NetWarden: Mitigating Network Covert Channels while Preserving Performance

Jiarong Xing Qiao Kang Ang Chen
Rice University

Abstract

Network covert channels are an advanced threat to the security of distributed systems. Existing defenses all come at the cost of performance, so they present significant barriers to a practical deployment in high-speed networks. We propose NetWarden, a novel defense whose key design goal is to *preserve TCP performance* while mitigating covert channels. The use of programmable data planes makes it possible for NetWarden to adapt defenses that were only demonstrated before as proof of concept, and apply them at linespeed. Moreover, NetWarden uses a set of performance boosting techniques to temporarily increase the performance of connections that have been affected by covert channel mitigation, with the ultimate goal of neutralizing the overall performance impact. NetWarden also uses a fastpath/slowpath architecture to combine the generality of software and the efficiency of hardware for effective defense. Our evaluation shows that NetWarden works smoothly with complex applications and workloads, and that it can mitigate covert timing and storage channels with little performance disturbance.

1 Introduction

Network covert channels are an advanced class of security threats to distributed systems. Using covert channels, an attacker can exfiltrate secret information from compromised machines without raising suspicion from firewalls, which typically only inspect packet payload. Covert *timing* channels [20, 21, 32, 46, 49, 61, 67] modulate packet timing to leak data, e.g., by using large and small inter-packet delays (IPDs) to encode ones or zeros in a secret message [21]. Covert *storage* channels [11, 24, 33, 37, 41, 51, 59], on the other hand, embed data inside packet headers, e.g., in the TCP sequence number [24] or ACK [50,51] fields. Covert channels have been demonstrated to be viable "in the wild" over long distances [21, 50], and major computer security standards—including the U.S. TCSEC [26], the European ITSEC [4], and the International standard Common Criteria [3]—explicitly require protection against covert channels.

Over the years, researchers have developed a variety of solutions to detect and mitigate network covert channels [17, 21, 24, 31, 52, 57]. For instance, in order to detect timing channels, existing detectors rely on statistical properties of known-good traffic IPDs to detect anomalous IPD modulation in a given traffic trace [21, 31]. In order to detect storage channels, existing detectors analyze packet header fields that could be used to encode data (e.g., TCP sequence number [24]) and look for anomalies. Upon detection, a range of mitigation techniques can then be applied, including buffering or delaying packets to disrupt the IPD patterns (for timing channels) [17,31], or setting certain header fields to controlled values (for storage channels) [24, 52].

It is perhaps unsurprising that no detector—whether for timing or storage channels—can achieve 100% accuracy. This is because the timing and header values of network traffic can be highly non-deterministic, as they depend on subtle interactions between the hosts and the network. For instance, a timing channel detector may raise a false alarm if IPDs suddenly increase, but this may have been caused merely by congestion. As further examples, the TCP protocol, which carries 99%+ traffic in modern datacenters [13], leaves many header values underspecified—e.g., the advertised receive window size may change dynamically based on the receiver's available buffer size, and the ACK number would reflect the amount of bytes that have been successfully received. A covert channel could easily hide itself in the permitted behaviors of TCP by "repurposing" these headers [50].

To compensate for detection inaccuracy, we could be more aggressive in mitigation—e.g., applying a blanket defense to all connections that *might* contain a channel. The obvious consequence here is performance degradation. Since most connections may be benign, an aggressive defense may unduly penalize legitimate flows. For instance, in order to mitigate covert timing channels, we could buffer or delay packets in a flow to disrupt their IPD patterns. However, this would increase latency and degrade TCP throughput. In order to mitigate covert storage channels, we could reset suspicious header fields to conservative values (e.g., reducing the receive window size), but this again would adversely affect the network transfer performance. Overall, we are faced with a concrete instance of the more general phenomenon that security comes at the cost of performance. Unfortunately, performance is a non-negotiable requirement in modern networks.

Our contribution. The key contribution of this paper is the design of a novel defense called NetWarden. It is a system that can support a range of covert channel defenses *in a performance-preserving manner* using a combination of three key techniques. First, NetWarden leverages *programmable data planes* in emerging switch hardware as a practical basis for covert channel defense. Programmable data planes can perform per-packet operations over header fields, which enables NetWarden to inspect and modify headers for storage channel mitigation without stalling the traffic. They can also support sophisticated data structures directly in switch hardware, which provides a building block for NetWarden to precisely monitor each connection and discover problematic protocol behaviors (e.g., abnormal IPDs, incorrect ACKs). Leveraging these features, NetWarden adapts a range of defenses that only exist as "proof of concept" today, and applies them to linespeed traffic with nanoseconds of extra delay.

Second, NetWarden also uses a set of *performance boosting* techniques to counteract the performance penalty due to covert channel defense. These techniques are inspired by results showing that the TCP congestion control mechanism can be manipulated to artificially inflate the sending rate [39]; NetWarden uses similar techniques for a very different goal. Concretely, NetWarden uses *ACK boosting* and *receive window boosting* to increase the sending rate of a connection. ACK boosting creates the illusion of a fast network, and receive window boosting creates the illusion of a high-performance receiver, ramping up the sending rate of the data source. NetWarden also temporarily caches excess packets locally; should any packets be dropped on their way to the receiver, NetWarden can still serve the data to the receiver as a proxy. NetWarden then uses these techniques in combination with defenses that usually lead to performance degradation, so that they neutralize each other's effects.

