USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Defending against Traffic Analysis Attacks with Flexible In-Network Obfuscation

Guorui Xie and Qing Li, *Pengcheng Laboratory;* Zhenning Shi, *Tsinghua Shenzhen International Graduate School;* Gianni Antichi, *Politecnico di Milano;* Yijia Zhu, *Xidian University;* Kejun Li and Changxing Weng, *Pengcheng Laboratory;* Sebastiano Miano, *Politecnico di Milano;* Yong Jiang, *Tsinghua Shenzhen International Graduate School and Pengcheng Laboratory;* Mingwei Xu, *Tsinghua University*

https://www.usenix.org/conference/nsdi26/presentation/xie-guorui

This paper is included in the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation.

May 4–6, 2026 • Renton, WA, USA

ISBN 978-1-939133-54-0

Open access to the Proceedings of the 23rd USENIX Symposium on Networked Systems Design and Implementation is sponsored by

جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

Defending against Traffic Analysis Attacks with Flexible In-Network Obfuscation

Guorui Xie[1]* Qing Li[1]† Zhenning Shi[2] Gianni Antichi[3] Yijia Zhu[4]
Kejun Li[1] Changxing Weng[1] Sebastiano Miano[3] Yong Jiang[2,1] Mingwei Xu[5]
[1]*Pengcheng Laboratory* [2]*Tsinghua Shenzhen International Graduate School*
[3]*Politecnico di Milano* [4]*Xidian University* [5]*Tsinghua University*

Abstract

Traffic analysis attacks can exploit side channels in encrypted traffic (e.g., packet sizes) to infer user activities. Existing defenses provide weak protection, impose excessive bandwidth overhead, or require hard-to-deploy coordination. We present Securitas, a novel network traffic obfuscation framework that protects from side-channel attacks using a learning-guided mix of packet fragmentation and insertion. We implemented Securitas on a number of different data planes: Tofino switch, AMD/Xilinx FPGA, eBPF, and BMv2. Experiments show that Securitas reduces attack accuracy by up to 95.89%, while consuming 42.69× less bandwidth than prior defenses. Real-world Internet tests confirm minimal performance impact, e.g., adding 0.15s to the web page load.

1 Introduction

Every day, network traffic carries sensitive user information that can be eavesdropped [11, 41]. Although encryption techniques (e.g., HTTPS [59]) have been widely adopted, they are not sufficient: Attackers can analyze traffic characteristics, such as address, size, or timing [39,55,60,65]. With such information, it is possible to determine what websites a user visited and when, which can in turn be used to mount targeted phishing attacks [45]. Also, attackers can analyze the encrypted traffic to discover IoT devices with known vulnerabilities for surveillance [68], to name a few.

Prior defenses have proposed to *obfuscate* traffic [15,22,27, 40,44,48,50,70,79], making it harder to analyze traffic characteristics (aka side-channel information). These obfuscation approaches range from ones that mainly hide network identifiers/addresses [22,70,71] to ones that also obfuscate, e.g., packet size, direction, or timing [15,27,40,44,48,50,69,74,79]. In this paper, we identify three key requirements for an effective solution: **1)** jointly obfuscate multiple types of side-channel information; **2)** impose low bandwidth overhead; **3)** support flexible obfuscation and de-obfuscation. As detailed in §2.2, existing schemes fail to meet them simultaneously.

The first requirement is motivated by the commoditization of machine learning-based tools that analyze user activities through multiple traffic properties (e.g., packet size, direction, timing) [49, 51, 77]. Especially with deep neural networks (DNN), such analysis yields high accuracy [3, 55], and is now applied in several scenarios, including website fingerprinting [9, 60, 65, 72], IoT fingerprinting [25, 39, 63], and application classification [46, 73, 78].

The second requirement relates to how traffic characteristics are obfuscated. Commonly, this is done by injecting fake packets into the network and padding real packets [27, 40, 48, 50]. As we show later (Figure 3 in §2.2), state-of-the-art (SOTA) solutions generate a substantial amount of extra data, ranging from 100% to 300% of the original traffic, thereby severely impacting application performance.

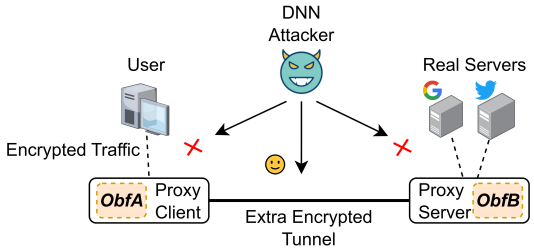


Figure 1: The widely adopted traffic obfuscation framework (*ObfA* and *ObfB* are two instances from the same scheme).

To illustrate the third requirement, we first review how most obfuscation approaches operate [15, 27, 40, 44, 50, 69, 79]. As shown in Figure 1, these systems typically deploy paired obfuscators (*ObfA* and *ObfB*) across two proxies. On the user side, a proxy client obfuscates outgoing requests (e.g., padding packets via *ObfA*) and then forwards them through an encrypted tunnel. The encrypted tunnel encapsulates the traffic to hide the address of the destination server and the obfuscation details. At the server side, a proxy server decapsulates the traffic, de-obfuscates it via *ObfB*, and forwards it

*Work was also done during the visit to Politecnico di Milano.
†Corresponding author.

to the real destination[1]. In this model, the assumption is that an attacker can only observe traffic within the encrypted tunnel: he cannot see the unobfuscated traffic before the proxy client. He may observe the de-obfuscated traffic after the proxy server, but the proxy server surrogates worldwide concurrent users, making it difficult to track the target user's traffic. However, installing a specific *ObfB* at the proxy server is rarely flexible, as reliable proxy infrastructure (e.g., Tor relays [24] or commercial VPN servers [62]) is operated by third-party organizations, where we have no control.

This paper proposes Securitas[2], which aims to satisfy all three requirements and work efficiently on the high-speed data plane. To achieve this goal, we face several challenges: **1)** How to resolve the conflict between requirements #1 and #3. Morphing packets within the encrypted tunnel is effective in hiding multiple types of information. However, such morphing usually requires a reverse operation of another organization (i.e., *ObfB* in the proxy server), resulting in deployment inflexibility. **2)** For requirement #2, although learning-based optimization can improve obfuscation [44, 50], it remains unclear how to establish an end-to-end bandwidth optimization in the practical scenario with multi-constraint, e.g., when the attacker's DNN is unknown, and the obfuscation operations are non-differentiable. **3)** Although Securitas is designed as an in-network service for speed gains, the data plane has limited programming capabilities, which complicates the implementation of current obfuscators. For example, Ditto [48] must occupy six loopback ports and nine priority queues to implement packet padding with only three predefined patterns, severely compromising its applicability. To handle these challenges, **we make the following contributions**:

- We design obfuscation operations based on the well-defined Internet protocols [13]. Using Figure 1 as an example, we make *ObfA* alter the user's outbound traffic by fragmenting real packets and inserting fake packets of small TTLs. The altered traffic then goes through the encrypted tunnel. By default, the real server will reassemble the fragments, and the halfway nodes (e.g., routers between the proxy server and the real server) discard fake packets when TTLs are reduced to zero. These core ideas eliminate *ObfB* and distribute *ObfB*'s functionality implicitly to network nodes and the real visited servers. We also use the ICMP as a supplement. The nodes that discard TTL-expired packets can send back ICMP replies [28]. ICMP replies (may be a small portion) go through the encrypted tunnel, obfuscating the inbound traffic and being dropped by *ObfA*. In this way, we finally obfuscate multiple properties and yield flexibility.
- We design a bandwidth-efficient obfuscation strategy via proxy training. During the proxy training, an agent is trained to output the optimal strategy, deciding which packets to obfuscate. Our proxy training is twofold: First, as the attack DNN is unknown, we use a proxy DNN instead, minimizing both bandwidth overhead and attack accuracy on the proxy DNN. As the proxy DNN can differ from the real attacker's, we also train the agent to minimize the similarity between pre- and post-obfuscation traffic for generalization. Second, though gradient descent is widely used for objective minimization [61], traffic obfuscation operations are not mathematically smooth, resulting in blocked gradient backpropagation. Hence, we utilize the policy gradient from reinforcement learning instead [76].
- We simplify our data plane implementation for applicability. The core idea is to implement both packet fragmentation and insertion in a unified and standardized way. We abstract both operations as follows: first, cloning the original packet, and then processing the original/cloned packets. The fragmentation operation manipulates the fragment-related headers/bytes in both the original and cloned packets, whereas the insertion operation mainly resets the contents in the cloned packet. In this way, Securitas can be implemented with standard modules like packet replication & buffer engine (PRE) and pipelines in P4 [10], being deployable for different data plane devices.

To the best of our knowledge, this is the first P4-based obfuscator that successfully runs on different targets (hardware: Tofino switch [8] and FPGA using AMD/Xilinx Vitis P4 IP [5], software: BMv2 switch [17] and eBPF [30]). We evaluate Securitas by three public traffic datasets [25, 26, 60] and *real-world* Internet tests. Experiments show that Securitas reduces the attacker's analysis accuracy up to 95.89%, having a 42.69× lower bandwidth overhead and a slight increase in web load delay (e.g., 0.15s).

2 Preliminary

In this section, we first discuss the threat model that we consider and then motivate the key requirements that an ideal obfuscator should meet. Finally, we make a case for a new solution, highlighting the challenges associated with it.

2.1 Threat Model

Our threat model, depicted in Figure 2, is mainly based on the currently prevalent work (e.g., [15, 27, 40, 44, 48, 50, 69, 79]) in Figure 1. The **DNN attacker** is positioned between two L3 proxies, observing the traffic information of packet identifiers, sizes, timings, directions, etc. The two proxies are: the proxy client acting as the gateway to forward user traffic, and the proxy server acting as a surrogate for the user's Internet visit. The proxy server is from a reputable vendor (e.g., a commercial VPN company) and works for global users. Thanks to the encrypted tunnel between the proxies, the attacker is unaware of the real traffic identifiers and the obfuscation details. The attacker cannot see the unobfuscated traffic inside the user

[1] Obfuscation is bi-directional, but this example shows user-to-server only.

[2] Roman goddess of security (https://github.com/xgr19/Securitas).

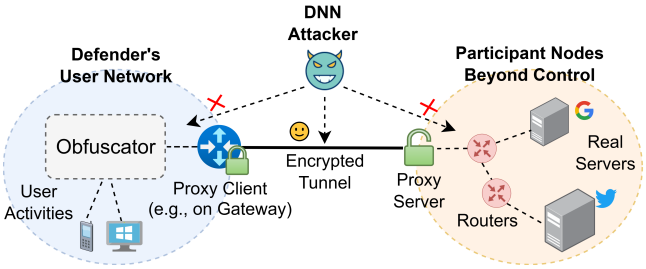


Figure 2: Our threat model (we are similar to current solutions in Figure 1, except we make a more realistic assumption that the right-side nodes *cannot* be modified).

network. Though he may know the traffic after the encrypted tunnel, that traffic has already been mixed by worldwide concurrent users and hinders his analysis.

The **defender** (i.e., the user network administrator) can control nodes in the user network to place an obfuscator. He can also use techniques (e.g., machine learning) to optimize the obfuscator, but unlike [44, 50], he cannot access the attacker's DNN (e.g., using the attacker's accuracy as training feedback is forbidden). Notably, the most significant difference from previous solutions is that we make a more realistic assumption on the right-side **participant nodes**: These nodes on the Internet are of different parties. They may join the defender's obfuscation/de-obfuscation, but cannot be modified.

2.2 Key Requirements

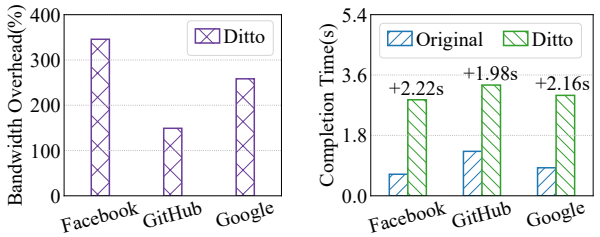
Based on our threat model in Figure 2, we identify three key requirements (i.e., *multiple information obfuscation*, *low bandwidth overhead*, *flexible obfuscation & de-obfuscation*) for an ideal obfuscator. As shown in Table 1, none of the SOTA approaches meet them all.

Multiple information obfuscation. The side-channel information in our threat model refers to the captured sequence of packet sizes, directions, and time intervals, along with the traffic identifiers (e.g., IP addresses) in a user-server session. However, some defenses [22, 70, 71] only hide the traffic identifiers. E.g., SPINE [22] and PINOT [70] assume that each network is configured with both IPv4 and IPv6 (and a unique IPv6 network ID). Their idea is to use programmable Tofino switches [8], replacing IPv4 headers with IPv6 headers and encrypting the original IPv4 addresses as suffixes of the corresponding IPv6 IDs. RAVEN [71] uses similar encryption in QUIC communication, encoding the QUIC connection ID as part of the IPv6 address. The encryption randomizes the traffic identifiers and hinders the attacker from identifying the specific communication. Also, encryption has a lightweight bandwidth overhead, as it generates little extra data.

Putting aside the inflexibility of installing IPv6/QUIC on our uncontrolled networks [1], if the attacker is physically located next to the user's network egress (e.g., outside the

Table 1: Comparison of existing schemes (as shown, none of them meet the three requirements we set).

