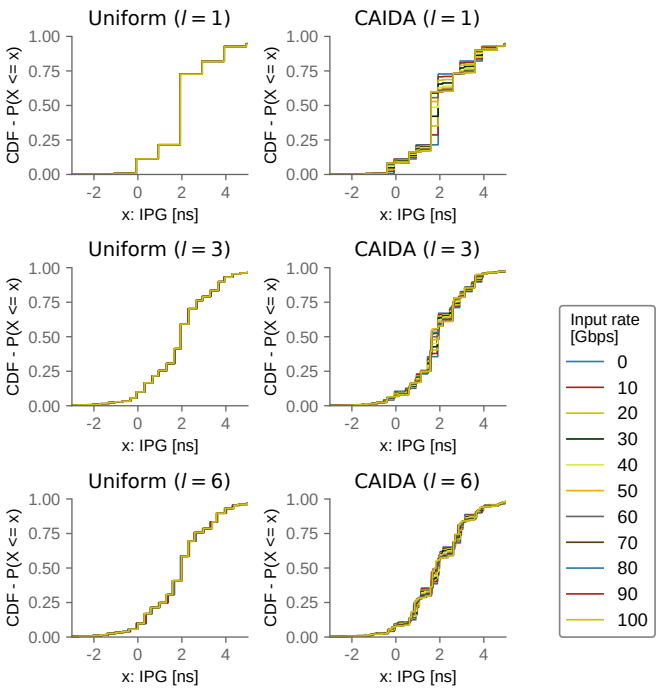


Fig. 8: IPG distributions. The IPG does not depend on the input rate (the 11 lines in each plot are overlapping).

is that we do not take the interactive application traffic into account when we computed the pattern. Then, it can happen that the measurements modified the traffic distribution to an extent where the pattern does not fit well anymore. Especially for longer patterns (because each state of the pattern has a small share of the total amount of transmitted packets, e.g., 1/6th $L = 6$) and for flows with many equal-sized packets (because then all packets are assigned to the same pattern state). Both the TCP and the UDP measurements consist of constant-size packets because iPerf maximizes the throughput.

C. Security in today's hardware

We now show that the hardware prototype obfuscates traffic such that the observed inter-packet times and packet sizes are independent from the real traffic and we show that the accuracy of a state-of-the-art attack is on par with random guessing.

Packet timings are independent from the real traffic Fig. 8 shows the Inter-Packet Gap distribution (IPG, the time between the end (last byte) of the previous packet and the start (first byte) of the current packet) for each dataset, pattern length and network load. Visually, it is clear that the distributions do not depend on the network load (the lines largely overlap). Analyzing the measurements further shows that a large fraction is within a typical error margin around the median value. For example, an attacker who can measure timestamps with a precision of $\pm$ 3.2 ns (as in a state-of-the-art capturing device [55]), could not distinguish between 92 % and 97 % (depending on dataset and pattern length) of the measurements.

The numbers in Fig. 8 are subject to imprecisions for two reasons: *(i)* we measure the timestamps in the ingress pipeline of the evaluation proxy switch (cf. Fig. 5), i.e., not on the link directly and not on a calibrated measurement device; and *(ii)* we measure the timestamp at the beginning of a packet, while the IPG refers to the time between the last byte of the previous packet and the first byte of the current packet. We therefore adjust the timestamp computationally as follows. For two packets of sizes s_0 and s_1 bytes arriving at timestamps t_0 and t_1 ($t_0 < t_1$) and a line rate of 100 Gbps, the IPG is

$$IPG = t_1 - t_0 + \frac{s_0 \cdot 8}{100 \cdot 10^9} \quad (6)$$

Packet sizes are independent from the real traffic We now evaluate whether the hardware prototype obfuscates traffic such that it follows the defined pattern. Round-robin scheduling in today's switches is designed to follow round-robin behavior *on average*, not necessarily in microscopic detail. This means that the switch performing round-robin scheduling could send $[P_1, P_1, P_2, P_2, P_3, P_3]$ instead of $[P_1, P_2, P_3, P_1, P_2, P_3]$ (where P_i represents a packet from queue i). However, since this behavior originates in the hardware implementation of the switch and not in the behavior of `ditto`, it does not affect `ditto`'s security.

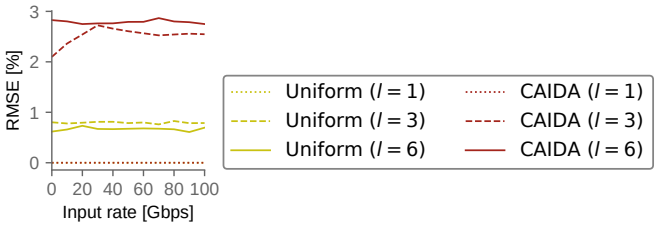


Fig. 9: `ditto` performs round-robin scheduling up to an error which does not depend on the input rate.