The third novelty in NetWarden comes from its *fastpath/slowpath architecture*. Programmable data planes have restricted programming models, so they cannot easily support all operations needed for covert channel defense. In the NetWarden architecture, the hardware fastpath supports a few key operations that need to run at linespeed, and the software slowpath supports more expressive, general-purpose operations that can only be invoked sparingly. Generally speaking, per-packet operations over constant-size states are pushed down to the fastpath for efficiency, and batch operations over growing states are hoisted up to the slowpath for generality. Having opposite tradeoffs, these two components complement each other in NetWarden to achieve an effective defense.

We have implemented a hardware prototype of NetWarden in P4 [9], performed a comprehensive set of evaluation using realistic traffic traces and applications, and released the source code in an online repository [7]. We have found that NetWarden can detect a range of network covert channels at full linespeed, mitigate them with negligible performance disturbance, and work smoothly with complex applications.

2 Overview

In this section, we introduce more background on network covert channels, discuss existing defenses and their limitations, and describe the key design techniques in NetWarden.

2.1 Network covert channels

Covert timing channels. Covert timing channels [21, 31, 45, 49] can exfiltrate secret data by modulating the IPDs of network traffic, e.g., by using large (small) IPDs to encode ones (zeros). Existing work has shown that these channels are practical even over a long distance.

Detection. Since the modulated traffic trace would have different IPD distributions from these of normal traffic, timing channel detectors look for statistical deviations between a given IPD distribution and a known-good distribution as obtained from training data [21, 21, 31, 57]. For instance, supposing that the known-good IPD data exhibits a normal distribution, a covert channel that uses small and large IPDs would distort that into a bimodal distribution. A detector can therefore detect signs of covert timing channels by looking for anomalous IPD distributions, e.g., by performing a Kolmogorov-Smirnov test [57] over IPD data. In practice, however, this is only viable in an offline manner—streaming high-speed traffic through these statistical detectors in real time would cause enormous overhead.

Mitigation. In principle, mitigating timing channels is easy. As discussed, we could buffer or inject random delays to network traffic to disrupt the IPD modulation [17]. However, this is only practical if detectors can precisely pinpoint flows for delay randomization. Otherwise, false positives in statistical detectors would cause normal flows to be penalized.

Covert storage channels. The simplest storage channels (Type-I) can encode data in optional or unused TCP/IP header fields, such as ToS, Urgent Pointer, and IPID fields [27]. More advanced channels (Type-II) encode data in header fields that are essential for protocol correctness, such as the TCP initial sequence number [24]. A particularly tricky class of channels (Type-III) can hide themselves in the inherent non-determinism of network traffic, e.g., embedding data into the receive window size or ACK fields [50].

Detection. A common strategy for detection is to inspect all header fields, and look for the existence of header fields that are rarely used or contain suspicious values. However, the need to inspect (and potentially modify) all packet headers already makes most software-based detectors impractical.

Mitigation. Temporarily shelving performance concerns, Type-I channels can be mitigated by setting optional header fields to controlled values. Type-II channels can also be mitigated using a similar strategy, but the defense needs to be stateful and apply the same actions to all packets in the flow to maintain correctness (e.g., adding a fixed offset to all TCP sequence numbers [24]). Type-III channels are the hardest, as they exploit the non-determinism in network traffic. To the best of our knowledge, no effective defenses exist today. NetWarden is the first defense against these channels, and it relies on visibility into the network traffic to resolve the non-determinism as much as possible.

2.2 Requirements for a practical defense

To summarize the above, existing defenses suffer from several limitations: the overhead that comes with inspecting all

Challenge	Technique(s)	Section(s)
Real-time header inspection/modification	Linespeed per-packet operations on programmable data planes	3.1
Resolving ambiguity when detecting advanced storage channels	Per-connection TCP state tracking	3.1
Boosting connection performance	ACK boosting + receive window boosting	3.2
Preserving performance despite mitigation	The principle of maximized transparency	3.3+3.4
Addressing the restrictions of the hardware programming model	Fastpath/slowpath defense architecture	4.1
Computing IPD in real time	Leveraging hardware timestamps + linespeed per-packet operations	4.2
Handling growing IPD state	IPD intervalization + sketching + software backstore	4.2+4.3
Minimizing fastpath/slowpath interaction	Fastpath IPD pre-checks + exact IPD monitoring for selected flows	4.2
Supporting sophisticated statistical tests	Fastpath characterizes IPDs, slowpath performs tests	4.2+4.3

Table 1: Key challenges and techniques in the design of NetWarden.

packet headers and/or timestamps in software, the inability to develop perfect detectors, and the performance penalty due to mitigation. Below, we dive deeper on these limitations to distill two key requirements for a practical defense.

Detection: inefficiency. Detecting covert channels requires per-packet operations, such as examining packet header fields and computing packet IPDs. At first glance, these operations do not seem very complicated to perform. However, the sheer volume and velocity of traffic in modern networks (e.g., 100Gbps per port, Tbps in aggregate) make even such operations infeasible unless we have specialized hardware support. Existing detectors built in general-purpose software are only demonstrated as proof of concept, working mostly in offline mode over low-speed or small samples of network traffic [20, 21, 32, 49].