	Multiple Information Obfuscation	Low Bandwidth Overhead	Flexible Obfuscation & De-obfuscation
RAVEN [71]	✗	✓	✗
SPINE [22]	✗	✓	✗
PINOT [70]	✗	✓	✗
BLANKET [50]	✓	✗	✗
Minipatch [44]	✓	✗	✗
WTF-PAD [40]	✓	✗	✗
Ditto [48]	✓	✗	✗
Minos [74]	✓	✗	✗
Our Securitas	✓	✓	✓



(a) Bandwidth overhead (b) Traffic completion time

Figure 3: Average bandwidth overhead ($\frac{\text{fake traffic}}{\text{original traffic}}$) and traffic completion time ($\frac{\text{whole traffic}}{\text{bandwidth}}$) of Ditto [48].

user network but next to the gateway in Figure 2), he can still capture packets of his desired user regardless of the obfuscated identifiers. Then, through packets' other unobfuscated information, the attacker can conduct multiple eavesdropping activities, ranging from website or IoT fingerprinting [9, 25, 39, 60, 63, 65, 72] to the application classification [46, 52, 73, 78]. Thus, a better obfuscator should alter multiple side-channel properties, not only packet identifiers.

Low bandwidth overhead. There exist solutions (e.g., [40, 44, 48, 50, 74]) that hide multiple information types by inserting fake packets or padding bytes into real packets, and consider the client-server proxy pair to encrypt the identifiers (see Figure 1). Nevertheless, more fake traffic will now compete with the original traffic for the network bandwidth. As such, the fake traffic should be as small as possible to minimize its interference. To understand this well, we discuss below one advanced solution, Ditto [48], which loops every real packet on the Tofino switch to pad bytes and also injects fake packets into the network.

We first selected traffic from popular real-world websites: Facebook (100 traces), GitHub (99 traces), and Google (99 traces). These traces are from [60], reflecting representative user activities such as searching and chatting. Then, we computed Ditto's bandwidth overhead and the corresponding traffic completion time in Figure 3, considering a recommended

home bandwidth of 100Mbps [37]. As shown, Ditto imposes high bandwidth overheads (149% $\sim$ 346% in Figure 3a), resulting in an important traffic interference between application packets and fake traffic. Specifically, the experiments show that the traces require 1.98s $\sim$ 2.22s more time to complete (Figure 3b). This is significant, as a 0.5s delay can kill up to 20% user satisfaction [66]. More results in §7.1 and Appendix D.1 also confirm the unsatisfactory bandwidth overhead/completion time of several SOTA solutions.

Flexible obfuscation & de-obfuscation. As discussed, the framework in Figure 1 enhances the obfuscation by using coordinative operations, e.g., padding packets on one side and recovering them on the other side. Due to the setup encrypted tunnel (e.g., by Tor or VPN proxies), the real traffic identifiers and the used obfuscation details are hidden as well. As the proxy server provides a surrogate for concurrent users, the traffic of mixed worldwide users between the proxy server and the real server is also stealthy, even if it has already been de-obfuscated by the proxy server.

However, such solutions need software (e.g., BLANKET [50], Minipatch [44], WTF-PAD [40]) or hardware (e.g., Ditto [48], Minos [74]) modifications of both sides. This seems impractical on the Internet. As stated in our threat model (§2.1), the node providing reputable proxy services (i.e., the proxy server) is owned by professional third parties, and we have no control over their modifications. Even if the modification is accepted, the obfuscation/de-obfuscation overhead on the proxy server may interfere with its normal surrogate performance. Ideally, we wish the obfuscator to be more flexible, located somewhere we can fully control, while allowing other Internet nodes to join freely and share the obfuscation/de-obfuscation overhead.

The need for a new solution. Because SOTA approaches miss all or some of the three key properties (see Table 1), we are motivated to propose a new solution, Securitas. Similar to previous in-network schemes [22, 48, 70, 71, 74], Securitas is designed to operate as a service on the data plane, enabling us to process packets swiftly and efficiently. Nonetheless, for Securitas to meet all the key requirements and work well on the programming-limited data plane, we face several challenges.

2.3 Challenges

Challenge 1: *How can we avoid ObfB to be more flexible while obfuscating multiple side-channel properties of the bi-directional session?*

As shown in §4, we tackle this based on two conventional Internet behaviors: **1)** When a router splits a packet into smaller pieces for better forwarding, the destination host should automatically conduct the packet reassembly [2]. **2)** Each time a packet is forwarded by a node, its TTL is reduced by 1. When a node receives a packet with TTL = 0, it should discard the packet and notify the source host of the TTL expiration via an ICMP packet [28]. Thus, Securitas can be implemented on a data plane connected to the proxy client, fragmenting packets or inserting packets with small TTLs in the user's outbound traffic. Then, multiple Internet nodes (i.e., the real server, the nodes between the proxy server and the real server) will implicitly de-obfuscate it by packet reassembly or dropping TTL-expired packets, relieving the processing overhead per node. As a supplementary technique, when a TTL-expired packet is dropped, the returning ICMP packet (with the Time Exceeded message) through the encrypted tunnel can partially obfuscate the user's inbound traffic.

Challenge 2: *How can we optimize bandwidth overhead and attack evasion, given the unseen attack DNN and the non-differentiable obfuscation in the real world?*

We aim to selectively morph partial packets per session to minimize bandwidth overhead while deceiving the DNN of the attacker. This idea is borrowed from "adversarial example" [14, 33], which adds small extra data to the input to cheat the DNN. Though a neural agent can be trained like BLANKET [50] or Minipatch [44] to output strategies for the selective obfuscation, they do not consider two practical problems in our threat model: **1)** In the real attack scenario, we cannot know the attack accuracy from the attacker's DNN to optimize our agent. **2)** The obfuscation operations (packet fragmentation and insertion) are non-differentiable, so the gradient backpropagation is blocked [75]. In §5, we solve these by proxy training. First, a proxy DNN representing the attack DNN is used to compute the attack accuracy and train the agent. Meanwhile, to improve the generalization of the agent on unseen attack DNNs, we consider an additional training objective: reducing the similarity of the traffic before and after applying our strategy. For the non-differentiable problem, we estimate the policy gradient [76] directly from the objectives (bandwidth overhead, attack accuracy, and traffic similarity), and then use it to optimize the agent. Through this end-to-end optimization, our agent's output strategy yields a better performance than [44, 50].

Challenge 3: *How can we simplify the obfuscation implementation so it is applicable for multiple data plane devices?*

Emerging data plane devices (e.g., Tofino switches [8]) offer near-nanosecond processing but have multiple programming limitations (e.g., limited memory and no for/while loops), which complicate the existing implementations and weaken their applicability (e.g., Ditto [48]). We tackle this by proposing to abstract the obfuscation operations in a unified and standardized workflow in §6. Regardless of the detailed obfuscations (e.g., packet fragmentation or insertion), we always start by cloning the packet to be obfuscated to get a copy of it. Then, we modify the headers and bytes in both packets according to the operations matched in the trained obfuscation strategy, i.e., making them all fragments or only the cloned packet into the small-TTL one. In this way, all of our obfuscations can be implemented by built-in modules in the P4 language [18] and are well-supported: the packet cloning is done by the PRE, and obfuscation operations in

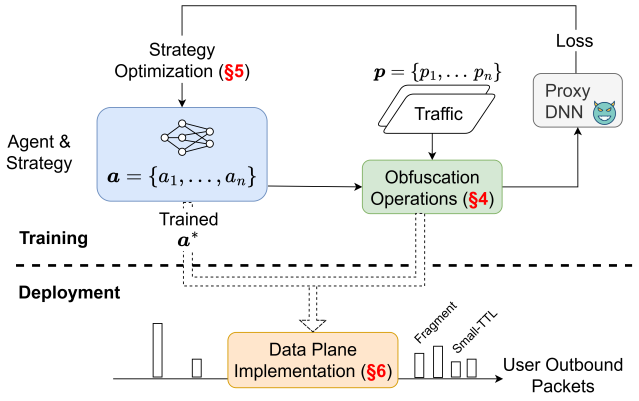


Figure 4: Securitas workflow.

our strategy are converted into match-action tables. Finally, we successfully deploy our system on several software (eBPF, BMv2) and hardware (Tofino, FPGA) data plane targets.

3 Securitas Overview

The overall workflow of Securitas is depicted in Figure 4. Our main **obfuscation operations** (§4) are fragmenting real packets and inserting fake packets with small-TTLs. The former is recovered by the reassembly on the real server, and the latter is discarded by Internet nodes between the proxy server and the real server (see our threat model in Figure 2). When the node discards fake packets because the TTL has been reduced to 0, it will send back ICMP notifications. These ICMP packets go through the encrypted tunnel between proxies, automatically obfuscating the user's inbound (Internet → user) traffic. Nevertheless, we encounter several issues when designing these operations. For example, we should carefully pick the TTL values to distribute the fake packet dropping to multiple Internet nodes for processing overhead relief. And there may be insufficient ICMP packets due to the ICMP rate control [35]. To handle these, we conduct a measurement, configuring the default TTL values (i.e., randomly varying from 1 to 8) and a conservative ICMP reply probability (i.e., 0.48, which also weakens the obfuscation effect of ICMP, see §7.3). Moreover, we invalidate the TCP/UDP checksums when generating the fake packets. Thus, if the real server receives fake packets unexpectedly, it can discard them too.

Our **strategy optimization** (§5) tries to search for an optimal obfuscation strategy. To improve the search efficiency, we parameterize the strategy and train a neural agent. This agent outputs the strategy sequence $\boldsymbol{a} = \{a_1, \ldots, a_n\}$, where integer a_i defines the obfuscation for the corresponding p_i in the packet sequence $\boldsymbol{p} = \{p_1, \ldots, p_n\}$ (a_i is valid iff p_i is a user's outbound packet). When we design $\boldsymbol{a}$, several problems arise. First, $\boldsymbol{a}$ is continuous from a_1 to a_n, but strategies with variable lengths or applied to discontinuous packets may perform better (especially in terms of bandwidth concern). Additionally, the defender may want to adjust the fragment/fake packet ratio based on his network conditions. Hence, we denote two special values, 0 and Pr. The value 0 is used to ignore some a_i for a more flexible strategy, e.g., if $a_i = 0$, we do nothing on p_i. Pr is the probability given by the defender. Then, for a valid a_i, a_i bytes are fragmented from p_i with probability Pr, or a small-TTL packet of a_i bytes is inserted after p_i with probability $1 - Pr$. Notably, when inserting a small-TTL packet, an ICMP packet also probabilistically appears after p_i.

Defining an objective (aka loss) and then using gradient descent to train the agent (minimize the loss) is a common optimization approach [61]. As the real attack DNN is unknown and will not give feedback to help us minimize the attack accuracy, we propose to use a proxy DNN. Given the possibilities of the real attack DNN, specifying an optimal proxy DNN is challenging. So we randomly choose a proxy DNN and turn to optimizing an extra loss instead, i.e., the similarity between $\boldsymbol{p}$ and its obfuscated $\hat{\boldsymbol{p}}$. With less similarity, different attackers may all be misled, and thus our agent is expected to output a generalized strategy. To sum up, we consider the bandwidth overhead, the attack accuracy after $\boldsymbol{p}$ is obfuscated to $\hat{\boldsymbol{p}}$, and the similarity between $\boldsymbol{p}$ and $\hat{\boldsymbol{p}}$ in our loss to facilitate the desired training. Another issue is that, as our obfuscation operations are not mathematically smooth (i.e., non-differentiable), the gradient cannot backpropagate following "proxy DNN → obfuscation operations → agent" by the chain rule [75]. Hence, we use the policy gradient [76] instead. Finally, these two proxy tricks (proxy DNN and policy gradient) form our proxy training for the agent. After the training convergence, we obtain an optimal strategy $\boldsymbol{a}^*$, where each $a_i \in \boldsymbol{a}^*$ now represents a deterministic operation (i.e., either fragmenting or inserting packets with a_i bytes).

Finally, we consider the **data plane implementation** (§6). When using P4 [18] for programming, we face several issues. First, the data plane follows a pipeline paradigm without loops or backtracking, so the order of instructions matters. Second, generating new packets for fragmentation/insertion is challenging because P4 primarily allows simple manipulations of existing packets. To handle these, we arrange the obfuscation as a one-directional logic. That is, first numbering the packets in a user-server session, fragmenting/inserting packets according to $\boldsymbol{a}^*$, and finally forwarding packets to networks. Notably, we unify the fragmentation and insertion by the same packet cloning manner. For a user's outbound p_i, we clone it to obtain its copy p_i'. Then, if a_i is related to a fragmentation operation, we remove $a_i/p_i.size - a_i$ from the head/tail of p_i/p_i' and update packet headers accordingly. If a_i is related to an insertion operation, we remove $p_i.size - a_i$ in the copy packet and reset its headers (e.g., TTL).

4 Obfuscation Operations

This section discusses our operations of packet fragmentation and small-TTL packet generation (with ICMP replies).

4.1 Packet Fragmentation

On the Internet, packets are layered (i.e., Ethernet layer, IP layer, TCP/UDP layer, and application layer in order) [12, 13]. Network nodes, such as routers, mainly operate at the IP layer, verifying the IP checksum and forwarding the packet to the next hop. If the next-hop network has a small maximum transmission unit (MTU), the current network node should perform the fragmentation operation, splitting the packet into smaller pieces for sending.