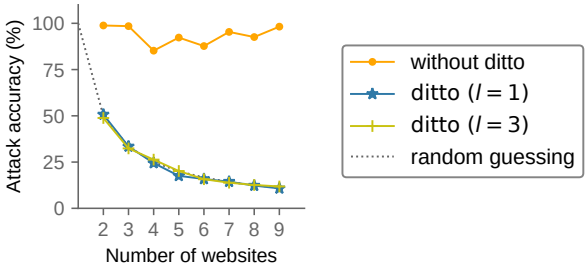


Fig. 10: DF attack accuracy. For `ditto`-protected traffic, the accuracy is on par with random guessing.

In Fig. 9 we show that the hardware prototype indeed performs round-robin scheduling on average and it produces a distribution which is close to uniform. In the plot, we show the Root Mean Square Error (RMSE) between the observed distribution and a uniform distribution. The error does not leak information about real traffic. Instead, it originates from the approximation of the 2-level hierarchical queueing and the required precise rate-control, which is more error-prone for small packets.

State-of-the-art attack does not work with `ditto` After showing that packet sizes and timings are independent from the real traffic and do not leak information in general, we now run a state-of-the-art attack to showcase that `ditto` is robust against this particular attack.

We run the Deep Fingerprinting (DF) attack developed by Sirinam et al. in [92]. This attack uses convolutional neural networks to identify visited websites. The inputs for training, validation and testing are sequences of packet directions (e.g., `[1,-1,-1]` when loading a website required one outgoing packet and two incoming ones).

To run the attack, we used the code published by the authors and the same parameters and input dimensions (1000 samples per website, at most 5000 packet directions per sample). To load the websites, we used the same setup as for the application performance experiment above and we added 5 ms latency between the two containers to make it realistic for real Internet traffic. We loaded the Alexa top 9 websites and recorded the traffic on the link protected by `ditto` using `tcpdump` [18]. In order to be able to record the traffic, we run `ditto` at only 500 Mbps per direction.[5] From the recorded traffic, we extracted the packet directions for 5000 packets starting with the first packet sent from the client to request a website. This makes it easier for the attack as it would be in practice because a real attacker could not distinguish between real and chaff packets and thus she could not determine the first packet of a request. We run the attack in the closed world setting as in [92], where there is no other real traffic besides the loaded website (which makes it easier for the attack) and we run the attack for traffic recordings containing between 2 and 9 websites (identify fewer websites is easier for the attack).

We depict the accuracy of the attack in Fig. 10 with and without `ditto` (each point in the plot is the average accuracy over 20 attack runs). We also depict the accuracy of an attacker randomly guessing (for reference). We see that the attack is unsuccessful on `ditto`-protected traffic: the accuracy is on-par with random guessing. Note that the results hold independently of the pattern length (here, 1 or 3). We also see that the attack is successful without `ditto`, as expected.[6]

[5]The attack would not perform better for a higher bandwidth.

D. Performance and efficiency in future hardware

We now simulate how `ditto` would run on future hardware with two extensions compared to our hardware: *(i)* the round-robin scheduler can directly send a chaff packet if a queue is empty; and *(ii)* the number of padding bytes is not limited.

In the following paragraphs, we show the simulated performance of `ditto` with respect to these two aspects:

- *Overhead:* Amount of padding and chaff packets depending on the input load and the pattern length
- *Reordering:* Out-of-order packets in TCP flows depending on the input load and the pattern length

Overhead We simulate `ditto` with different input loads (10–100 %) and pattern lengths (1–32) to evaluate the overhead added to real traffic. To measure overhead, we use 4 metrics:

- *Chaff overhead:* The number of chaff bytes that `ditto` sends to always transmit at line rate
- *Padding overhead:* The padding that `ditto` adds to packets in order to fit into the pattern
- *Buffer space usage:* The required buffer space to store packets until they fit in the pattern
- *Switching delay:* The number of packets that `ditto` transmits between two packets that arrive subsequently

Overhead depending on the network load Fig. 11 shows how the network load impacts the overhead created by `ditto`.

We show the results for all three datasets. For CAIDA and UNIFORM, we show the results for different pattern lengths (1, 3 and 6). For CONSTANT, we only show a pattern of length 1 because this is already the most efficient pattern. As expected, longer patterns fit the actual traffic distribution better and therefore create lower padding- and chaff overhead.

The *chaff overhead* decreases with increasing input load because the more real traffic there is to send, the fewer chaff packets need to be added to fill the link.

[6]We observed that the model is overfitting in some cases for the unprotected datasets. To limit this, we added an early stopping mechanism that stops the training when there was no significant improvement in the last 3 epochs [5].