Platforms that can handle high-speed traffic do exist—the switch hardware is customized to process traffic at linespeed with minimal overheads. However, traditional switches can only perform simple operations such as IP-based packet forwarding. Covert channel defense requires more sophisticated operations, such as inspecting/modifying headers and computing/testing IPDs, which go much beyond the capability of traditional switch hardware.

As a very basic requirement, we need *an efficient detector that can operate over linespeed traffic without stalling it.*

Detection: inaccuracy. A detector's accuracy in terms of false positive and negative rates is equally important. As we discussed, statistical detectors inevitably have some level of inaccuracy due to the inherent ambiguity and non-determinism of network traffic. Training data might be too small or too specific, or network conditions may have changed over time. One could always re-train the detectors with higher-quality or more data to improve the accuracy, but developing a perfect statistical detector is always difficult.

Alternatively, we could avoid the need for statistical tests by eliminating non-determinism. Suppose we could tightly control a system's expected behaviors (e.g., buffer size, kernel state, execution timing), then we can precisely detect with high (or perhaps even perfect!) accuracy when something goes wrong. Indeed, such software can be built using system-enforced determinism [16,22,68], which in turn yields very high accuracy in covert channel detection [22] and mitigation [68]. However, systems like these require intrusive changes to, or complete rewrites of, the OS kernel or the VMM, rendering a practical deployment quite challenging.

Mitigation: performance penalty. The inaccuracy of detectors does not interact well with the fact that mitigation techniques tend to cause performance penalty, e.g., injecting extra packet delays. If we could detect with perfect accuracy that a connection contains a covert channel, then we can aggressively mitigate the channel despite performance penalty, or perhaps even shut down the connection altogether. However, with unreliable detectors, this runs into the risk of causing performance drops of legitimate flows. Unfortunately, when faced with making tradeoffs between security and performance, the balance tends to tip towards the latter. While this practice could (and should) change over time, having to choose between security and performance certainly hinders practical defenses even further.

We thus arrive at our second requirement: to achieve a practical defense, *we either need a perfect detector, or we tolerate detection inaccuracy by designing mitigation techniques that preserve performance.*

2.3 Key techniques of NetWarden

NetWarden satisfies the above requirements by designing *linespeed covert channel detectors* and *performance-preserving mitigation techniques*. Table 1 summarizes the new techniques in our design; we elaborate more below.

Technique #1: Use programmable data planes. NetWarden achieves linespeed detection by leveraging programmable data planes, which are available in recent switch and NIC models (e.g., Intel FlexPipe [5], Broadcom Trident 4 [2], Netronome Agilio [6], and Barefoot Tofino [1]). These hardware provide new features that were originally designed for better *networking*, but interestingly, we observe that the same features match the requirements of covert channel defense surprisingly well.

Programmable data planes can perform per-packet header operations at linespeed. The packet processing pipeline in recent switches can be programmed using high-level languages (e.g., P4 [9]) to specify custom match/action behaviors and perform header inspections/modifications. This can be used as a building block for defending against covert storage channels. Moreover, they have a fine-grained timestamping facility. This was originally designed for achieving higher network visibility for diagnosis, but it also provides useful support for timing channel detection. Finally, they support sophisticated

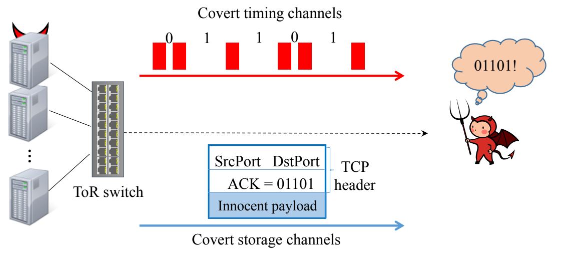


Figure 1: NetWarden can be deployed in a ToR switch to protect a rack of servers hosting sensitive data.

data structures that can sustain linespeed reads and writes using stateful registers. NetWarden can use this feature for precise connection monitoring, which further enables targeted covert channel mitigation.

Technique #2: Performance boosting. Moreover, NetWarden specifically designs for a key goal: *preserving performance*. In addition to customizing existing (and performance-degrading) defenses for programmable data planes, we also design a set of performance-boosting defenses. Using them in combination, NetWarden can neutralize the overall performance impact of covert channel mitigation. Some of these defenses, however, go beyond the capability of the switch hardware, and require a certain level of general-purpose software support, leading to our third technique.

Technique #3: Fastpath/slowpath defense. Programmable data planes have a rather restricted programming model, so they cannot support all operations that we need for covert channel defense. For instance, they can provide packet timestamps and perform simple IPD range checks, but statistical tests over IPD distribution are not implementable in hardware. Therefore, another design principle of NetWarden is to offload key primitives to the data plane as a fastpath defense, and then perform the rest of the processing in software slowpath. The slowpath could either reside in the local switch control plane, which has general-purpose CPUs and abundant RAM, or in a co-located server directly connected to the switch [43]. Either way, the defense is achieved by a division of labor between the fastpath and the slowpath.

2.4 Scenarios, assumptions, and non-goals

Combining these techniques, NetWarden can be easily deployed on a Top-of-Rack (ToR) switch to protect a rack of machines (Figure 1), whether in a cloud datacenter or an enterprise network, as their settings are similar in many aspects (e.g., servers organized in racks, served by ToR switches). It is not necessary that all servers or VMs in a NetWarden-enabled rack must use its service—since covert channels are used to exfiltrate secret data, we expect that the protected machines/VMs are typically sensitive file servers. An operator could easily configure NetWarden to inspect only a subset of the traffic (e.g., to/from the sensitive file servers) and directly forward the rest using regular routing tables.