Taking the IPv4 packet as an example, we split it into two fragments in Figure 5. As shown, after being assigned x or $y-x$ bytes from the original packet, each fragment should update its related IP headers to be legal again. For **Fragment1**, its total IP layer length is $totalLen = z+x$. As it is not the last fragment, its more-fragment flag (MF) is 1. Its IP checksum is also recomputed according to the updated IP headers above. In **Fragment2**, besides the updated $totalLen = z+(y-x)$ and IP checksum, its $offset$, which indicates the location of its payload in the original packet, should be updated too. The IPv4 offset is in units of 8 bytes, so here $offset = x/8$. As Fragment2 is the last fragment, its $MF = 0$.

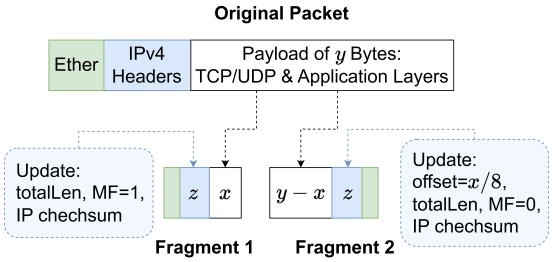


Figure 5: Packet fragmentation and IPv4 header update ($z = 20$ is the number of IPv4 header bytes; fragmentation is bandwidth-efficient as payload bytes are reusable).

In Securitas, we use the same procedure as Figure 5 to morph packets (due to the limited P4 programmability, we consider splitting one packet into only two pieces). As for how many bytes (i.e., x) are fragmented from the original packet, it is decided by the strategy in §5. When the destination node receives our fragments, it will automatically reassemble them according to the $offset$ in Fragment2 and verify the reassembled payload: If the payload has a valid TCP/UDP checksum, it is passed to the upper application. Otherwise, the packet is discarded silently.

4.2 Small-TTL Packet and the ICMP Reply

When a packet is sent out across the Internet, there is a risk that it will continue to pass among routers indefinitely. To mitigate this, packets are assigned a value called time-to-live (TTL). When a router receives a packet, it decrements the TTL by 1 and then forwards the packet to its next hop. However, if the decremented TTL reaches 0, the router will discard the packet and create a message to notify the sender of this TTL expiration. The message is in the ICMP format [28] and contains 28 bytes from the original packet (20-byte IP headers plus 8-byte payload).

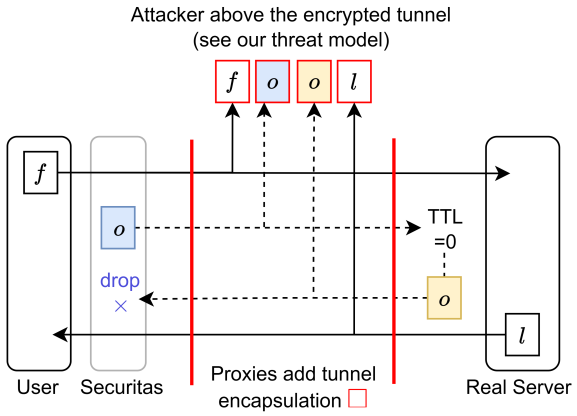


Figure 6: Securitas sends a small-TTL packet (the left "o") and discards the returned ICMP packet (the right "o"). The proxies add/remove tunnel encapsulation to hide the details.

We thus suggest creating fake packets of different sizes (sizes are decided by the strategy in §5) and small TTLs, actively triggering the TTL expiration for obfuscation. If the sent-back ICMP packets (due to TTL expiration) are received, we simply discard them. An intuitive example is shown in Figure 6, where a user and a server communicate using two packets, but the attacker receives four packets. The extra two packets are the fake packet (sent by Securitas) and the returned ICMP reply when the TTL of the fake packet is 0. However, the attacker is unaware of these packet details because they are hidden by the encrypted tunnel. Actually, this tunnel is established through packet encapsulation by our proxy client and proxy server (see §2.1). Such encapsulation also wraps the packet with an extra IP layer (with a normal TTL). Thus, our small-TTL trick only works after the proxy server decapsulates the packet.

To make this obfuscation operation practical, there are several problems. First is how to select the TTL values. Ideally, we want fake packets to be dropped by multiple nodes/routers, relieving the processing overhead per node. Hence, we measure a week-long traceroute [23] of the hop counts for popular servers from [60]. After analyzing the measurement results (detailed in Appendix A), we find that TTL from $1 \sim 8$ can include many Internet nodes for the packet dropping. Second, since the routing path to servers is not always fixed, the fake packets may unexpectedly arrive at the real server. Thus, we also invalidate the TCP/UDP checksums of the fake packets so that the real server can drop them via its TCP/UDP checksum validation.

Moreover, as ICMP rate control may exist in different autonomous systems, the measurement results in Appendix A reveal that Internet nodes/routers (before hop # 8) send ICMP

packets with a probability ≥ 0.48. Thus, we assume that when a TTL is reduced to zero on a router, the router sends an ICMP reply with a probability of 0.48[3]. Another two assumptions are: Due to the line-rate processing of the data plane, the fake packet is sent immediately after its predecessor (no time interval); The time we receive the ICMP packet is a random value between the time we send the corresponding fake packet and the time we receive the next packet from the real server.

5 Obfuscation Strategy Optimization

This section first introduces our obfuscation strategy, and then the way to train it for both bandwidth and attack evasion gains.

5.1 Strategy Design

As discussed, the ICMP packets are automatically (and probabilistically) sent by the Internet nodes. Therefore, we mainly reflect fragmentation and small-TTL packet insertion in the strategy. For a packet sequence $\boldsymbol{p} = \{p_1, \ldots, p_n\}$, we define the strategy as an integer sequence of $\boldsymbol{a} = \{a_1, \ldots, a_n\}$, where a_i indicate splitting p_i into two pieces of a_i and $p_i.size - a_i$ bytes or inserting a packet of a_i bytes after p_i.

Given these, the first problem is to decide the related operation per a_i. By using packet fragmentation, we benefit from its low bandwidth overhead as most bytes are from the original packet (see Figure 5). Nonetheless, reassembling too many fragments may burden the server side. Although inserting small-TTL packets increases the bandwidth requirement, it enables multiple nodes on the routing path to join the obfuscation/de-obfuscation, thereby distributing the processing overhead. Hence, we allow the defender to assign ratios of these two operations as needed flexibly. That is, with a predefined hyperparameter Pr, packet fragmentation and insertion occur with probabilities of Pr and $1 - Pr$, respectively. Another issue is that $\boldsymbol{a}$ has a fixed length n, covering all packets in $\boldsymbol{p}$. However, $\boldsymbol{p}$ maintains packets in both directions (user to server or vice versa), but packet fragmentation/insertion is only applied to the user's outbound traffic. Additionally, even in the outbound direction, not all packets should be obfuscated, as this may reveal high bandwidth overhead. Thus, we further refine the strategy with the two techniques below. First, we check the direction of p_i, and a_i is applied only if p_i is an outbound packet of the user. Otherwise, a_i is ignored. Second, a_i can be a special value of 0, which indicates that we take no action for its corresponding p_i. The special 0 value allows us to process packets flexibly, e.g., processing partial and discontinuous p_i.

Finally, the above ideas are reflected in the function $\Phi(\boldsymbol{p}, \boldsymbol{a})$ as Algorithm 1. Lines $3 \sim 5$ are our ignored cases, i.e., $a_i = 0$ or the corresponding p_i is sent back from the server to the user. Then, Lines $6 \sim 11$ implement the obfuscation operation

[3] As TTL and ICMP settings change in different networks, the defender may repeat such measurements.

Algorithm 1 Obfuscation Function $\Phi(\boldsymbol{p}, \boldsymbol{a})$

Input: Defender's preference $Pr \in (0, 1.0)$.
Output:Return processed $\boldsymbol{p}$ as $\hat{\boldsymbol{p}}$.

```
1:  Random.seed = 0.                          # Fixed for better training
2:  for a_i ∈ a do
3:      if a_i is 0 or p_i is inbound then
4:          Continue.                         # Jump to next a_i.
5:      end if
6:      rnd = Random().
7:      if rnd < Pr then
8:          Replace p_i by splitting it into two fragments of a_i
            and p_i.size − a_i bytes.         # See Figure 5.
9:      else
10:         Insert a fake packet with a small TTL and a_i bytes
            immediately after p_i.
11:     end if
12: end for
13: if training then
14:     Insert ICMP packets somewhere after the small-TTL
        packets according to §4.2.            # Only used in §5.2.
15: end if
```

selection according to the defender's Pr. As the random seed is fixed in Line 1, the obfuscation related to each $a_i \in \boldsymbol{a}^*$ is somewhat deterministic. Note that we simulate the ICMP packets generated by the Internet nodes in Lines $13 \sim 15$ for the training in next §5.2. This simulation follows our real-world measurement results in §4.2. With $\Phi(.)$ and the attacker's DNN f, we then want to find the optimized strategy that minimizes attack accuracy and bandwidth overhead:

$$\boldsymbol{a}^* = \arg\min_{\boldsymbol{a}} \text{Accuracy}(f(\Phi(\boldsymbol{p}, \boldsymbol{a}))) + \text{BandwidthOverhead}(\Phi(\boldsymbol{p}, \boldsymbol{a})). \quad (1)$$

5.2 Agent Training

Randomly searching $\boldsymbol{a}^*$ is difficult, because there are exponential M^n combinations when $a_i \in [0, M], \boldsymbol{a} = \{a_1, \ldots, a_n\}$. Hence, we suggest parameterizing $\boldsymbol{a}$ and using Stochastic Gradient Descent (SGD [61]) for its efficient optimization. To do so, a trainable recurrent LSTM agent:

$$\begin{aligned} out, h &= \text{LSTM}(a_{i-1}, h), \\ P(a_i | a_1, \ldots, a_{i-1}) &= \text{softmax}(out), \end{aligned} \quad (2)$$

is used to output $\boldsymbol{a}$, where h is the hidden state maintained by the LSTM when generating a_i, and softmax(.) [31] is to converts $out \in \mathbb{R}^M$ into a probability distribution vector over integers $\{0, \ldots, M\}$. LSTM explores a_i by sampling these candidate integers according to this probability vector. And the first a and h are zeros when LSTM generates $\boldsymbol{a}$.

To train the LSTM with SGD, we define a training loss ℓ

to reflect our objective in Eq. 1:

$$\ell = \text{Confidence}(f(\hat{\boldsymbol{p}})) + \text{ByteRatio}(\boldsymbol{p}, \hat{\boldsymbol{p}}), \tag{3}$$

where $\hat{\boldsymbol{p}}_i = \Phi(\boldsymbol{p}_i, \text{LSTM}(0,0))$. Confidence(.) returns the confidence score (predicted by attacker f) of the true class label for $\boldsymbol{p}_i$, and $\text{ByteRatio}(\boldsymbol{p}_i, \hat{\boldsymbol{p}}_i) = \frac{\hat{\boldsymbol{p}}_i.bytes - \boldsymbol{p}_i.bytes}{\boldsymbol{p}_i.bytes}$ evaluates the ratio of extra packet bytes after $\boldsymbol{p}_i$ is obfuscated. When ℓ is reduced in the training, the attack accuracy and the bandwidth overhead are optimized. Nevertheless, two new problems arise. First of all, the attack DNN of a real attacker is unknown, which hinders our confidence computation. Second, when generating $\hat{\boldsymbol{p}}_i$, $\Phi(.)$ (detailed in Algorithm 1) is not mathematically differentiable. Thus, the widely used chain rule [75] is blocked when the gradient is backpropagated to update the LSTM.

To tackle these problems, we use a *proxy training* manner: We use a proxy model f' to compute the confidence. However, a side effect is that if f' is selected improperly, the obtained $\boldsymbol{a}^*$ may have less generalization as f' is likely to be different from the real attacker f. But due to the attacker agnosticism, selecting f' is challenging. Finally, we use a trick — randomly selecting f' but modifying ℓ to include the traffic similarity:

$$\begin{aligned}\ell =& \text{Confidence}(f'(\hat{\boldsymbol{p}})) + \text{ByteRatio}(\boldsymbol{p}, \hat{\boldsymbol{p}}) \\ &+ \text{Similar}(\boldsymbol{p}, \hat{\boldsymbol{p}}),\end{aligned} \tag{4}$$

where $\text{Similar}(\boldsymbol{p}_i, \hat{\boldsymbol{p}}_i) = \frac{|\hat{\boldsymbol{p}}_i - \boldsymbol{p}_i|_2}{|\boldsymbol{p}_i|_2}$ is the normalized euclidean distance. As Similar(.) value decreases in the training, $\hat{\boldsymbol{p}}_i$ further diverges from $\boldsymbol{p}_i$, which can also lead the unknown attacker to make wrong predictions.