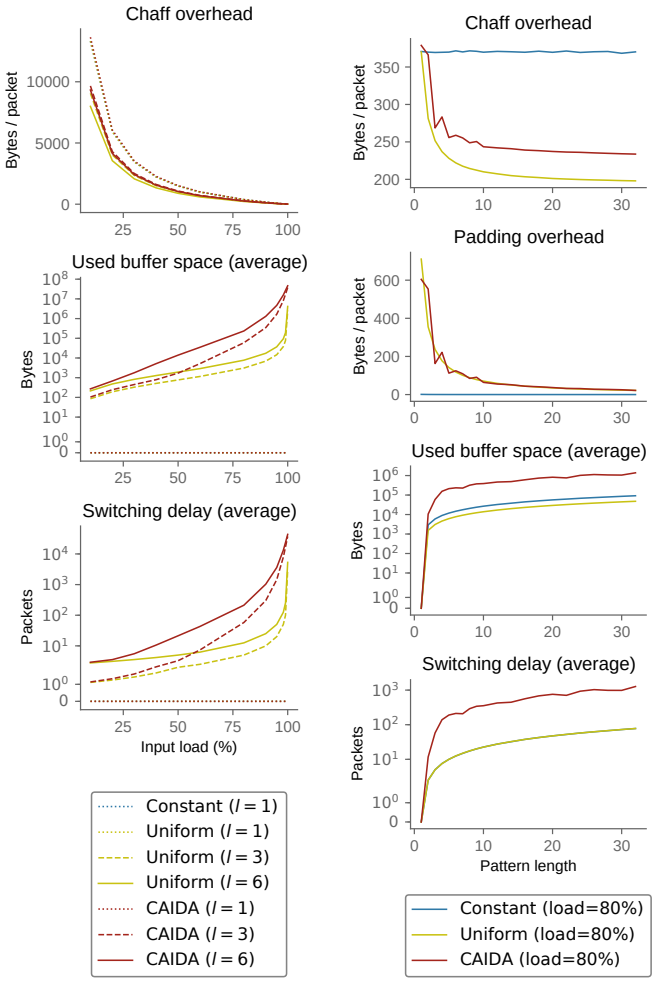


Fig. 11: Overhead depending on the input load. `ditto`'s overhead in terms of buffer space and introduced switching delay is small up to an input load of around 80 %.

Fig. 12: Overhead depending on the pattern length. Long patterns result in small padding and chaff overhead but require more buffer space and introduce more delay.

We observe that the chaff overhead is larger for short patterns. This is because short patterns need to consist of larger packets (e.g., a pattern of length 1 results in sending MTU-sized packets constantly) and therefore, even if the number of chaff packets is smaller, the number of chaff *bytes* is larger.

`ditto` needs to buffer packets when the sequence of the pattern does not match the sequence of the incoming packets. For small loads, this is less critical because there is more time between two subsequent incoming packets (e.g., if the time between two incoming packets is larger than the time it takes to iterate over the pattern once, the buffer is always empty when the new one arrives). For high input loads, discrepancies between the sizes of the incoming packets and the outgoing pattern have a higher impact.

However, as Fig. 11 shows, 1 MB of buffer space is sufficient for up to 90 % (CAIDA) or 99 % (UNIFORM) load. Patterns of length 1 do not require buffering because `ditto` pads and sends each packet immediately.

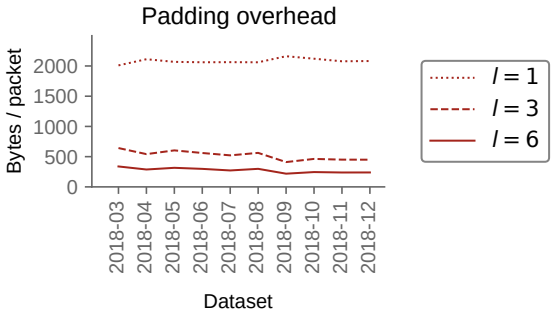


Fig. 13: Overhead for using the same pattern (length l) over 10 subsequent months. The padding overhead does not increase because the traffic distribution roughly stays the same.

The *switching delay* measures the number of packets sent between two incoming packets. Therefore, it evolves similarly to the buffer overhead. Again we observe a slow increase until the switch starts to get congested around 90 % network load.

Overhead depending on the pattern length Fig. 12 shows how the pattern length impacts the overhead. The results are for a high network load of 80 %, which means that there are usually real packets ready to be sent in each pattern state. In this case, longer patterns result in less *chaff overhead* and *padding overhead*. This is because longer patterns fit the actual traffic distribution better and therefore require sending less additional traffic. At the same time, longer patterns lead to increased *buffer usage* and *switching delay* because it is more difficult to fit the incoming packets to the right pattern state.

Overhead for long-term use of a pattern `ditto` computes the pattern based on the packet size distribution and then applies it for future traffic. This is always secure, but not necessarily efficient. If the distribution of the real traffic changes, it is worth computing a new pattern (which can be deployed without interrupting the switch [60]). We confirm with the results in Fig. 13 that `ditto` can apply the same pattern over a long period (10 months) with a nearly constant overhead.

Packet reordering We now evaluate the impact of `ditto` on the ordering of packets. We again simulate `ditto` with different input loads (between 10 and 100 %) and different pattern lengths (1–32).

We focus on CAIDA in this experiment because the other datasets do not contain TCP flows. We randomly select 100k TCP flows with at least 2 packets from CAIDA and count the number of reordered packets for each of them (the sampled flows are the same across all experiments).