Assumptions and threat model. Similar as existing work, we assume that NetWarden has access to known-good IPD data collected by the administrator to perform statistical tests. This is a reasonable assumption, because the protected servers are controlled by the network administrator, and the hosted services are typically configured by the administrator. Our threat model is that any layer of the server/VM stack can be compromised by an attacker, who wants to exfiltrate data to an external accomplice via network covert channels. Attackers leaking data explicitly via packet payload are outside our model. We also assume that the NetWarden device is trusted.

Non-goals. We note that the primary contribution of NetWarden is a general system that can support a wide range of existing and new defenses while preserving network performance. As such, improving existing techniques for mitigating specific channels, or designing detection algorithms with higher accuracy, are not our main focus.

3 Performance-Preserving Defenses

In this section, we first describe how NetWarden can support a set of basic defenses that do not consider performance implications. We then characterize how some of them may degrade performance, and design performance-boosting techniques to neutralize the overall impact.

3.1 Programmable data plane defenses

We describe a basic set of defenses that NetWarden can support in the data plane, and explain the hardware features they rely on. Most of the defenses below are simply adapted from existing work, with the exception of Type-III storage channel defenses—they are made possible because NetWarden can precisely monitor every single packet in every connection.

Type-I storage channel defenses. The simplest of storage channels embed covert data into optional or unused fields, such as the TCP reserved bits, optional TCP flags (e.g., URG, NS, ECE), IPID, and TTL. Existing work has developed *deterministic and stateless* defenses, which can be naturally supported by NetWarden's ability to perform linespeed header inspection and modification. For instance, we can set these fields to values configured by the network operator—e.g., clearing reserved bits, substituting the IPID with a random number, and setting TTL to 64. For optional TCP flags that are rarely in use, we can simply clear these bits. To apply these defenses, the operator needs to ensure that the configured values do not break needed functionality (e.g., the TTL should be large enough to avoid premature packet drops).

Type-II storage channel defenses. More advanced storage channels overload header fields that are essential to protocol correctness, e.g., the TCP sequence number and non-optional TCP flags (e.g., SYN/ACK/RST/FIN). Statically setting these fields to fixed values would break TCP semantics; instead, we need *stateful* defenses against these channels. For TCP sequence numbers, we could replace the initial sequence number with another number, record the offset in a table, and consistently apply the same offset to all subsequent packets. For TCP flags, the defense needs to ensure that SYN packets only appear during connection establishment, and RST/FIN packets during teardown. NetWarden can support Type-II defenses due to its ability to modify headers efficiently and the support for stateful tables that can sustain linespeed reads/writes.

Type-III storage channel defenses (new). These channels hide themselves in the inherent non-determinism of protocol behaviors, so they require more sophisticated defenses. For instance, the partial ACK channel [50] can encode data in the offset between the ACK number n and the highest sequence number N seen, i.e., leaking a secret $\delta = N - n$. The receive window size channel embeds secret data into the advertised receive window field in the TCP header; since this value depends on the available buffer size, it may naturally change over the course of a connection. Defenses against these channels need to explicitly handle the non-determinism. Here, programmable data planes play a critical role—NetWarden can track the state of every connection on a per-packet basis to resolve the ambiguity as much as possible. This leads to several new defenses that are unique to NetWarden.

Concretely, NetWarden remembers for each connection the highest sequence number seen, and detects whether a given ACK packet acknowledges the full or a partial sequence space. It then performs *ACK aggregation* to drop partial ACKs and wait for the full ACK to arrive (when the host has processed all received bytes). If the full ACK does not arrive after a timeout period, NetWarden generates an ACK that acknowledges the highest/full sequence number of the previous batch of packets. This would mitigate the partial ACK channel with a tunable amount of extra delay that can be configured by NetWarden. To mitigate the receive window size channel, NetWarden performs *receive window sanitization* to remove the least significant bits of `rwnd`, reducing the number of bits that can be repurposed by a tunable amount, e.g., `rwnd&=0xff00`. Here, ACK aggregation might incur extra delay, and receive window sanitization might potentially limit the sending window growth (if the connection happens to be bottlenecked by the receive window size). Nevertheless, we can configure the amount of delay or window reduction to minimize performance penalty; when needed, we can always boost the performance of the affected connections.

Timing channel defenses. Covert timing channel detectors [21,21,31,57] work by measuring the statistical deviation between a given trace and a known-good trace in terms of their IPD distributions, e.g., using a Kolmogorov-Smirnov test [57]. Upon detection, the defense could add random delay or buffer packets to destroy the IPD modulation. NetWarden can compute IPDs for all connections in hardware, so the software only needs to perform statistical tests and IPD modulation. This is already more efficient than existing detectors that perform both in software; in Section 4, we will further optimize this to avoid sending all IPD data to software.

3.2 Performance boosters

The above defenses always make conservative decisions, so they are always safe. However, some of them could cause performance degradation (discussed later in Section 3.3). Before delving into the details of the performance analysis, we first design a set of defenses that can boost performance—they are essentially "positive twins" of the defenses above. The performance boosters work by manipulating the TCP congestion control mechanism to present false illusions to the sender and receiver, somewhat analogous to "performance-enhancing" proxies [18]. Since TCP tightly couples congestion control with reliability mechanisms, we need to ensure that these defenses do not break the reliability of the transfer.