As for the blocked gradient update, we derive the policy gradient [76] from ℓ instead to train the LSTM's parameters θ. Given the probabilistically sampled $\boldsymbol{a}$ and multiple $\boldsymbol{p}$ in the training dataset, we minimize the loss expectation $\mathbb{E}[\ell]$ by descending θ via the policy gradient ∇_θ:

$$\begin{aligned}\theta &= \theta - \nabla_\theta \mathbb{E}[\ell] \\ &= \theta - \nabla_\theta \sum_{\boldsymbol{p}} \sum_{\boldsymbol{a}} \ell_{\Phi(\boldsymbol{p},\boldsymbol{a})} \cdot \Big(\prod_{i=1}^{n} P(a_i | a_1, \ldots, a_{i-1})\Big).\end{aligned} \tag{5}$$

Furthermore, the above equation can be efficiently approximated by the batched training:

$$\theta = \theta - \lambda \frac{1}{B} \sum_{i=1}^{B} \ell_{\Phi(\boldsymbol{p}_i,\boldsymbol{a}_i)} \cdot \nabla_\theta \log(P_{\boldsymbol{a}_i}), \tag{6}$$

where λ is the learning rate, B is the number of training packet sequences in every training batch (i.e., the batch size). For each $\boldsymbol{p}_i$, there is a sampled $\boldsymbol{a}_i$ from the LSTM model, and $P_{\boldsymbol{a}_i} = \prod_{i=1}^{n} P(a_i | a_1, \ldots, a_{i-1})$ is the correlated sampled probability of $\boldsymbol{a}_i$ whose length is n. Formally, the packet sequences in the training dataset are divided into batches of size B, and we continuously update θ with these enumerated batches. Once the LSTM model is converged, we can run it by LSTM$(0,0)$, obtaining the optimized $\boldsymbol{a}^*$.

6 Data Plane Implementation

In the past, people needed specific knowledge for programming on different devices, e.g., $P4_{14}$ for Tofino switch [10], BCC for eBPF [58]. Fortunately, with the emerging $P4_{16}$ language [18], we can now conveniently install Securitus on multiple targets. Nonetheless, $P4_{16}$ (P4 for short) programming has several limitations. First, a standard Portable Switch Architecture (PSA [34])-based P4 program consists of three modules: *ingress pipeline*, *packet replication & buffer engine (PRE)*, and *egress pipeline*. P4 lacks efficient instructions to route/loop packets among these modules, so packets must be processed by these modules in order. Second, P4 only allows simple instructions for its high-speed processing. Hence, it is impossible to actively generate packets of totally new and arbitrary content via P4. To tackle these, we abstract the new packet generation requirements in fragments and fake packets as the packet cloning in the PRE. Then, we unroll the loop of Algorithm 1 into a one-directional process as shown in Figure 7, i.e., from the ingress pipeline to the PRE, and finally to the egress pipeline (see Appendix B and B.1 for more details).

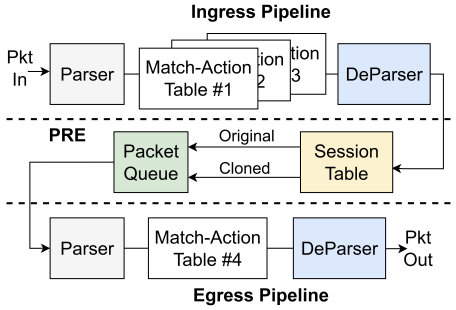


Figure 7: P4 implementation of Securitas.

Ingress Pipeline: The ingress parser is programmed to parse predefined headers (e.g., the five-tuple) of an input packet. Then, the parsed headers are sent to the subsequent customized *three* match-action tables: Table #1 has no entries, and its default action is to count the number i of packets per user-server session, using the five-tuple in the form of {user IP, user port, server IP, server port, protocol}. Table #2 uses two metavariables (i.e., packet direction and count value i) as keys to match table entries and trigger operations. If the packet is a user's inbound one, Table #2 does nothing and jumps to the ingress deparser. Otherwise, i will index an a_i which relates to an obfuscation operation a_i. As discussed, $a_i = 0$ also triggers jumping to the ingress deparser. When $a_i \neq 0$: If a_i relates to a fake packet insertion operation, Table #2 sets the packet's metavariable $Case = 1$. If a_i indicates a

fragmentation operation, Table #2 sets the packet's $Case = 2$ (in this case, if $a_i \geq packet.size$, we will fail to fragment the packet and roll back to $Case = 1$ instead). Then, for the packet with $Case = 1$ or $Case = 2$, Table #3 uses a_i as the key, assigning the packet with a *SessionID* (used later for packet cloning). Finally, the ingress deparser reorganizes headers and metavariables into the packet and sends it to the PRE.

PRE: We install a required session table consisting of $(SessionID, ClonePktLen)$ pairs to the PRE. If a packet matches some *SessionID*, the PRE will automatically clone an additional instance of it. The PRE also truncates the cloned packet according to the *SessionID*'s correlated length *ClonePktLen*. We set each *ClonePktLen* equal to a specific a_i. This is because Lines 8 and 10 of Algorithm 1 need new packets of a_i bytes. As packets are handled one by one, the original and cloned packets (with metavariable $Type = Origin/Cloned$) will be buffered in the PRE's queue before entering the egress pipeline.

Egress Pipeline: The egress parser is programmed to parse packet headers and metavariables. Then, the parser triggers further parsing based on two parsed metavariables (*Case* and *Type*): If this is the fake packet case and the packet is cloned (i.e., $Case = 1, Type = Cloned$), the parser further parses the TCP/UDP checksum of this packet; If this is the fragment case and the packet is the original one (i.e., $Case = 2, Type = Origin$), the parser parses and discards the first *ClonePktLen* (equal to a_i) payload bytes of the packet, because we want to obtain a fragment of $p_i.size - a_i$ bytes for Line 8, Algorithm 1; Otherwise, no more parsing is required. After the egress parser, we allocate Table #4 to update the parsed data: **1)** For the packet of $Case = 1, Type = Cloned$, we process it to be fake by assigning it a random TTL and a wrong TCP/UDP checksum. **2)** For the fragmented packet ($Case = 2, Type = Cloned/Origin$), we update its related IP headers (see Figure 5). Notably, the tail of the cloned packet ($Type = Cloned$) is truncated by the PRE, so we process it to be the first fragment, and the corresponding original packet ($Type = Origin$) is the second fragment. **3)** For packets that do not meet the conditions above, we do nothing. Finally, the egress deparser reorganizes the processed headers back to the packet and forwards it out. Our obfuscation completes here, without the complicated packet recirculation process in [48].

7 Evaluation

In this section, we first compare Securitas with four SOTA obfuscators on datasets and synthetic traffic (§7.1). Then, we discuss the performance of obfuscation on both hardware and software with real-world traffic (§7.2). Finally, §7.3 discusses Securitas itself in depth. If not specified, the default settings in Securitas are: $Pr = 0.7$ (the probability of fragments); $M = 88$ (the max number of bytes per newly generated packet); learning rate $\lambda = 0.001$ and batch size $B = 256$. More experimental settings and results are in Appendix C and D, respectively.

7.1 Comparison with Baselines

Dataset results: In this experiment, we compare Securitas with four obfuscators: Minipatch [44], BLANKET [50], WTF-PAD [40], and Ditto [48] using Python. For the comparison, we consider three attack scenarios (9 attack DNNs in total) with three public datasets: *Web fingerprinting* (attacker's DNN: DF [65], AWF [60], and Var-CNN [9]) on traffic of websites from the *CW100* dataset [60]; *IoT fingerprinting* (attacker's DNN: ANN [63], LSTM and BILSTM are both from [25]) on traffic of seven IoT devices in the *HomeMole* dataset [25]; *Application classification* (attacker's DNN: App-Net [73], FS-Net [46], and Transformer [78]) on traffic of seven types from the *ISCX* dataset [26]. The length of packet sequences in the datasets is padded to a fixed number (e.g., 2K). We respectively use DF, ANN, and App-Net as the proxy DNN f' to train Securitas per attack scenario. The training of these f' follows the instructions in their papers. And *Securitas** is the ideal case where we use the exact attack DNN to train Securitas.

Table 2 depicts the attackers' accuracy before and after applying the obfuscators. As shown, Securitas* and Securitas effectively protect the traffic. Especially on the website and IoT fingerprinting scenarios, Securitas* and Securitas make attackers have the first and second lowest accuracy, respectively. For example, with Securitas, Var-CNN's accuracy is from 99.40% to 3.51% (reduced by 95.89%). This also significantly reveals the generalization of Securitas as it is trained on the proxy DNN (DF). Although our solution is slightly weaker than some obfuscators, such as Ditto, in the application classification scenario, Table 3 shows that Securitas* and Securitas outperform all obfuscators in terms of bandwidth overhead. E.g., Securitas has a 42.69× lower bandwidth than Ditto in the application classification (6.21% vs. 265.11%). In our opinion, the main reason for Securitas' superiority is our well-formatted obfuscation strategy optimization (§5), which uses an end-to-end training to optimize attack accuracy, bandwidth overhead, and generalization together.

More comparisons with Ditto: As Securitas is implemented on the data plane, its processing is obviously more efficient than those acting as browser plugins [40, 44, 50]. Thus, we here mainly compare Securitas with the SOTA in-network obfuscator, Ditto [48]. The compared obfuscator instances are from the web fingerprinting scenario, and the synthetic traffic is based on the CW100 dataset. Our tests are on a Tofino testbed in Figure 8 (see Appendix C.2 for details).

To evaluate the maximum processing throughput, TRex [67] in Server1 first sends simulated "real" traffic of $20 \sim 100$Gbps (based on the packet size distribution of the dataset) to Switch1. Then, the real traffic mixed with the fake obfuscation traffic is forwarded to Switch2. Finally, TRex records the received de-obfuscated real traffic from Switch2. Figure 9a shows the real traffic throughput. The highest throughput TRex received is 98.19Gbps ("No De-

Table 2: Attackers' accuracy before (i.e., No Defense) and after we apply the five obfuscators.

	Website Fingerprinting			IoT Fingerprinting			Application Classification		
	DF	AWF	Var-CNN	ANN	LSTM	BILSTM	App-Net	FS-Net	Transformer
No Defense	98.78%	98.65%	99.40%	90.20%	83.30%	86.53%	78.80%	79.80%	72.60%
Minipatch	8.13%	10.59%	8.94%	47.96%	40.12%	41.76%	73.49%	67.43%	38.62%
BLANKET	14.42%	17.14%	9.84%	44.50%	48.72%	50.85%	19.20%	**16.20%**	24.60%
WTF-PAD	18.98%	9.89%	14.01%	29.85%	32.62%	32.30%	**16.66%**	16.66%	20.95%
Ditto	3.77%	2.36%	3.71%	25.15%	23.40%	28.76%	22.40%	22.80%	**8.20%**
Securitas	3.01%	1.96%	3.51%	23.49%	25.62%	24.27%	31.20%	37.80%	16.8%
Securitas*	**3.01%**	**0.75%**	**2.71%**	**23.49%**	**22.58%**	**20.07%**	31.20%	34.00%	14.40%

Table 3: Obfuscators' bandwidth overhead (WTF-PAD and Ditto have the same results per attack scenario because their defenses are dataset not attacker-related; Securitas has the same results as it is trained on one proxy DNN per attack scenario).

	Website Fingerprinting			IoT Fingerprinting			Application Classification		
	DF	AWF	Var-CNN	ANN	LSTM	BILSTM	App-Net	FS-Net	Transformer
Minipatch	3.39%	2.98%	2.63%	21.70%	41.77%	43.12%	129.36%	310.34%	279.20%
BLANKET	26.45%	23.35%	26.61%	25.25%	25.14%	25.13%	21.92%	21.90%	21.90%
WTF-PAD	29.09%	29.09%	29.09%	133.15%	133.15%	133.15%	257.33%	257.33%	257.33%
Ditto	92.13%	92.13%	92.13%	186.57%	186.57%	186.57%	265.11%	265.11%	265.11%
Securitas	3.31%	3.31%	3.31%	4.15%	4.15%	4.15%	6.21%	6.21%	6.21%
Securitas*	**3.31%**	**3.30%**	**3.29%**	**4.15%**	**4.06%**	**4.08%**	**6.21%**	**6.11%**	**6.06%**

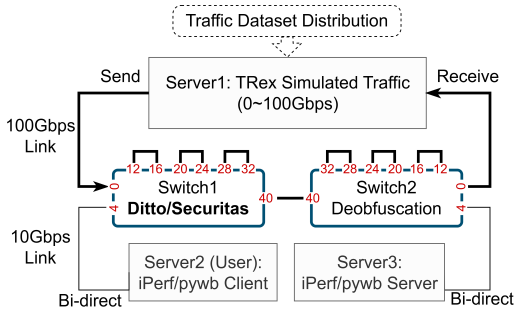
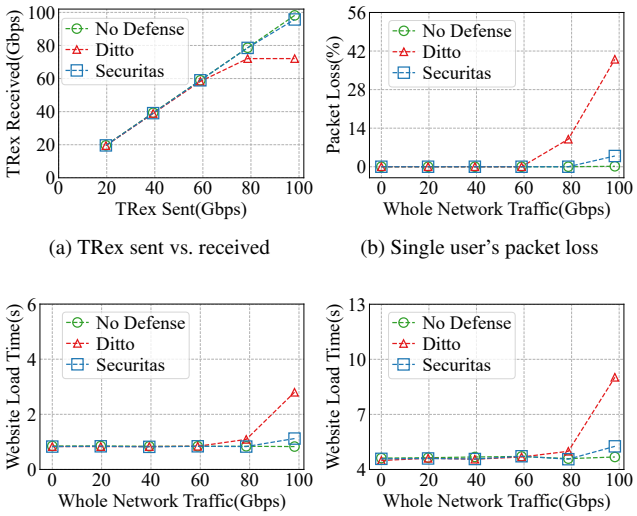


Figure 8: Tofino switch testbed (the complicated paired ports 12 $\sim$ 32 are only required by Ditto).