To measure reordering, we use the following metrics:

- *Reordered packets:* Packets that were out of order (i.e., packet i arrived before packet $i-1$)
- *Flows with reordered packets:* Flows with at least one reordered packet

Reordering depending on the network load Fig. 14 shows how the network load impacts the reordering. As expected,

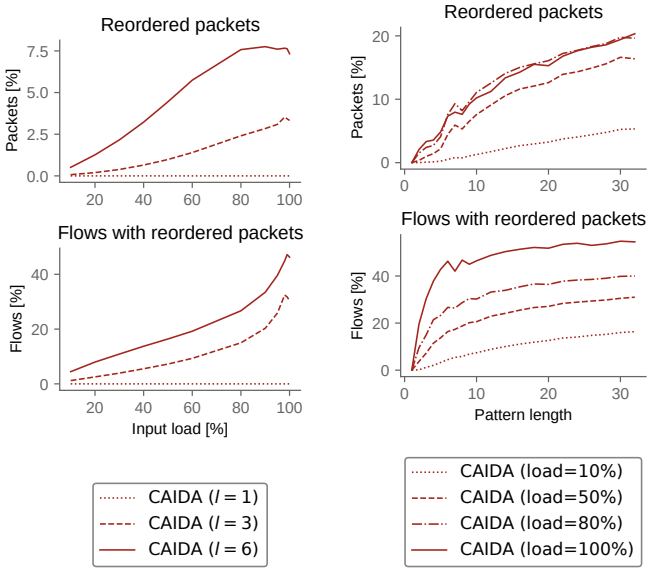


Fig. 14: Packet reordering depending on the input load. 92 % of the packets remain in order for the highest load.

Fig. 15: Packet reordering depending on the pattern length. Longer patterns lead to more reordering.

higher input load leads to more reordering. But even in fully loaded networks, less than 8 % of the packets are reordered. 8 % of reordered packets impact at most 47 % of flows because many short flows are reordered.

We point out that these results show a worst case because of the way how our simulator works and how the dataset was collected. The throughput of the CAIDA dataset as it was captured is around 4 Gbps [39], which corresponds to an input load of 4 % in our simulation. To simulate higher input loads, we replay CAIDA at higher speed (up to 25× the original speed), which creates unrealistically high-bandwidth flows. For example, a user downloading a file with 1 Gbps is simulated as a user with a 25 Gbps connection.

Reordering depending on the pattern length Fig. 15 shows how the pattern length impacts the reordering. As expected, longer patterns lead to more reordering because the sequence of outgoing packets is more constrained.

X. Related work

Preventing traffic-analysis attacks has been an active research area for many years. However, existing work focuses on preventing traffic analysis attacks for Internet users. As we elaborated earlier, these systems are largely orthogonal to `ditto` because protecting WAN traffic presents both new challenges (e.g., high throughput) and opportunities (e.g., control over network devices). Even though existing systems could be applied in WANs too, they would not perform well enough (cf. simulations in §IX) since they are not optimized for this setting or they require modifications of the end hosts.

Most existing work is application-specific (e.g., to prevent website fingerprinting [6], [71], [95]–[97], ensure anonymous communication [32], [70] or protect IoT devices [26]) and/or requires additional servers to relay traffic [19], [84], [98]. In contrast to these approaches, `ditto` operates at the network layer, protects all traffic and does not need additional servers or modifications at the clients. Below and in Table II, we summarize the most relevant work related to `ditto`.

Widely used protocols and libraries already allow adding random number of bytes to the plaintext before encrypting it. Examples include GnuTLS [6], SSH [13], and IPSec [15]. However, this only adds a small amount of anonymity (the volume increases by a random amount within some bounds) and it does not provide timing- or path anonymity.

Onion routing and mix networks TOR [19], [48], the most widely used anonymity network today, and similar systems (e.g., [42], [73], [86] use onion routing [57] to hide the source and the destination of traffic. However, they do not prevent timing attacks and they do not hide the traffic volume.

HORNET [42] is similar to TOR in the sense that it uses onion routing but it operates on the network layer. To obfuscate the traffic volume, HORNET adds padding to packets such that all packets have the same size.

PriFi [32] is based on Dining Cryptographers networks (DC-nets) [41] where each participating node is assigned a time slot in which it can (and must) send a message. This provides volume-, timing- and path anonymity because the observed traffic is always the same, but it reduces the total throughput linearly with the participating nodes.

Loopix [84] mixes real and chaff traffic in dedicated mix nodes and achieves low-latency communication with up to 300 messages per second while hiding the sender and receiver of messages as well as whether they are currently active.

Padding and traffic shaping Like `ditto`, several works aim at hiding the traffic volume by adding padding, chaff packets and/or by sending packets according to a predefined schedule. Examples of such systems include the works of Guan et al. [58], Wang et al. [98] and Wright et al. [100].

In [51], Dyer et al. show that existing padding approaches do not provide enough security and they suggest BuFLO as a solution with high security. BuFLO [51] pads all packets to the same size; delays them such that the time between two packets is constant; and sends chaff packets such that each flow has a certain minimal length. However, as the authors state in the paper, this approach is inefficient because of the constant packet size and inter-packet time. The key difference to `ditto` is the deployment scenario: BuFLO runs on end hosts and obfuscates each flow individually. This leads to the large overhead mentioned in the paper. `ditto` runs in the network and obfuscates traffic on a per-link basis according to an efficient pattern (instead of making all packets constant-size), which leads to less overhead. Furthermore, `ditto` does not leak information about flow durations or sizes.