ACK boosting. This technique aims to counteract the effect of extra delays due to covert channel defense. The primary source of extra delays is the timing channel defense that disrupts IPD patterns by buffering packets. (ACK aggregation only results in small amounts of delay, because TCP usually acknowledges every other packet.) This technique *prefetches* data from the sender by generating ACK packets from NetWarden on behalf of the actual receiver. This defense can be further parameterized by $\delta_t \in [0, RTT]$, which is the interval between the time NetWarden sees a data packet and the time it generates an ACK. The lower-bound 0 comes from the fact that NetWarden cannot proactively acknowledge a packet before it is sent; the upper-bound RTT comes from the fact that, the actual client ACK arrives an RTT later, so applying ACK boosting after an RTT would not be useful. This technique hides the latency for a) the data packet to propagate to the receiver, b) the receiver to process the data and generate the real ACK, and c) for the ACK to propagate back to the sender. In effect, it creates the illusion of a shorter RTT as perceived by the sender, thus ramping up the sending rate faster.

Receive window boosting. This technique counteracts *receive window sanitization*, by enlarging the receive window size field of a packet to create the illusion of a high-performance receiver. A simple heuristic, for instance, is to ensure a similar amount of boosting as the window reduction.

Buffering + proxying. The above two techniques may trigger extra packets. Therefore, NetWarden needs to buffer these packets temporarily in case the receiver does not have sufficient buffer size to process them, or if these packets would be lost in transmission; NetWarden serves them to the receiver from its buffer when needed. The buffered data will be gradually removed when the actual ACKs from the receiver arrive at NetWarden. For ACK boosting, the actual ACKs do not need to be forwarded to the sender, since from the sender's perspective, the corresponding data packets have already been successfully received.

3.3 Performance implications

Next, we explain the performance implications of these defenses and how we can use them in combination to preserve performance. At a high level, TCP performance depends on three factors: a) the amount of available data at the source, b) the receiver's ability to ingest incoming data, and c) the network condition. The TCP sender transmits data in rounds, dumping one window of packets per round-trip time (RTT). The sending window size `swnd` is determined by the minimum of congestion window size `cwnd`, which reflects the network condition, and the receive window size `rwnd`, which reflects the receiver's ability to process new data. The `rwnd` value can be directly retrieved from the TCP packet header, as advertised by the receiver. The `cwnd` value, on the other hand, is computed by the sender for each RTT based on its congestion control algorithm. A wide variety of TCP variants exist, and at the heart of their difference is the congestion signals they rely on, and their algorithms for adjusting the window size.

• **Loss-based congestion control.** Classic TCP variants, such as Reno [14], New Reno [28], and CUBIC [58], respond to packet loss as signals of congestion. A much simplified view of New Reno, for instance, is that it initially sets `cwnd` to be a small constant (e.g., 10 MSS), and then doubles the window size for each RTT, resulting in exponentially larger bursts of packets. After `cwnd` reaches a certain threshold, the growth rate would change from exponential to linear, e.g., by one MSS per RTT. Such window growth would be disrupted if there is packet loss. Loss is detected when the sender has received three duplicate ACK packets, or when no ACK packets have arrived for an extended period of time (i.e., an RTO, or retransmission timeout). Upon duplicate ACKs, the sender cuts back its `cwnd` (e.g., roughly in half). Upon RTO, which indicates more severe congestion, it cuts back the `cwnd` more aggressively (e.g., resetting it to one MSS). In both cases, TCP retransmits the unacknowledged packets until the arrival of new ACKs drives it back to its normal course.

Takeaway #1: Preserving sending dynamics. Suppose that the source has infinite amount of data to send, and that no packet loss happens, then we can statically determine the TCP *sending dynamics*—a series $\mathbf{w}= w_0, w_1, \cdots, w_\infty$, where w_i is the `cwnd` size for the i-th RTT as measured by the number of MSS-sized packets. It is worth noting that only when the source runs out of data or packets get dropped would the sending dynamics change. In particular, the RTT values does not impact the series $\mathbf{w}$. Therefore, as long as our defenses do not cause packet drops, as perceived by the sender in the form of triple duplicate ACKs and RTOs, then we can preserve the sending dynamics and the number of RTTs it takes to transfer a file. If a file has $N\times$MSS bytes, then the number of rounds for the transfer to complete is the smallest k for which $\sum_{i=0}^{k} w_i \geq N$ holds. Consider a timing channel defense, where we buffer each burst of packets by a fixed delay Δ to destroy IPD modulation. From the sender's perspective, it would only perceive a path with an increased latency, i.e., $RTT^+ = RTT + \Delta$, but the dynamics would stay the same.

Takeaway #2: Preserving throughput. However, there is still a performance penalty due to the "increased" RTT. Although TCP takes the same number of rounds to transfer the same amount of data, the absolute value of an RTT has increased due to the mitigation. Assuming this inflates the RTT by Δ, then overall it would increase the flow completion time by $k \times \Delta$, because the throughput for the i-th burst has decreased from $\frac{w_i}{RTT}$ to $\frac{w_i}{RTT+\Delta}$. Therefore, if a defense wants to preserve the throughput of TCP, then it could either a) ensure that (or create the illusion of) $\Delta = 0$, or b) change the sending dynamics of TCP by increasing the burst sizes, with the eventual goal of ensuring $\frac{w_i}{RTT} = \frac{w_i + w_{\delta_i}}{RTT+\Delta}$ where w_{δ_i} is the amount of size increase for the i-th burst.