(a) TRex sent vs. received (b) Single user's packet loss

(c) Single user's Google load time (d) Single user's iQIYI load time

Figure 9: Tofino switch testbed results (a better obfuscator should have performance closer to "No Defense").

fense") and then 95.71Gbps (Securitas). However, Ditto only reaches 72.07Gbps. The main reason is that Ditto loops almost every packet for padding and constantly injects new fake packets, dramatically reducing its efficiency.

To test the user experience, we use Server2 as the user to visit Server3. By iPerf [42], we collect the user's packet loss under different speeds of background traffic (sent by TRex) in Figure 9b. We also snapshot two representative web servers: Google (with 15 HTML files and 25 images), and iQIYI (with 6200 HTML files and 170 images). They are replayed by pywb [38] on Server3. Figure 9c and 9d are the average web load times on Server2. According to these results, Securitas performs more similarly to the network without an obfuscator (i.e., "No Defense") and thus has less disruption to the user than Ditto. For example, when the whole network traffic reaches around 100Gbps, the website load times in Figure 9d of "No Defense" and Securitas are 4.68s and 5.26s, respectively. However, it costs Ditto 9.02s.

7.2 Real-Wold Tests

Real-world tests with Tofino: Unlike previous solutions that mainly report simulation results, due to Securitas' flexible obfuscation, we can explore its performance on the Internet. To do so, we modify our Tofino switch testbed according to our threat model in Figure 2 (detailed in Appendix C.3). Then, Server1 runs Locust [36] to simulate 30 concurrent users browsing 26 global web servers through a commercial proxy server (NordVPN [62]) in the USA. The web requests go through a path as follows: "Server1 $\rightarrow$ Switch1 $\rightarrow$ Server2

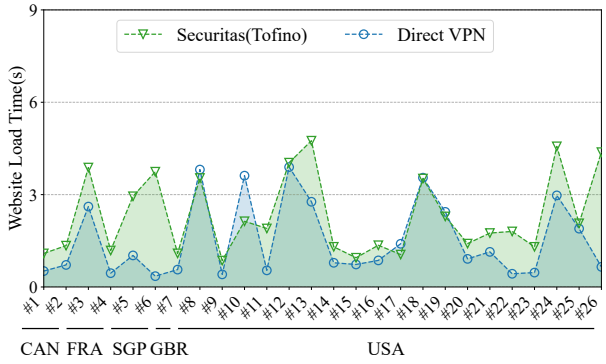


Figure 10: Average web load time before (Direct VPN) and after Switch1 installs Securitas (the 26 web servers from [60] are hosted in Canada, France, Singapore, Britain, and USA).

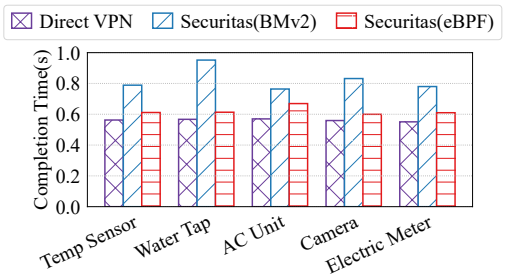
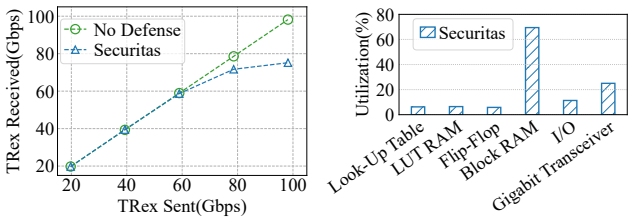


Figure 11: Traffic completion time of IoT tasks between the IoT devices and the commodity IoT hub ("Direct VPN" indicates that IoT devices connect to the hub without Securitas).

(proxy client [47]) → NordVPN → Internet nodes and final destination". Figure 10 shows the web load time on Server1. Thanks to the low bandwidth overhead, Securitas has a slight interference with the web performance, e.g., the web load time of Web#12 is 3.89s and 4.04s before and after we apply Securitas (i.e., increased by 0.15s). Besides, according to our traceroute on all web servers, the number of distinct hops and ASes we passed is 112 and 15, which significantly reveals that our obfuscated traffic can be forwarded legally by different organizations on the Internet.

Real-world tests with BMv2 and eBPF: Azure publishes its IoT APIs [6] to allow us to communicate with the off-the-shelf Azure IoT hub. To study how Securitas impacts the IoT activities in the IoT fingerprinting scenario, we build a software testbed (based on BMv2 or eBPF, see Appendix C.4) to connect to NordVPN and then the European Azure IoT hub. In this test, we simulate five IoT devices and record their task completion time (e.g., data uploading to the hub) in Figure 11. As shown, Securitas results in a slight but acceptable increase in task latency, e.g., 0.56s, 0.83s, and 0.60s for the Camera cases of Direct VPN, Securitas (BMv2), and Securitas (eBPF). Note that Securitas (BMv2) always yields a higher latency, because BMv2 is designed for P4 testing rather than the production environment [20].



(a) Packet Throughput (b) Resource Utilization

Figure 12: The results of our FPGA testbed.

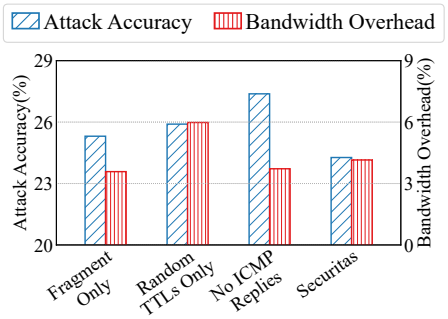


Figure 13: The performance of different obfuscation settings in Securitas (the original attack BILSTM accuracy 86.53%).

7.3 Microbenchmarks

Results on FPGA: To evaluate Securitas on FPGA devices, we also build an FPGA testbed (detailed in Appendix C.5). Similar to Figure 9a, Figure 12a shows the TRex's "real" traffic throughput in the web fingerprinting scenario. As shown, Securitas yields the highest throughput of 75.17Gbps, which is $\sim$ 20Gbps lower than the FPGA board can provide in the No Defense case. By analyzing with Vivado ILA [4], we find that the bottleneck is our egress pipeline, which has several complex tasks for the used Vitis P4 IP [5], such as maintaining the parsing process for different packet types and re-computing packets' IP checksum, and thus slows down the whole processing. Besides, Figure 12b is the FPGA resource utilization. As depicted, Securitas consumes a small fraction of resources. The max consumption is the Block RAM (69.46%), which is used for match-action tables and packet buffering.

Ablation results: Figure 13 is the ablation experiment of our obfuscation operations. As shown, "Securitas" (fragments, small-TTL packets, and ICMP replies) achieves a good balance between attack evasion and bandwidth overhead. We also show that the attack accuracy without ICMP obfuscation ("No ICMP Replies": fragments plus small-TTL packets only) is 27.38%. Compared to the "Securatus" case, the contribution of ICMP replies appears small. However, this is an expected result of our conservative setting (assuming a severe ICMP rate control in networks): with $Pr = 0.7$ and ICMP sending probability of 0.48, the ICMP proportion in the obfuscation is only approximately 14.4% (0.3×0.48). The defender can tune this setting according to the network states.

8 Discussion

Cases that Securitas may fail: Although the throughput of Securitas based on Tofino switches is 95.71Gps, the proxy client/server may not keep up with such throughput and cannot handle the "overflow" traffic. Also, if the user is located in a region where proxies are prohibited, Securitas may be compromised, as our obfuscation has obvious features (e.g., packet fragment flags and TTL values) without the proxy's encrypted tunnel. Actually, in such a situation, other obfuscators (e.g., [15, 27, 40, 44, 50, 69]) will be weakened too. Moreover, Securitas does not cover complex traffic features (e.g., the traffic bursts or the average packet size in a specific time window) and thus cannot handle the related attacks.

Obfuscation strategy updating: As introduced in §5, we train the strategy to optimize the attack evasion and the bandwidth overhead. So if the traffic characteristics are changed, retraining is required. For instance, the software/website used by the user is updated and has significantly different packet sequences. Fortunately, we do not need to recompile and reinstall the whole P4 program because the strategy is converted into P4 entries of match-action tables. Once the retraining is done, we only need to reinstall the table entries.

The overhead/impact on the network: One potential concern of Securitas is the receiver's fragment reassembly overhead (e.g., CPU and memory consumption). Fortunately, our real-world tests (§7.2) show that 26 Internet servers can respond normally (e.g., with a 0.15s increase in web load time). Besides, Securitas provides a tunable trade-off through Pr (see Algorithm 1) so that the defender can decrease Pr to reduce fragments and increase the ratio of small-TTL packets, shifting the overhead to other network nodes (which handle TTL expiration). Also, modern servers and NICs have accelerated reassembly (e.g., Generic Receive Offload [57]), which can mitigate the overhead too. Another concern may be whether our cloned small-TTL packets could affect the TCP behavior. E.g., these packets might accidentally arrive at the server side. If the cloned packet happens to be a duplicated ACK packet, TCP may trigger a decrease in the congestion window, slowing down the connection. Recall that Securitas injects cloned packets with small TTLs plus wrong checksums. So if the TTL is unexpectedly not decreased to zero and the packet goes to the server, the checksum validation can ensure that this packet will be discarded before TCP processing [53].

The impact on latency-sensitive applications: While Securitas enhances privacy, the additional processing overhead inherently impacts the user experience. To mitigate this, this paper focuses on the fast in-network mechanisms (Tofino, eBPF, etc.). Though the latency in our eBPF implementation (Figure 11) still looks high, this result was from simulations on a performance-limited laptop (detailed in Appendix C.4). By selecting a device of higher performance, Securitas is expected to get more latency gains as eBPF has demonstrated its capability in production deployments [64].

Robustness for an adaptive attacker: The attacker may retrain his DNN on obfuscated traffic. However, Securitas maintains several hyperparameters, e.g., fragment sizes and ratio Pr. As they are securely held by the defender, the attacker has to enumerate and train against a massive combination of them, which is computationally exhausting. Besides, our strategy optimization can be extended into a Generative Adversarial Network (GAN) style [32], incorporating an evolving proxy attacker for continuous obfuscation improvement.

9 Related Work

DNN-based Traffic analysis attack. For encrypted traffic, the side-channel information is still discriminative. Attackers thus feed such information to DNNs to identify user activities [55]. E.g., website fingerprinting attacks [9, 60, 65, 72] aim to infer the user-browsed websites by analyzing packets' directions and inter-arrival timings. The attacks of IoT fingerprinting [25, 39, 63] and application classification [46, 73, 78] are used to infer the IoT devices and the application types (e.g., email or chatting). **Obfuscation-based defense**. Several solutions are proposed to mitigate the mentioned attacks [15, 22, 27, 40, 44, 48, 50, 70, 71, 74, 79]. Their main idea is to obfuscate the traffic's side-channel information. For example, [27, 40, 44] use different schemes to insert fake packets adaptively. Ditto [48] pads each packet to have a predefined size and also adds fake packets. [22, 70, 71] use different algorithms to encrypt the addresses/QUIC connection IDs, hiding the attacker from observing the original traffic identifiers. These traffic obfuscators are either deployed as flexible plugins [15, 27, 40, 44, 50] on end-user browsers/onion routers, or as in-network services [22, 48, 70, 71, 74].

10 Conclusion

This paper proposes Securitas to evade traffic analysis attacks. Based on packet fragmentation, small-TTL packet insertion, and implicit ICMP replies, Securitas provides flexible traffic obfuscation. With the learning optimization, Securitas effectively balances attack evasion and bandwidth overhead. Also, our well P4 implementation makes Securitas applicable to multiple hardware/software targets. Thorough experiments show the superiority of Securitas, e.g., reducing attack accuracy by 95.89% and having a web load delay of 0.15s.

Acknowledgments. We thank Aurojit Panda, our reviewers, and shepherd Hyojoon Kim. This work was funded by the Basic and Frontier Research Project of PCL (2025QYB019, 2025QYA001), the Major Key Project of PCL (PCL2025A09), the Open Project of National Engineering Laboratory for Technology of Internet Domain Name (KF202512), the Program of China Scholarship Council (CSC202306210169), and the Shenzhen Key Laboratory of Software Defined Networking (ZDSYS20140509172959989).

References

[1] Using dns to estimate the worldwide state of ipv6 adoption. https://blog.cloudflare.com/ipv6-from-dns-pov. Accessed: 2024-06-10.

[2] Internet Protocol. RFC 791 https://datatracker.ietf.org/doc/html/rfc791#page-2, September 1981.

[3] Mahmoud Abbasi, Amin Shahraki, and Amir Taherkordi. Deep learning for network traffic monitoring and analysis (NTMA): A survey. *Computer Communications*, 170:19–41, 2021.

[4] Inc. Advanced Micro Devices. Integrated logic analyzer (ila). https://www.xilinx.com/products/intellectual-property/ila.html. Accessed: 2024-10-09.