In [37] and [36], Cai et al. present CS-BuFLO, an improved congestion-sensitive version of BuFLO. CS-BuFLO adapts the transmission rate depending on how much traffic the client tries to send. This makes it more efficient but also less secure because it leaks information about the sender's volume. While CS-BuFLO has less overhead than BuFLO, it suffers from similar limitations: Since the padding happens per flow or per

System	Year	Deployment[1]	Techniques[2]	Volume anonymity	Timing anon.	Path anon.	Throughput[3]	Main overhead / bottleneck
TOR/Onion R. [19], [57]	1999	C/S	P, (C, D,) R	✗	~	✓	100 Mbps	Latency (send via relays)
NetCamo [58]	2001	C/S/N	P, C, R	~	~	✗	N/A[4]	Per-flow padding
Wang et al. [98]	2008	S	P, C, D	~	~	✓	10 Gbps	Per-flow padding, latency (via server)
BuFLO [51]	2012	C/S	P, C, D	~	✓	✗	320 Mbps	All packets have the same size
CS-BuFLO [36], [37]	2012	C/S	P, C, D	~	~	✗	400 Mbps	All packets have the same size
HORNET [42]	2015	C/N	P, R	~	✗	✓	8 Gbps	Onion routing and constant-size packets
WTF-PAD [67]	2016	C/S	P, C	~	~	✗	N/A[4]	Per-flow padding
PriFi [32]	2017	C/S	P, C, D	✓	✓	✓	100 Mbps	Throughput (1 client can send per slot)
Loopix [84]	2017	C/S	P, C, D, R	✓	~	✓	4 Mbps	Per-device obfuscation, computation
TARANET [43]	2018	C/S/N	P, C, D	✗	~	✓	4 Gbps	Per-flow(let) obfuscation
ditto	**2021**	**N**	P, C, D	✓	✓	✓	**100 Gbps**	Switch resources, pattern efficiency

[1] C: Client; S: Server, N: Network [2] P: padding, C: chaff packets, D: delay, R: routing [3] Results from the respective paper. When applicable, we use the following assumptions: throughput for 1 device port or link, 100 Gbps line rate, 1500 B packet or message size, 1000 users or devices, 1 server with a 10 Gbps connection, clients with 1 Gbps connections [4] No throughput measurements in the paper. But the per-flow obfuscation makes the throughput is significantly worse compared to ditto.

TABLE II: Comparison of ditto's key properties with related work. Related work focuses on preventing traffic-analysis attacks on shared links, which adds other constraints compared to ditto and generally results in worse performance.

device, the overhead created by many flows or devices sums up in the network. Further, CS-BuFLO leaks information about the total volume of a flow or device and the sending rate while ditto does not because it runs in the network.

TARANET [43] shapes traffic into constant-rate flowlets at the hosts. The system then makes sure that these flowlets achieve the constant rate despite dynamic network events such as packet loss. Similarly to ditto, TARANET mixes real and chaff packets, but in contrast to ditto, TARANET requires support from the end host.

XI. Conclusion

This paper shows that it is possible to obfuscate volume- and timing properties of wide area network (WAN) traffic directly in the network data plane, using existing hardware, and with a small performance overhead.

ditto mixes real and chaff traffic and it adds padding to packets such that they follow a predefined pattern with respect to packet size and timing.

Two insights allow ditto to achieve high performance (up to 70 Gbps per 100 Gbps switch port for real Internet backbone traffic and interactive applications) and perfect security (observed traffic is independent from real traffic): *(i)* the traffic pattern is efficient because it fits the actual traffic distribution in the protected network; and *(ii)* existing network devices offer the features which are needed to perform packet padding and mixing with chaff traffic at line rate.

Acknowledgements

We are grateful to the anonymous reviewers for their insightful comments and helpful feedback. We also thank Sharat Madanapalli for providing us his framework to generate application traffic. Further, we thank the members of the Networked Systems Group at ETH Zürich and Vladimir Gurevich for their valuable feedback at various stages of this project.

This work was partly supported by armasuisse Science and Technology (S+T) under the Zurich Information Security and Privacy Center (ZISC) grant.

References

[1] Alexa top sites in united states. https://www.alexa.com/topsites/countries/US. (Accessed on 07/21/2020).

[2] Arista 7170 series. https://www.arista.com/en/products/7170-series.

[3] Barefoot networks, google cloud, onf and p4.org to showcase p4 runtime-based control of network switches. https://finance.yahoo.com/news/barefoot-networks-google-cloud-onf-120000850.html. (Accessed on 04/15/2021).

[4] Barefoot networks wins deals from at&t, tencent, alibaba and baidu for programmable switches. https://www.thefastmode.com/technology-solutions/10724-barefoot-networks-wins-deals-from-at-t-tencent-alibaba-and-baidu-for-programmable-switches. (Accessed on 04/15/2021).

[5] Earlystopping. https://keras.io/api/callbacks/early_stopping/.

[6] GnuTLS. https://gnutls.org.

[7] Ikt-systeme der armee. https://www.vtg.admin.ch/de/aktuell/themen/programme-projekte/ikt-systeme-der-armee.html#collapse_id_content_vtg-internet_de_aktuell_themen_programme-projekte_ikt-systeme-der-armee_jcr_content_contentPar_accordion_1. (Accessed on 06/10/2021).