Applying this takeaway to the defenses, if a defense does not affect the RTT (e.g., Type-I/II defenses), then they already preserve TCP throughput. If a defense increases RTT, then we can either ramp up the sending window by generating boosted ACKs to enlarge each burst size; or, equivalently, we can ensure that the sender perceives the same RTT before the defense, e.g., by injecting ACKs exactly one RTT after a burst is sent. In this way, although the actual receiving time of packets is still delayed by Δ, the delays per batch are masked by the parallelized sending. Without mitigation, the sender sends out the last batch of packets at $k \times RTT$, and they arrive at the receiver at $(k+1) \times RTT$; with mitigation, the sender still sends out the last batch at $k \times RTT$, but they arrive at the receiver at $(k+1) \times RTT + \Delta$. In other words, the overall increase of transfer time is only Δ for the entire transfer.

• **Delay-based congestion control.** Some TCP variants such as TCP Vegas [19] and FAST [40] adjust their `cwnd` based on delay increase rather than loss, so that they can detect the onset of congestion early before loss occurs. TCP Vegas, for instance, keeps track of the lowest RTT seen in a connection, and continues to measure the RTT experienced by a batch of packets. It can then compute whether the current sending rate `cwnd`/RTT is too high or too low, and decrease or increase the window accordingly. In other words, if a defense results in a sudden RTT increase, then TCP Vegas would take this as a congestion signal, and start to reduce its sending window, resulting in a different (slower) sending dynamics.

Takeaway #3: Preserving latency. For these TCP variants, we need to ensure that they do not perceive the extra delay due to covert channel mitigation. One solution for this is to use a stable RTT, e.g., as measured in the beginning of the connection, for all boosted ACKs. This achieves stable performance, but does not account for potential performance changes during the connection. A more advanced method is to measure the RTT continuously, and use the latest measurement results to drive the generation of pre-ACKs, adjusting to RTT changes in real time.

3.4 Principle of maximized transparency

Our final principle for applying these defenses is to make NetWarden as transparent to the end hosts as possible. Concretely, NetWarden always tracks the RTT of a connection, and it periodically relays the most recent RTT value to the sender by generating pre-ACKs at that time. This principle of "maximized transparency" allows us to apply defenses without requiring exact knowledge about the TCP variants in use. By faithfully relaying RTT and loss signals to the sender, NetWarden also minimizes the amount of "discrepancy" between the perceived network condition at the sender and the actual network condition. In other words, NetWarden does not blindly create the illusion of stable RTTs or low loss, but rather adjusts to the network condition in real time.

4 The NetWarden System

Next, we describe the fastpath/slowpath architecture of NetWarden, and how the two components work with each other for covert channel mitigation.

4.1 Design principles

The main research question in architecting NetWarden is to identify the right "division of labor" between the fastpath and the slowpath, and to carve out a proper boundary between the two to minimize crosstalk. Overall, our architecture is centered around three guiding principles.

• *Principle #1: Per-packet operations are pushed down to the fastpath for acceleration, and batch operations are lifted up to the slowpath for generality.*

The data plane hardware is highly-optimized for packet processing. Therefore, operations that need to be performed over every single packet should be offloaded to the fastpath for high efficiency. Operations invoked over batches of packets, on the other hand, usually involve loops or other sophisticated processing; these go beyond the programming model of the data plane. Fortunately, batch operations are usually performed at lower frequency and are not in the critical path for processing, making them a better fit for the software slowpath.

• *Principle #2: Data structures accessed per-packet are implemented in the fastpath. Data structures with constant state are preferred when possible, and data structures whose state could grow over time would require backstore support.*

The slowpath DRAM cannot sustain per-packet memory accesses at linespeed. Programmable data plane hardware, on the other hand, is customized for linespeed memory accesses. Moreover, it is preferable to use data structures whose state is small and does not grow over time. If state could grow in a per-packet manner, the data structure would need to be co-designed with slowpath support, using an abstraction that we call "backstores" (Section 4.3). When needed, fastpath state can be evicted to and fetched back from the slowpath.

• *Principle #3: The frequency and volume of communication between the fastpath and the slowpath should be minimized.*

The interconnect between the fastpath and the slowpath has bandwidth and latency bottlenecks, whether it is a PCIe bus that connects the switch control and data planes, or an Ethernet/RDMA [43] cable that connects the switch to an external server. Therefore, we should design the fastpath/slowpath interface to minimize crosstalk as much as possible.

Individually, both the fastpath and the slowpath have notable limitations, but when taken together, they complement each other. NetWarden combines their respective advantages to achieve an effective defense. Applying these principles to covert channel mitigation results in the following division of labor: The fastpath performs a) connection monitoring, b) IPD characterization and pre-checks, and c) storage channel defense. The slowpath performs a) statistical IPD tests, b) timing channel defense, and c) performance boosting. Figures 2 illustrates these components; we discuss more below.