[5] Inc. Advanced Micro Devices. Vitis networking p4. https://docs.amd.com/r/en-US/ug1308-vitis-p4-user-guide. Accessed: 2024-08-13.

[6] Microsoft Azure. Azure iot python sdk. https://github.com/Azure/azure-iot-sdk-python. Accessed: 2024-07-09.

[7] Barefoot Networks. Tofino native architecture. https://github.com/barefootnetworks/Open-Tofino/blob/master/PUBLIC_Tofino-Native-Arch.pdf. Accessed: 2025-01-13.

[8] Barefoot Networks. Tofino switch. https://www.barefootnetworks.com/products/brief-tofino/. Accessed: 2024-06-13.

[9] Sanjit Bhat, David Lu, Albert Kwon, and Srinivas Devadas. Var-cnn: A data-efficient website fingerprinting attack based on deep learning. *Proceedings on Privacy Enhancing Technologies*, 2019(4):292–310, 2019.

[10] 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. *Computer Communication Review*, 44(3):87–95, 2014.

[11] Aaron Boyd. Interior ig team used evil twins and $200 tech to hack department wi-fi networks. https://www.nextgov.com/cybersecurity/2020/09/interior-ig-team-used-evil-twins-and-200-tech-hack-department-wi-fi-networks/168521/. Accessed: 2024-06-09.

[12] Robert T. Braden. Requirements for Internet Hosts - Application and Support. RFC 1123 https://www.rfc-editor.org/info/rfc1123, October 1989.

[13] Robert T. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 https://datatracker.ietf.org/doc/html/rfc1122, October 1989.

[14] Tom B Brown, Dandelion Mané, Aurko Roy, Martín Abadi, and Justin Gilmer. Adversarial patch. *arXiv preprint arXiv:1712.09665*, 2017.

[15] 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*, pages 121–130. ACM, 2014.

[16] The NetFPGA Community. Netfpga. https://netfpga.org/. Accessed: 2024-06-09.

[17] The P4 Language Consortium. Behavioral model (bmv2). https://github.com/p4lang/behavioral-model. Accessed: 2024-08-13.

[18] The P4 Language Consortium. $P4_{16}$ language specification. https://p4.org/p4-spec/docs/P4-16-v1.0.0-spec.html. Accessed: 2024-06-10.

[19] The P4 Language Consortium. P416 reference compiler. https://github.com/p4lang/p4c. Accessed: 2024-10-09.

[20] The P4 Language Consortium. Performance of bmv2. https://github.com/p4lang/behavioral-model/blob/main/docs/performance.md. Accessed: 2024-10-13.

[21] Intel Corporation. Introduction to p416, intel® tofino™ family and intel p4 studio™ sde. https://www.intel.com/content/www/us/en/products/docs/network-io/ethernet-programmable-switch/ica-xfg-101-prospectus.html. Accessed: 2024-06-09.

[22] Trisha Datta, Nick Feamster, Jennifer Rexford, and Liang Wang. SPINE: surveillance protection in the network elements. In *Proceedings of the 9th USENIX Workshop on Free and Open Communications on the Internet*. USENIX Association, 2019.

[23] die.net. traceroute - print the route packets trace to network host. https://linux.die.net/man/8/traceroute6. Accessed: 2024-09-10.

[24] Roger Dingledine, Nick Mathewson, and Paul F. Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th USENIX Security Symposium*, pages 303–320. USENIX, 2004.

[25] Shuaike Dong, Zhou Li, Di Tang, Jiongyi Chen, Menghan Sun, and Kehuan Zhang. Your smart home can't keep a secret: Towards automated fingerprinting of iot traffic. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 47–59. ACM, 2020.

[26] Gerard Draper-Gil, Arash Habibi Lashkari, Mohammad Saiful Islam Mamun, and Ali A. Ghorbani. Characterization of encrypted and VPN traffic using time-related features. In *Proceedings of the International Conference on Information Systems Security and Privacy*, pages 407–414. SciTePress, 2016.

[27] Kevin P. Dyer, Scott E. Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 332–346. IEEE, 2012.

[28] Firewall.cx. Icmp protocol - part 7: Time exceeded message analysis. https://www.firewall.cx/networking/network-protocols/icmp-protocol/icmp-time-exceeded.html. Accessed: 2024-08-13.

[29] Linux Foundation. Data plane development kit (DPDK). http://www.dpdk.org/. Accessed: 2024-07-15.

[30] Linux Foundation. ebpf: Dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io/. Accessed: 2024-07-15.

[31] The PyTorch Foundation. Softmax. https://pytorch.org/docs/stable/generated/torch.nn.Softmax.html. Accessed: 2024-07-13.

[32] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. Generative adversarial nets. In *Proceedings of the Annual Conference on Neural Information Processing Systems*, pages 2672–2680, 2014.

[33] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *Proceedings of the 3rd International Conference on Learning Representations*, 2015.

[34] The P4.org Architecture Working Group. $P4_{16}$ portable switch architecture. https://p4.org/p4-spec/docs/PSA.html. Accessed: 2024-06-13.

[35] Hang Guo and John S. Heidemann. Detecting ICMP rate limiting in the internet. In *Proceedings of the Passive and Active Measurement - 19th International Conference*, volume 10771, pages 3–17. Springer, 2018.

[36] Jonatan Heyman, Carl Byström, Joakim Hamrén, and Hugo Heyman. Locust - an open source load testing tool. https://locust.io/. Accessed: 2024-10-09.

[37] HighSpeedInternet.com. How much internet speed do i need? https://www.highspeedinternet.com/how-much-internet-speed-do-i-need. Accessed: 2024-06-14.

[38] ikreymer. Web archiving tools for all. https://pypi.org/project/pywb/. Accessed: 2024-08-09.

[39] Houda Jmila, Gregory Blanc, Mustafizur R. Shahid, and Marwan Lazrag. A survey of smart home iot device classification using machine learning-based network traffic analysis. *IEEE Access*, 10:97117–97141, 2022.

[40] Marc Juarez, Mohsen Imani, Mike Perry, Claudia Díaz, and Matthew Wright. Toward an efficient website fingerprinting defense. In *Proceedings of the 21st European Symposium on Research in Computer Security*, volume 9878 of *Lecture Notes in Computer Science*, pages 27–46. Springer, 2016.

[41] Ahmed Najah Kadhim and Sattar B Sadkhan. Security threats in wireless network communication-status, challenges, and future trends. In *Proceeding of the International Conference on Advanced Computer Applications*, pages 176–181. IEEE, 2021.

[42] Lawrence Berkeley National Laboratory. iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr/contact.php. Accessed: 2024-07-09.

[43] Sándor Laki, Radostin Stoyanov, Dávid Kis, Robert Soulé, Péter Vörös, and Noa Zilberman. P4pi: P4 on raspberry pi for networking education. *Computer Communication Review*, 51(3):17–21, 2021.

[44] Ding Li, Yuefei Zhu, Minghao Chen, and Jue Wang. Minipatch: Undermining dnn-based website fingerprinting with adversarial patches. *IEEE Transactions on Information Forensics and Security*, 17:2437–2451, 2022.

[45] Mimecast Services Limited. Targeted phishing attacks are on the rise. https://www.mimecast.com/content/targeted-phishing/. Accessed: 2024-08-10.

[46] Chang Liu, Longtao He, Gang Xiong, Zigang Cao, and Zhen Li. Fs-net: A flow sequence network for encrypted traffic classification. In *Proceedings of the 38th IEEE Conference on Computer Communications*, pages 1171–1179. IEEE, 2019.

[47] Daniel Markuson. How to set up a vpn on your router. https://nordvpn.com/blog/setup-vpn-router/. Accessed: 2024-10-09.

[48] Roland Meier, Vincent Lenders, and Laurent Vanbever. ditto: WAN traffic obfuscation at line rate. In *Proceedings of the 29th Annual Network and Distributed System Security Symposium*. The Internet Society, 2022.

[49] Andrew W. Moore and Denis Zuev. Internet traffic classification using bayesian analysis techniques. In *Proceedings of the International Conference on Measurements and Modeling of Computer Systems*, pages 50–60. ACM, 2005.

[50] Milad Nasr, Alireza Bahramali, and Amir Houmansadr. Defeating dnn-based traffic analysis systems in real-time with blind adversarial perturbations. In *Proceedings of the 30th USENIX Security Symposium*, pages 2705–2722. USENIX Association, 2021.

[51] Thuy T. T. Nguyen and Grenville J. Armitage. A survey of techniques for internet traffic classification using machine learning. *IEEE Communications Surveys and Tutorials*, 10(1-4):56–76, 2008.

[52] Sanghak Oh, Minwook Lee, Hyunwoo Lee, Elisa Bertino, and Hyoungshick Kim. Appsniffer: Towards robust mobile app fingerprinting against VPN. In *Proceedings of the ACM Web Conference 2023*, pages 2318–2328. ACM, 2023.

[53] The Linux Kernel Organization. The linux kernel archives. https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/net/ipv4/tcp_ipv4.c?h=v6.8.8#n2197. Accessed: 2024-09-23.

[54] Tomasz Osinski, Jan Palimaka, Mateusz Kossakowski, Frédéric Dang Tran, El-Fadel Bonfoh, and Halina Tarasiuk. A novel programmable software datapath for software-defined networking. In *Proceedings of the 18th International Conference on emerging Networking EXperiments and Technologies*, pages 245–260. ACM, 2022.

[55] Eva Papadogiannaki and Sotiris Ioannidis. A survey on encrypted network traffic analysis applications, techniques, and countermeasures. *ACM Computing Surveys*, 54(6):123:1–123:35, 2022.

[56] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the Advances in Neural Information Processing Systems*, pages 8024–8035. Curran Associates, 2019.

[57] DPDK Project. Generic receive offload library. https://doc.dpdk.org/guides-25.11/prog_guide/generic_receive_offload_lib.html. Accessed: 2026-01-10.

[58] IO Visor Project. Bpf compiler collection. https://github.com/iovisor/bcc. Accessed: 2024-06-13.

[59] Eric Rescorla. HTTP Over TLS. RFC 2818 https://www.rfc-editor.org/info/rfc2818, May 2000.

[60] Vera Rimmer, Davy Preuveneers, Marc Juarez, Tom van Goethem, and Wouter Joosen. Automated website fingerprinting through deep learning. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*. The Internet Society, 2018.

[61] S Ruder. An overview of gradient descent optimization algorithms. arxiv 2016. *arXiv preprint arXiv:1609.04747*, 2020.

[62] Nord Security. Nordvpn. https://nordvpn.com/offer/. Accessed: 2024-08-09.

[63] Mustafizur R. Shahid, Gregory Blanc, Zonghua Zhang, and Hervé Debar. Iot devices recognition through network traffic analysis. In *Proceedings of the IEEE International Conference on Big Data*, pages 5187–5192. IEEE, 2018.

[64] Nikita Shirokov and Ranjeeth Dasineni. Open-sourcing katran, a scalable network load balancer. https://engineering.fb.com/2018/05/22/open-source/open-sourcing-katran-a-scalable-network-load-balancer/. Accessed: 2026-01-10.

[65] 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*, pages 1928–1943. ACM, 2018.

[66] Liran Tal. Every millisecond matters: Performance monitoring in the instant gratification era. https://www.glassbox.com/blog/every-millisecond-matters-performance-monitoring-in-the-instant-gratification-era/. Accessed: 2024-05-14.

[67] Cisco TRex Team. Trex: Realistic traffic generator. https://trex-tgn.cisco.com/. Accessed: 2024-07-09.

[68] Inc. TRENDnet. Letter from the president: Ip camera hack. https://www.trendnet.com/langen/press/view.asp?id=1960. Accessed: 2024-07-10.

[69] Gunjan Verma, Ertugrul N. Ciftcioglu, Ryan Sheatsley, Kevin S. Chan, and Lisa Scott. Network traffic obfuscation: An adversarial machine learning approach. In *Proceedings of the Military Communications Conference*, pages 1–6. IEEE, 2018.

[70] Liang Wang, Hyojoon Kim, Prateek Mittal, and Jennifer Rexford. Programmable in-network obfuscation of dns traffic. In *Proceedings of the 2021 DNS Privacy Workshop of Network and Distributed System Security Symposium*, pages 1–10. DNS Privacy Project, 2021.

[71] Liang Wang, Hyojoon Kim, Prateek Mittal, and Jennifer Rexford. RAVEN: stateless rapid IP address variation for enterprise networks. *Proceedings on Privacy Enhancing Technologies*, 2023(3):194–210, 2023.

[72] Tao Wang. High precision open-world website fingerprinting. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 152–167. IEEE, 2020.

[73] Xin Wang, Shuhui Chen, and Jinshu Su. App-net: A hybrid neural network for encrypted mobile traffic classification. In *Proceeding of the 39th IEEE Conference on Computer Communications*, pages 424–429. IEEE, 2020.

[74] Zihao Wang, Qing Li, Guorui Xie, Dan Zhao, Kejun Li, Zhuochen Fan, Lianbo Ma, and Yong Jiang. Minos : A lightweight and dynamic defense against traffic analysis in programmable data planes. In *Proceedings of the USENIX Annual Technical Conference*, pages 399–415. USENIX Association, 2025.

[75] Wikipedia. Chain rule. https://en.wikipedia.org/wiki/Chain_rule. Accessed: 2025-07-23.