[8] Intel® tofino™ series programmable ethernet switch asic. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series/tofino.html.

[9] iperf. https://iperf.fr/.

[10] P4-16 portable switch architecture (PSA). https://p4.org/p4-spec/docs/PSA-v1.1.0.html.

[11] Passive fiber optic network tap | g-tap m series | gigamon. https://www.gigamon.com/products/access-traffic/network-taps/g-tap-m-series.html. (Accessed on 06/08/2021).

[12] Pjsip. https://www.pjsip.org/.

[13] RFC4253. https://datatracker.ietf.org/doc/html/rfc4253.

[14] RFC4301. https://datatracker.ietf.org/doc/html/rfc4301.

[15] RFC4303. https://datatracker.ietf.org/doc/html/rfc4303.

[16] Scapy. https://scapy.net/.

[17] SWAN scottish wide area network. https://www.scottishwan.com.

[18] Tcpdump & libpcap. https://www.tcpdump.org/.

[19] Tor project. https://www.torproject.org/.

[20] Wide area networks. https://portal.ct.gov/DAS/BEST/Network-Services/Wide-Area-Networks.

[21] IEEE standard for local and metropolitan area networks: Media access control (mac) security. *IEEE Std 802.1AE-2006*, 2006.

[22] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. Peek-a-boo: I see your smart home activities, even encrypted! In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '20. ACM, 2020.

[23] Amazon. Global infrastructure. https://aws.amazon.com/about-aws/global-infrastructure/.

[24] Mikhail Andreev, Avi Klausner, Trishita Tiwari, Ari Trachtenberg, and Arkady Yerukhimovich. Nothing but net: Invading android user privacy using only network access patterns. *arXiv preprint arXiv:1807.02719*, 2018.

[25] APCON. Passive optical tap. https://www.apcon.com/hardware/network-taps/apcon-tap. (Accessed on 06/08/2021).

[26] Noah Apthorpe, Danny Yuxing Huang, Dillon Reisman, Arvind Narayanan, and Nick Feamster. Keeping the smart home private with smart(er) iot traffic shaping. *Proceedings on Privacy Enhancing Technologies*, 2019(3), 01 Jul. 2019.

[27] The Atlantic. The creepy, long-standing practice of undersea cable tapping. https://www.theatlantic.com/international/archive/2013/07/the-creepy-long-standing-practice-of-undersea-cable-tapping/277855/. (Accessed on 04/06/2021).

[28] Georgia Technology Authority. Wan service. https://gta.georgia.gov/gta-services/georgia-enterprise-technology-services-gets/managing-your-gets-services/wan-service. (Accessed on 06/15/2021).

[29] Aviatrix. Is amazon inter-region peering encrypted? https://aviatrix.com/learn-center/answered-access/is-amazon-inter-region-peering-encrypted/.

[30] Microsoft Azure. Backbone networking infrastructure. https://azure.microsoft.com/en-us/global-infrastructure/global-network.

[31] Michael Backes, Goran Doychev, Markus Dürmuth, and Boris Köpf. Speaker recognition in encrypted voice streams. In *European Symposium on Research in Computer Security*. Springer, 2010.

[32] Ludovic Barman, Italo Dacosta, Mahdi Zamani, Ennan Zhai, Bryan Ford, Jean-Pierre Hubaux, and Joan Feigenbaum. Prifi: A low-latency local-area anonymous communication network. *arXiv preprint arXiv:1710.10237*, 2017.

[33] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review*, 44(3), 2014.

[34] Jonas Bushart and Christian Rossow. Padding ain't enough: Assessing the privacy guarantees of encrypted DNS. In *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)*. USENIX Association, 2020.

[35] Arturo Cabanas. Managing security on AWS. https://d1.awsstatic.com/events/Summits/PublicSector2020/Managing_Security_on_AWS_MGMT104_ENGLISH.pdf, 2020.

[36] Xiang Cai, Rishab Nithyanand, and Rob Johnson. Cs-buflo: A congestion sensitive website fingerprinting defense. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, 2014.

[37] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012.

[38] CAIDA. The CAIDA anonymized internet traces dataset (april 2008 - january 2019). https://www.caida.org/data/passive/passive_dataset.xml.

[39] CAIDA. Trace statistics for CAIDA passive oc48 and oc192 traces. https://www.caida.org/data/passive/trace_stats/.

[40] Catapult-project. catapult/web_page_replay_go at master. https://github.com/catapult-project/catapult/tree/master/web_page_replay_go.

[41] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology*, 1(1), 1988.

[42] Chen Chen, Daniele E Asoni, David Barrera, George Danezis, and Adrain Perrig. Hornet: High-speed onion routing at the network layer. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[43] Chen Chen, Daniele E Asoni, Adrian Perrig, David Barrera, George Danezis, and Carmela Troncoso. TARANET: Traffic-Analysis Resistant Anonymity at the Network Layer. *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*, 2018.

[44] Hong-Yen Chen and Tsung-Nan Lin. The challenge of only one flow problem for traffic classification in identity obfuscation environments. *IEEE Access*, 2021.