4.2 The fastpath defense

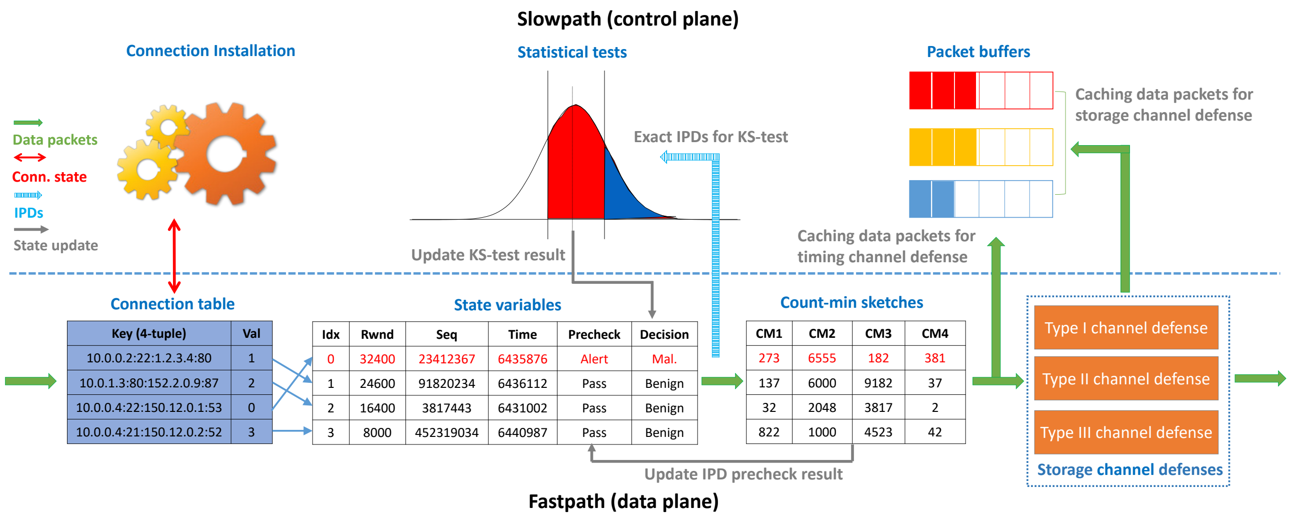
• A key primitive for detecting covert channels is a hardware data structure that monitors every TCP connection.

Connection monitoring. The monitoring table is organized as a key/value store, where the key is a TCP connection's flow ID (i.e., source/destination IPs and ports), and the value is an index to a set of register arrays. Using this index, we can further write into or read from stateful registers that record the TCP state for each direction of this connection, such as a) the highest sequence number seen, b) the timestamp of the last outgoing packet, c) receive window size penalty, and d) an RTT estimate. NetWarden uses a packet's flow ID to index this table, and updates a)-c); for each burst, it computes the timestamp difference between the outgoing and the returning packets to maintain d) an RTT estimate. For new connections, NetWarden sends the SYN packets to the control plane for entry installation. The size of this connection table is predefined at compilation stage to accommodate the maximum number of connections the operator wants to support; its state does not grow at runtime.

• NetWarden has several components for detecting covert timing channels.

IPD computation. Computing IPDs requires per-packet operations, therefore it needs to occur on the fastpath (principle #1). NetWarden leverages the fine-grained timestamp facility in the switch, which provides nanosecond granularity timestamps when packets enter the processing pipeline. Retrieving timestamps is akin to accessing registers, which can be performed at linespeed.

Since every packet would produce additional IPD data, this creates challenges for state maintenance. Directly apply principle #2 above would result in a solution that sends all IPD data to the software slowpath. However, this would create very high communication overheads between the fastpath and



Key (4-tuple)	Val
10.0.0.2:22:1.2.3.4:80	1
10.0.1.3:80:152.2.0.9:87	2
10.0.0.4:22:150.12.0.1:53	0
10.0.0.4:21:150.12.0.2:52	3

Idx	Rwnd	Seq	Time	Precheck	Decision
0	32400	23412367	6435876	Alert	Mal.
1	24600	91820234	6436112	Pass	Benign
2	16400	3817443	6431002	Pass	Benign
3	8000	452319034	6440987	Pass	Benign

CM1	CM2	CM3	CM4
273	6555	182	381
137	6000	9182	37
32	2048	3817	2
822	1000	4523	42

Figure 2: The architecture of the NetWarden system.

the slowpath. NetWarden instead designs four optimization techniques to reduce state growth as much as possible, and only invokes the slowpath to monitor connections that might contain covert timing channels. Specifically, a) IPD intervalization prevents state from accumulating per packet, b) IPD sketching further reduces the state using approximation, c) IPD pre-checks perform simple range checks in hardware, and d) we only send exact IPD data to the slowpath if a connection is labeled by the pre-checks as suspicious.

IPD intervalization. This technique trades off some IPD accuracy to prevent per-packet state growth. Concretely, we can keep the *distribution* of the IPDs instead of the exact IPD values. This can be achieved by, for instance, maintaining k counters for a fixed numbers of IPD intervals $[0, t_2)$, $[t_2, t_3)$, $\cdots$, $[t_k, \infty)$, and incrementing the counter for a particular interval for each computed IPD. These intervals are constant in state and do not grow over time, which is already a step forward. However, keeping a set of counters for each connection still requires a significant amount of memory resources.