[76] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8:229–256, 1992.

[77] Charles V. Wright, Fabian Monrose, and Gerald M. Masson. On inferring application protocol behaviors in encrypted network traffic. *Journal of Machine Learning Research*, 7:2745–2769, 2006.

[78] Renjie Xie, Yixiao Wang, Jiahao Cao, Enhuan Dong, Mingwei Xu, Kun Sun, Qi Li, Licheng Shen, and Menghao Zhang. Rosetta: Enabling robust tls encrypted traffic classification in diverse network environments with tcp-aware traffic augmentation. In *Proceedings of the ACM Turing Award Celebration Conference*, pages 131–132, 2023.

[79] Xiaokuan Zhang, Jihun Hamm, Michael K. Reiter, and Yinqian Zhang. Statistical privacy for streaming traffic. In *Proceedings of the 26th Annual Network and Distributed System Security Symposium*. The Internet Society, 2019.

A Routing Path Measurement

To decide the TTL values of the fake packets, we use traceroute [23] to measure the representative servers in the CW100 dataset [60] for a week. Similar to settings in §7.2, the traceroute traffic path in our measurement is: "traceroute → proxy client → proxy server (NordVPN) → Internet nodes and the final destination". The measured probability that a hop sends back an ICMP reply, and the accumulated probability that a hop is the destination, are in Figure 14.

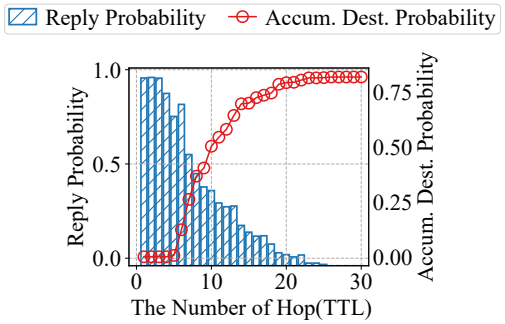


Figure 14: Measurement results on the CW100 dataset [60].

When we design Securitas, we hope that most of the fake packets can be discarded before reaching the destination, thereby avoiding extra processing overhead on the server side. Additionally, we aim to receive ICMP replies to obfuscate the user's inbound traffic. After analyzing Figure 14, we set the TTL in a fake packet as a random integer $\in [1,8]$. As shown, when TTL≤ 8, we can reach 36.70% destinations (because the accumulative probability is 0.367), and we can receive an ICMP reply with a probability ≥ 0.48.

Notably, one may repeat such measurements to precisely configure TTLs when deploying Securitas in his network. Nevertheless, leaving the default TTL setting unchanged is still practical due to the following reasons: **1)** When Securitas generates fake packets, it ensures that these packets are invalid in the transport layer (i.e., with wrong TCP/UDP checksums). As such, if a fake packet arrives at the destination (the TTL is unexpectedly not reduced to 0), it can still be discarded and has little interference with the real traffic. **2)** §7.3 and Figure 13 show that without ICMP replies, Securitas can also defend against the attacker (with an acceptable performance decrement).

B Data Plane Implementation Details

Our detailed P4 implementation is shown in Figure 15. This P4 program is one-directional, i.e., packets enter the ingress pipeline in order, then are processed by the packet replication & buffer engine (PRE), finally, the egress pipeline processes the packets and forwards them out.

Ingress Pipeline: The ingress parser is programmed to parse predefined header fields (e.g., the five-tuple) of packets by the standard P4 function `extract(.)`. Then, the parsed headers are sent to the subsequent customized *three* match-action tables:

- Table #1 has no entries, and its default action is to count the number of packets per user-server session. For each packet, its five-tuple (in the rearranged form of {user IP, user port, server IP, server port, protocol}) is hashed to obtain

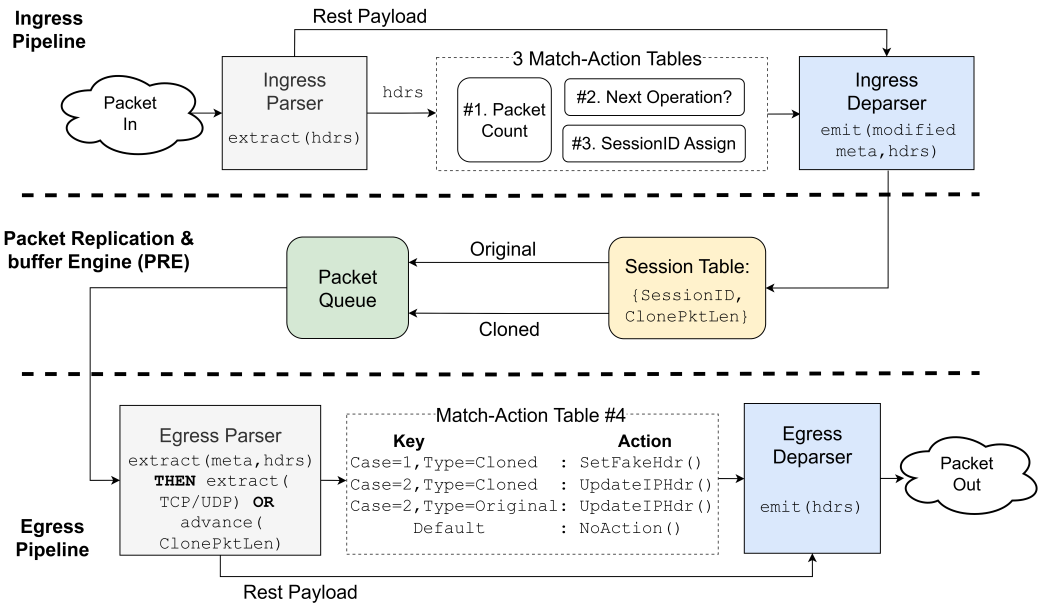


Figure 15: Securitas on the standard Portable Switch Architecture (PSA) of P4.

a register index v. Then, the value i of the v-th register is increased as $i = i+1$. Here, we allocate a large number of registers (i.e., 2^{18}) to mitigate the possible hash collisions.

- Table #2 uses two metavariables (i.e., register value i, and packet *direction*) as keys to match table entries, deciding the next operation. If $direction = server \rightarrow user$, the packet is a user's inbound one, so we do nothing and jump to the ingress deparser by the `exit` command. Otherwise, i will index an a_i, making the packet insertion/fragmentation associated with a_i the next operation. If a_i is related to the fake packet insertion, Table #2 sets the packet's metavariable $Case = 1$. If a_i has a related fragmentation operation, Table #2 sets the packet's $Case = 2$. Notably, if $a_i \geq packet.size$, we cannot fragment the packet and roll back to set $Case = 1$ instead. Moreover, a_i can be zero, which also leads to jumping to the ingress deparser directly.

- Table #3 checks the packet of $Case = 1$ or 2, using a_i as the key to assign the packet a *SessionID*. *SessionID* is used later in the PRE for packet cloning.

Finally, the ingress deparser utilizes the standard P4 function `emit(.)` to reorganize the headers and the modified metavariables back to the packet, allowing us to pass the processing context to the PRE and the egress pipeline.

PRE: A useful technique for network devices is packet mirroring, i.e., sending the packet to its normal destination and a copy of the packet as received to another output port (e.g., to a monitoring device). PSA has defined such a packet mirroring/cloning technique in its PRE module. Hence, we suggest using PRE to generate extra packets of desired lengths for our obfuscation. Specifically, the PRE maintains a special session table consisting of $(SessionID, ClonePktLen)$ pairs, which can be configured in advance through the control plane API. After our configuration, when a packet has a specific *SessionID*, the PRE will clone an additional instance of this packet. The PRE will also automatically truncate the cloned packet if its size exceeds the *SessionID*'s correlated *ClonePktLen*. In Securitas, *ClonePktLen* is configured to be equal to a_i[4] to obtain cloned packets of required sizes. Because packets are handled one by one, the original and cloned packets (with metavariable $Type = Cloned$) should then be temporarily buffered in the queue in the PRE before the following egress pipeline processing.

Egress Pipeline: When a packet arrives at the egress pipeline, the egress parser is also programmed to use `extract(.)` for packet header and metavariable parsing. Then, the parser triggers further parsing based on two parsed metavariables (*Case* and *Type*):

- $Case = 1, Type = Cloned$: If this is the fake packet case and the packet is cloned, we further parse the transport layer (TCP/UDP) checksum of this packet.

- $Case = 2, Type = Origin$: If this is the fragment case and the packet is the original one, we parse and discard its first *ClonePktLen* (equal to a_i[5]) payload bytes by the standard P4 function `advance(.)`.

- If not the two cases above, no more parsing is required, and we exit the parser.

After the egress parser, we allocate one match-action table

[4] This is because Lines 8 and 10 of Algorithm 1 need new a_i-byte packets.

[5] We want to obtain a fragment of $p_i.size - a_i$ bytes for Line 8, Algorithm 1.

(i.e., Table #4) to update the parsed data. The entries (key-action pairs) in Table #4 are introduced as follows:

- For the cloned packet of $Case = 1, Type = Cloned$, we should process it to be fake with a random TTL and a wrong TCP/UDP checksum, which is done by hashing in our customized P4 action `SetFakeHdr(.)`.
- For the fragmented packet ($Case = 2, Type = Cloned/Origin$), we update its related IP headers (i.e., $totalLen$, MF, IP checksum, and $offset$, see Figure 5) via our P4 action `UpdateIPHdr(.)`. Action `UpdateIPHdr(.)` processes cloned and original packets differently: the tail of the cloned packet ($Type = Cloned$) is truncated by the PRE, so we process it to be the first fragment in this action, and the corresponding original packet ($Type = Origin$) is the second fragment.
- For packets that do not meet the entries above, we do nothing (i.e., `Default: NoAction`).

Finally, in the egress deparser, we reorganize the processed headers back to the packet through `emit(.)` again and forward it out.

By primarily utilizing the standard P4 modules (pipelines, PRE), functions (`extract(.)`, `advance(.)`, etc.), and customized match-action tables, this implementation is expected to run on various hardware and software data plane targets. Although some targets may have their exclusive P4 characteristics, e.g., PRE is instantiated as TM (Traffic Manager) in Tofino [8], and eBPF [30] cannot use range match in match-action tables, these are minor issues; please see our discussion in the next Appendix B.1.

B.1 Data Plane Examples

As aforementioned, $P4_{16}$ and PSA are becoming the new programming standard, supported by multiple data plane targets [8, 16, 17, 29, 30, 43]. Several prevalent data plane examples are discussed below. Although some of them may not fully support PSA due to their own characteristics, they can adopt our above implementation with minor modifications.

BMv2 and eBPF [17, 30] are two software targets. Notably, BMv2 is the popular reference P4 software switch, which perfectly supports $P4_{16}$, PSA, and thus our implementation. eBPF has a better processing performance than BMv2. With P4C [54], we can install PSA-based P4 programs on eBPF. However, eBPF currently does not support range match in match-action tables. Hence, we should use other match types (i.e., exact or ternary match) when writing match-action tables for eBPF.

Tofino switch [8] executes $P4_{16}$ programs using its own Tofino Native Architecture (TNA [7]), a PSA variant with slight differences. TNA consists of {ingress pipeline, traffic manager (TM), egress pipeline}, where TM acts similarly as the aforementioned PRE in PSA, i.e., packet cloning and buffering. The main difference is that, except for assigning *SessionID* to packets, we should explicitly call an extern function `mirror(.)` in the ingress deparser to trigger TM's packet cloning.

FPGA devices now support $P4_{16}$ and partial of the PSA thanks to the Vitis P4 IP core [5]. AMD/Xilinx maintains the Vitis P4 IP core in its Vivado suite. Given P4 codes maintaining pipelines (i.e., ingress and egress pipelines in Figure 15), Vivado will compile them as Vitis P4 IP cores to run on the FPGA device. However, Vivado does not have the PRE module, so we should write a module for packet cloning and buffering. Fortunately, there are several off-the-shelf FPGA components (e.g., AXI4-Stream Data FIFO IP core, and registers) that can easily finish the cloning and buffering functionalities.

C Detailed Experimental Settings

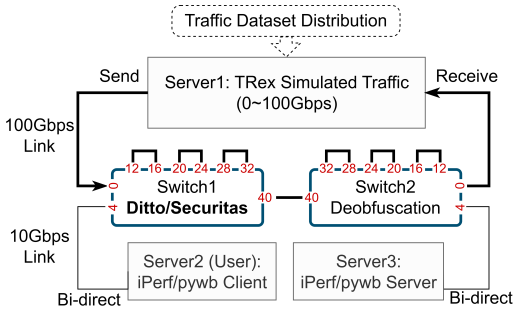
C.1 Models

Securitas' obfuscation strategy $\boldsymbol{a}$ is parameterized by a single-layer unidirectional LSTM with 32 hidden units, which autoregressively processes the embedding of the previous a_i. The LSTM's hidden state is projected to a set of logits corresponding to the length of $\boldsymbol{a}$, and then normalized via a softmax function. We train Securitas, Minipatch [44], BLANKET [50], and the attack models [9, 25, 46, 60, 63, 65, 73, 78] on the SYS-4029GP-TRT server with NVIDIA GeForce RTX 2080 Ti GPU (11GB). All codes are implemented through Python 3.8 and the machine learning library PyTorch 1.9.1 [56]. Each of the used datasets (CW100 [60], HomeMole [25], ISCX [26]) is randomly split into three parts: 70% for training, 10% for validation, and 20% for testing.