[45] Cisco. System security configuration guide for cisco 8000 series routers, ios xr release 7.0.x - implementing trustworthy systems. https://www.cisco.com/c/en/us/td/docs/iosxr/cisco8000/security/70x/b-system-security-cg-cisco8000-70x/implementing-trustworthy-systems.html.

[46] B. Coskun and N. Memon. Tracking encrypted voip calls via robust hashing of network flows. In *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2010.

[47] Network Critical. Passive fiber taps. https://www.networkcritical.com/fiber-taps. (Accessed on 06/08/2021).

[48] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[49] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A Ghorbani. Characterization of encrypted and vpn traffic using time-related. In *Proceedings of the 2nd international conference on information systems security and privacy (ICISSP)*, 2016.

[50] R. Dubin, A. Dvir, O. Pele, and O. Hadar. I know what you saw last minute—encrypted http adaptive video streaming title classification. *IEEE Transactions on Information Forensics and Security*, 12(12), 2017.

[51] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE Symposium on Security and Privacy*, May 2012.

[52] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15)*, Tokyo, Japan, 2015.

[53] Don Fedyk. Ethernet-traffic flow security. https://www.ieee802.org/1/files/public/docs2019/new-fedyk-traffic-flow-security-0219.pdf, 2019.

[54] S. Feghhi and D. J. Leith. A web traffic analysis attack using only timing information. *IEEE Transactions on Information Forensics and Security*, 11(8), 2016.

[55] fmadio. 100% line rate 100g packet capture. https://www.fmad.io/products-100G-packet-capture.html. (Accessed on 04/08/2021).

[56] Ya Gao and Zhenling Wang. A review of p4 programmable data planes for network security. *Mobile Information Systems*, 2021, 2021.

[57] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Communications of the ACM*, 42(2), 1999.

[58] Yong Guan, Xinwen Fu, Dong Xuan, P.U. Shenoy, R. Bettati, and Wei Zhao. Netcamo: camouflaging network traffic for qos-guaranteed mission critical applications. *IEEE Transactions on Systems, Man, and Cybernetics*, 31(4), 2001.

[59] The Guardian. Gchq taps fibre-optic cables for secret access to world's communications. https://www.theguardian.com/uk/2013/jun/21/gchq-cables-secret-world-communications-nsa. (Accessed on 06/08/2021).

[60] David Hancock and Jacobus Van der Merwe. Hyper4: Using p4 to virtualize the programmable data plane. In *Proceedings of the 12th International on Conference on emerging Networking EXperiments and Technologies*, 2016.

[61] Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *25th USENIX Security Symposium (USENIX Security 16)*, Austin, TX, 2016. USENIX Association.

[62] Craig Hill and Stephen Orr. Innovations in ethernet encryption (802.1ae - MACsec) for securing high speed (1-100ge) wan deployments. https://www.cisco.com/c/dam/en/us/td/docs/solutions/Enterprise/Security/MACsec/WP-High-Speed-WAN-Encrypt-MACsec.pdf.

[63] Chi-Yao Hong, Srikanth Kandula, Ratul Mahajan, Ming Zhang, Vijay Gill, Mohan Nanduri, and Roger Wattenhofer. Achieving high utilization with software-driven wan. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, 2013.

[64] C. Hopps. Ip-tfs: Ip traffic flow security using aggregation and fragmentation. https://tools.ietf.org/id/draft-ietf-ipsecme-iptfs-06.html, 2021.

[65] North Dakota ITD. Wide area network (wan). https://www.nd.gov/itd/services/wide-area-network-wan.

[66] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, et al. B4: Experience with a globally-deployed software defined wan. *ACM SIGCOMM Computer Communication Review*, 43(4), 2013.

[67] Marc Juarez, Mohsen Imani, Mike Perry, Claudia Diaz, and Matthew Wright. Toward an efficient website fingerprinting defense. In Ioannis Askoxylakis, Sotiris Ioannidis, Sokratis Katsikas, and Catherine Meadows, editors, *Computer Security – ESORICS 2016*, Cham, 2016. Springer International Publishing.

[68] Keysight. Flex tap passive fiber optical taps. https://www.keysight.com/ch/de/products/network-visibility/network-taps/flex-tap-fiber-optical.html. (Accessed on 06/08/2021).

[69] Elie F. Kfoury, Jorge Crichigno, and Elias Bou-Harb. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. *IEEE Access*, 9, 2021.

[70] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A scalable, traffic analysis resistant anonymity network for voip systems. In *ACM SIGCOMM Computer Communication Review*, volume 45. ACM, 2015.

[71] Wen Ming Liu, Lingyu Wang, Pengsu Cheng, Kui Ren, Shunzhi Zhu, and Mourad Debbabi. Pptp: Privacy-preserving traffic padding in web-based applications. *IEEE Trans. Dependable Sec. Comput.*, 11(6), 2014.

[72] Yaqun Liu, Jinlong Zhao, Guomin Zhang, and Changyou Xing. Netobfu: A lightweight and efficient network topology obfuscation defense scheme. *Computers & Security*, 110, 2021.

[73] Jon McLachlan, Andrew Tran, Nicholas Hopper, and Yongdae Kim. Scalable onion routing with torsk. In *Proceedings of the 16th ACM conference on Computer and communications security*, 2009.

[74] Roland Meier, David Gugelmann, and Laurent Vanbever. iTAP: In-network traffic analysis prevention using software-defined networks. In *Proceedings of the Symposium on SDN Research*. ACM, 2017.

[75] Roland Meier, Petar Tsankov, Vincent Lenders, Laurent Vanbever, and Martin Vechev. NetHide: Secure and practical network topology obfuscation. In *27th USENIX Security Symposium (USENIX Security 18)*, Baltimore, MD, 2018. USENIX Association.

[76] Microsoft. Azure encryption overview. https://docs.microsoft.com/en-us/azure/security/fundamentals/encryption-overview.

[77] Sandra Kay Miller. Hacking at the speed of light. Securitysolutions.com, 2006. Apr 1, 2006.

[78] J. Muehlstein, Y. Zion, M. Bahumi, I. Kirshenboim, R. Dubin, A. Dvir, and O. Pele. Analyzing https encrypted traffic to identify user's operating system, browser and application. In *2017 14th IEEE Annual Consumer Communications Networking Conference (CCNC)*, 2017.

[79] Netberg. Aurora 710. https://netbergtw.com/products/aurora-710.

[80] Edgecore Networks. Wedge 100bf-65x. https://www.edge-core.com/productsInfo.php?cls=1&cls2=180&cls3=181&id=334.

[81] Se Eun Oh, Shuai Li, and Nicholas Hopper. Fingerprinting keywords in search queries over tor. *Proceedings on Privacy Enhancing Technologies*, 2017(4), 2017.

[82] OVH. OVH tasks. http://travaux.ovh.net/?do=details&id=10705&, 2014.

[83] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*. ACM, 2011.

[84] Ania Piotrowska, Jamie Hayes, Tariq Elahi, Sebastian Meiser, and George Danezis. The Loopix Anonymity System. *USENIX Security*, 2017.

[85] Profitap. Fiber taps. https://www.profitap.com/fiber-taps/. (Accessed on 06/08/2021).

[86] Michael G Reed, Paul F Syverson, and David M Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected areas in Communications*, 16(4), 1998.

[87] Brendan Saltaformaggio, Hongjun Choi, Kristen Johnson, Yonghwi Kwon, Qi Zhang, Xiangyu Zhang, Dongyan Xu, and John Qian. Eavesdropping on fine-grained user activities within smartphone apps over encrypted network traffic. In *WOOT*, 2016.

[88] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Beauty and the burst: Remote identification of encrypted video streams. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, 2017. USENIX Association.

[89] Tal Shapira and Yuval Shavitt. Flowpic: Encrypted internet traffic classification is as easy as image recognition. In *IEEE INFOCOM 2019-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, 2019.

[90] Naveen Kr Sharma, Chenxingyu Zhao, Ming Liu, Pravein G Kannan, Changhoon Kim, Arvind Krishnamurthy, and Anirudh Sivaraman. Programmable calendar queues for high-speed packet scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2020.

[91] Y. Shi and S. Biswas. Website fingerprinting using traffic analysis of dynamic webpages. In *2014 IEEE Global Communications Conference*, 2014.

[92] Payap Sirinam, Mohsen Imani, Marc Juarez, and Matthew Wright. Deep fingerprinting: Undermining website fingerprinting defenses with deep learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, New York, NY, USA, 2018. Association for Computing Machinery.

[93] Vincent F Taylor, Riccardo Spolaor, Mauro Conti, and Ivan Martinovic. Robust smartphone app identification via encrypted network traffic analysis. *IEEE Transactions on Information Forensics and Security*, 13(1), 2018.

[94] Liang Wang, Hyojoon Kim, Prateek Mittal, and Jennifer Rexford. Programmable in-network obfuscation of traffic. *arXiv preprint arXiv:2006.00097*, 2020.

[95] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *23rd USENIX Security Symposium (USENIX Security 14)*, San Diego, CA, 2014. USENIX Association.

[96] Tao Wang and Ian Goldberg. Walkie-talkie: An efficient defense against passive website fingerprinting attacks. In *26th USENIX Security Symposium (USENIX Security 17)*, Vancouver, BC, 2017. USENIX Association.

[97] Tao Wang and Ian Goldberg. Walkie-talkie: An efficient defense against passive website fingerprinting attacks. In *26th USENIX Security Symposium (USENIX Security 17)*, 2017.

[98] Wei Wang, Mehul Motani, and Vikram Srinivasan. Dependent link padding algorithms for low latency anonymity systems. In *Proceedings of the 15th ACM conference on Computer and communications security*. ACM, 2008.

[99] X. Wang, S. Chen, and S. Jajodia. Network flow watermarking attack on low-latency anonymous communication systems. In *2007 IEEE Symposium on Security and Privacy (SP '07)*, 2007.

[100] Charles V Wright, Scott E Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *NDSS*, volume 9. Citeseer, 2009.

[101] Fan Zhang, Wenbo He, Xue Liu, and Patrick G Bridges. Inferring users' online activities through traffic analysis. In *Proceedings of the fourth ACM conference on Wireless network security*. ACM, 2011.