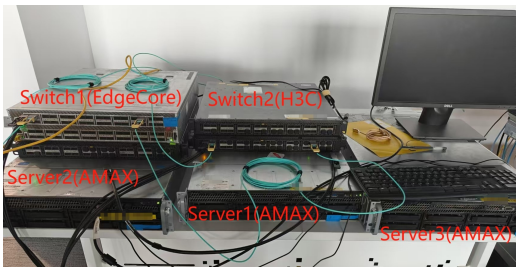
C.2 Tofino Switch Testbed

In our Tofino testbed (Figure 16), the two Tofino switches are EdgeCore Wedge 100BF-65X (Switch1) and H3C S9850-32H (Switch2). Switch1 is deployed with Ditto or Securitas, and Switch2 performs the corresponding de-obfuscation. For Ditto's de-obfuscation, we simply deploy another Ditto instance on Switch2. However, P4 does not support reassembling fragments, so Switch2 makes de-obfuscation for Securituas as follows: For the fake small-TTL packets, they are discarded after TTL checking; For two fragments of a packet, the first fragment is discarded, and the second one is padded with fake bytes to match the original packet size. We use the Tofino compiler SDE 9.10 [21] to compile our P4 codes.

The three servers (Server1, Server2, Server3) are all from AMAX with the OS of Ubuntu 22.04. Server1 runs TRex 3.04 [67], an open-source traffic generator that helps us generate traffic of up to 100Gbps. Server2 simulates the user, running clients of iPerf 2.1.5 [42] and pywb 2.8.3 [38]. iPerf is a network tool that helps us test the network states (e.g.,



(a) Logic



(b) Reality

Figure 16: Tofino switch testbed (the complicated paired ports 12∼32 are only used by Ditto).

packet loss), and pywb can capture website content and replay it. Server3 installs the server modules of iPerf and pywb.

C.3 Modified Tofino Switch Testbed for Real-World Tests

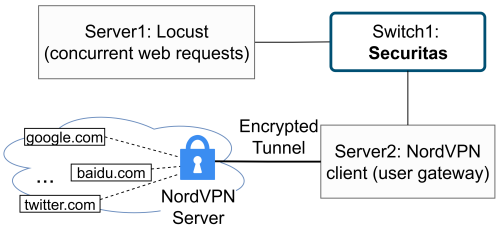


Figure 17: Modified Tofino switch testbed for real-world tests.

To make real-world Internet tests, we modify the Tofino switch testbed in Figure 16. According to the new testbed in Figure 17, Server1 runs Locust 2.25.0 [36] and simulates 30 users for web browsing in parallel. In Server2, we install the proxy client (strongSwan 5.6.2) of NordVPN according to [47]. We also configure Server2 to forward traffic from Server1 using iptables commands[6]. Then, we subscribe to a proxy server (NordVPN [62]) in LA, US, using it to visit web servers. We finally test 26 public web servers from [60], which are hosted in five countries (Canada, France, Singapore, Britain, and the USA). Our total tests go through 112 and 15 distinct hops and ASes, respectively.

[6] https://man7.org/linux/man-pages/man8/iptables.8.html

C.4 Software Testbed for Real-World Tests

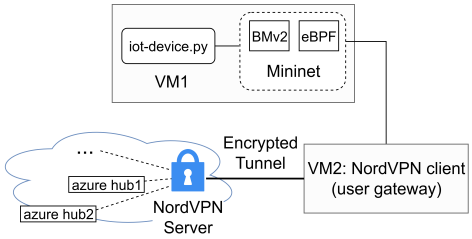
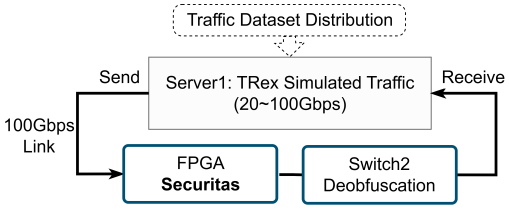


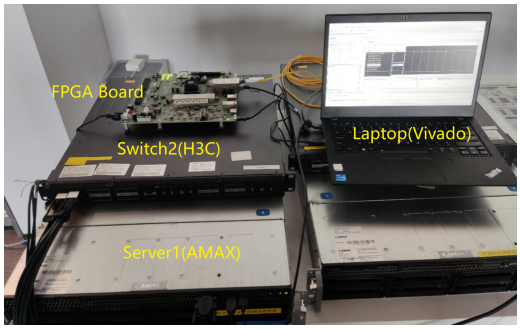
Figure 18: Software testbed for real-world tests (VM1 and VM2 are on the same laptop).

To build the software testbed (Figure 18), we instantiate virtual machines (VM1 and VM2) on a Lenovo laptop by VMware Workstation 17.5.1[7]. Data planes (BMv2/eBPF) are based on the Mininet 2.3.1b4[8], and P4 codes are compiled by P4C variants of p4c-bv2-psa 0.0.1 (for BMv2) or p4c-ebpf 1.2.4.8 (for eBPF) [19]. In VM1, we use the Python library azure-iot-device 2.13.0 [6] to test our connections to Azure IoT hub on the Internet. The settings of the proxy client and iptables on VM2 are the same as Appendix C.3.

C.5 FPGA Testbed



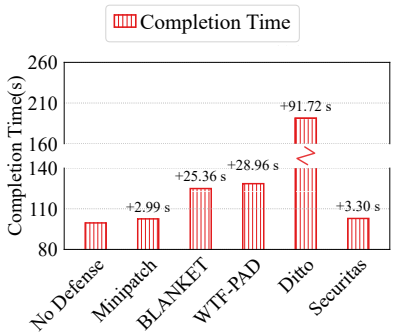
(a) Logic



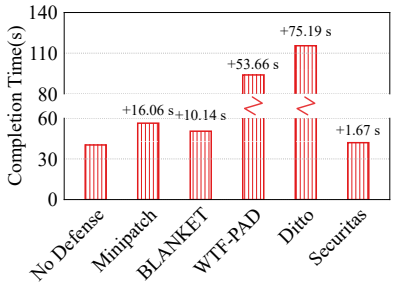
(b) Reality

Figure 19: FPGA testbed (according to the user guide, the max throughput of this FPGA board is 100Gbps).

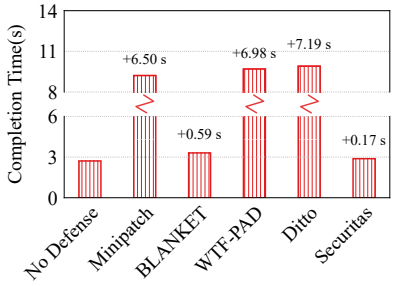
[7] https://www.vmware.com/products/desktop-hypervisor
[8] https://mininet.org/



(a) Web fingerprinting scenario



(b) IoT fingerprinting scenario



(c) Application classification scenario

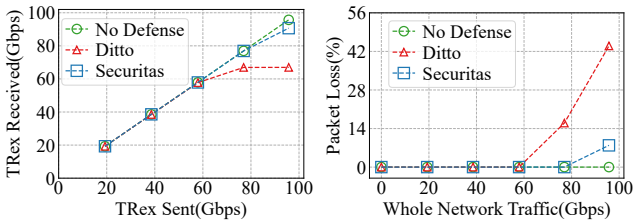
Figure 20: Traffic completion time before (i.e., "No Defense") and after obfuscation (better obfuscators have lower values).

The FPGA testbed used in §7.3 is illustrated in Figure 19. We use an FPAG board of Zynq UltraScale+ MPSoC ZU19EG. Server1 and Switch2 are from the Tofino switch testbed (see Figure 16 in Appendix C.2). As this FPGA board supports a throughput of up to 100Gbps, we also connect these devices via 100Gbps links for tests. To install Securitas on the FPGA, we use the Vitis P4 IP and Vivado 2024.1 [5].

D More Experimental Results

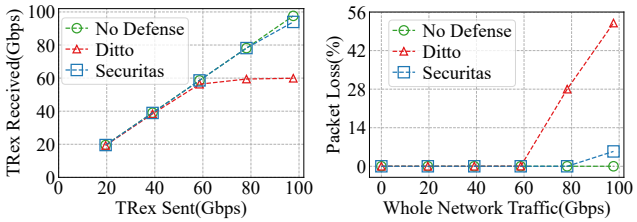
D.1 More Comparisons with Baselines

Figure 20 is the traffic completion time of different obfuscators. This experiment considers the same traffic datasets used in the three attack scenarios in §7.1. They are: the CW100 dataset [60] for the web fingerprinting scenario, the HomeMole dataset [25] for the IoT fingerprinting scenario, and the ISCX dataset [26] for the application classification scenario. Here, the traffic is transmitted through a simulated 100Mbps link [37]. According to the previous Table 3, we know that Securitas has the smallest bandwidth overhead. As such, Securitas also has the lowest completion time among the obfuscators, e.g., only adds an extra 1.67s when transmitting the obfuscated traffic in the HomeMole dataset.



(a) TRex sent vs. received (b) Single user's packet loss

Figure 21: Tofino switch testbed results with synthetic traffic from the HomeMole dataset (a better obfuscator should have performance closer to "No Defense" so that it can have little interruption to the network).



(a) TRex sent vs. received (b) Single user's packet loss

Figure 22: Tofino switch testbed results with synthetic traffic from the ISCX dataset (a better obfuscator should have performance closer to "No Defense" so that it can have little interruption to the network.

Except for Figure 9 in §7.1, we also make extended comparisons with the advanced in-network obfuscator Ditto [48] on throughput and packet loss in Figures 21 and 22. These experiments are conducted by our Tofino switch testbed in Figure 16. These results confirm that Securitas has a better performance than Ditto. For example, in Figure 21a, the maximum real traffic Securitas can handle (obfuscate) per second is 90.69Gb, while this number in Ditto is 66.98Gb. Besides, Figure 23 is Switch1's resource utilization of Ditto [48] and Securitas on the Tofino switch testbed. Thanks to our standardized and simplified implementation, Securitas is more lightweight than Ditto. For example, Ditto consumes more Tofino stages, i.e., 83.33% vs. 66.67% in Securitas.

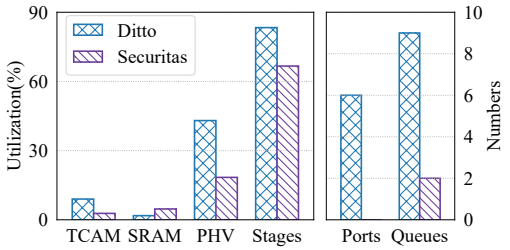


Figure 23: Tofino switch resource utilization (TCAM and SRAM: memory for table entries/registers; PHV: containers for parsed packet headers; Stages: pipeline stages; Ports and Queues are for packet forwarding).

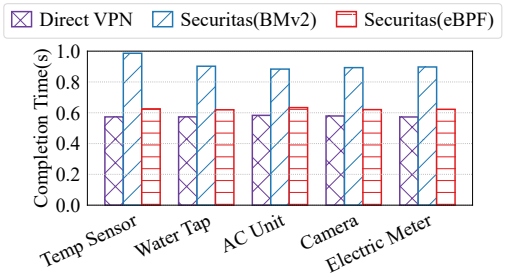


Figure 24: Traffic completion time of IoT tasks between the IoT devices and the Asian Azure IoT hub ("Direct VPN" indicates that IoT devices connect to the hub without Securitas).

D.2 More Results of Software Testbed and Microbenchmarks

As Azure has networks distributed worldwide, in addition to its European Azure IoT hub discussed in §7.2, we also test Securitas' Internet performance by connecting the five simulated IoT devices to the Asian Azure IoT hub through our software testbed. Figure 24 illustrates the results. Similar to §7.2, using Securitas will increase the traffic completion time slightly, e.g., 0.60s (Securitas (eBPF)) vs. 0.56s ("Direct VPN") for the Camera. However, this cost is worthwhile: Securitas adds small overhead (0.04s↑), but makes the user safer in the meantime. Again, Securitas (BMv2) has a worse performance because BMv2 is a P4 debug tool and is not aimed to provide competitive performance [20].

In Securitas, our obfuscation strategy is obtained by training a neural agent on the dataset (see §5.2). According to Eq. 6, the batch size B, i.e., the number of packet sequences in a training batch, plays an important role when we train the agent. Figure 25 depicts the training states when B has different values. As shown, using $B = 256$ can have a lower loss (i.e., a better performance), and the training is converged after 400 epochs. Also, according to our records, training the agent for 500 epochs costs us $\sim$ 3.5 hours.

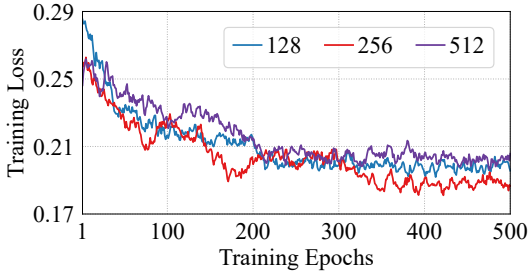


Figure 25: The number of packet sequences per batch (i.e., batch size) and its effect on the training loss of the ISCX dataset (the lower, the better).