IPD sketching. We further avoid the need of keeping per-connection intervals using *sketching*, which trades off per-connection granularity for space savings using count-min sketches (CMSketches) [23]. At a high level, a CMSketch consists of an array of counters that can be shared by all connections. Instead of using k counters for each connection, we could use k CMSketches for all connections. If an IPD falls into $[t_i, t_{i+1})$, we increment the counter for the corresponding connection in the i-th sketch. To increment the counter for a connection, the CMSketch first computes h CRC hash values of the connection/flow ID, obtaining $i_1 = CRC_1(conn), \cdots, i_h = CRC_h(conn)$. It then uses i_1-i_h as indexes into the counter array $c[\cdot]$, and increments the respective counters $c[i_1], \cdots, c[i_h]$. To retrieve a counter for a connection, we similarly compute h indexes using the same CRC functions, and use the minimum value as the estimate: $\min\{c[i_1], \cdots, c[i_h]\}$. Though simple, CMSketches provide strong accuracy guarantees [23]. CRC hash functions and counters are supported by the switch hardware, as they are needed by functions like load balancing. NetWarden leverages these features to perform IPD sketching entirely in hardware.

IPD pre-checks. NetWarden also performs simple range checks on the IPD distribution as a first-pass detection. Periodically (e.g., for every i-th packet in a connection), NetWarden queries the IPD distributions from the CMSketches and compares them with the known-good distribution. These pre-checks only involve arithmetic comparisons, which are supported by the switch hardware. If a connection exhibits a notably abnormal deviation from the expected distribution, NetWarden would label the connection as suspicious and perform the next step for exact IPD monitoring.

Selective exact IPD monitoring. Connections that exceed the pre-check thresholds are subjected to tighter scrutiny. NetWarden performs software-based statistical tests for these connections in the slowpath using exact IPD data. NetWarden skips the IPD intervalization and sketching steps for suspicious connections, and directly inserts them into a separate table instead. For all connections in this table, NetWarden sends all computed IPDs without any approximation in order to achieve full fidelity.

• Performance-degrading defenses against storage channels can be fully supported on the fastpath.

Performance-degrading defenses. These defenses modify headers of outgoing packets and set them to controlled values. These operations are needed per packet and they involve constant (Type-II/III) or no (Type-I) state. Per principles #1 and #2, such defenses are hosted on the fastpath.

4.3 The slowpath defense

• Statistical IPD tests and timing channel mitigation only need to be performed occasionally over batches of packets, so NetWarden hosts them in the software slowpath.

Statistical IPD tests. This component works together with IPD pre-checks on the fastpath. It receives exact IPDs from connections identified by the pre-checks as potentially malicious, and performs full-blown statistical tests for timing channel detection. NetWarden can easily support existing detectors (e.g., KS test [57]) or new detectors that may be developed in the future. These statistical detectors measure the distance between a given IPD distribution and the known-good distribution, and raise alarms if the distance exceeds a pre-defined threshold. Upon detection, NetWarden would apply mitigation techniques to the detected connections.

Timing channel mitigation. This component buffers packets in suspicious connections identified by the statistical tests to disrupt the timing modulation. NetWarden temporarily holds a burst of packets in a cache and sends them out back-to-back when a timer fires. The buffering time can be configured by the network administrator.

• All performance boosters require slowpath support, because they may cause extra data packets to be transmitted. This in turn requires temporary buffering and proxying.

Backstores. The key abstraction NetWarden provides in the slowpath is backstores. A backstore provides support for the defense by mapping a connection to its relevant state and pointing to functions that need to be applied on this state. NetWarden has three backstores: two for boosting the performance of connections that have gone through timing and storage channel mitigation, respectively, and a third for statistical IPD tests. The fastpath and the slowpath communicate with each other through these backstores by sending network packets (for packet buffers) and by a hardware mechanism in the switch called "digests" (for IPD data; see Section 5.1). The fastpath could send data to the respective backstore by tagging the packets using the backstore ID (e.g., IPD data for statistical tests). The slowpath could also inject packets from the backstore back to the fastpath (e.g., pre-ACKs for boosting performance).

The first backstore keeps a periodic timer for the connection, whose value is set to the connection's estimated RTT; it also keeps a list of buffered packets. NetWarden uses these timers to trigger pre-ACKs for maintaining the TCP sending rate. The extra packets that are triggered by the boosting would be appended to the packet buffers. The second backstore buffers packets for receive window boosting. Here, enlarging the window size can be performed entirely by the fastpath. However, this may trigger extra packets that the receiver is not yet ready to process. NetWarden appends these packets to the buffer in case they will not be successfully received. The third backstore is for statistical tests, whose purpose we have already explained above. NetWarden supports this using the same backstore abstraction, which maps connections to IPD data, and includes function pointers to statistical tests as well as timing channel mitigation.

4.4 Self defense

Principle #3 allows us to systematically understand the conditions under which the fastpath and the slowpath may communicate. This further enables us to identify traffic patterns that would create expensive processing in software, and monitor these patterns to guard against potential attacks. Specifically, two of NetWarden's backstores buffer packets for mitigating timing and storage channels, and the third backstore keeps IPD data for statistical tests. Apart from these backstores, NetWarden also invokes the software for inserting new connections to the monitor table (Section 4.2). This leads to three potential attack vectors we should protect against.

Bufferbloat attacks. An adversary could intentionally cause a large amount of packet buffering to launch a memory-based denial-of-service attack to NetWarden. When boosting performance (whether for timing or storage channel defense), NetWarden needs to buffer the extra packets temporarily. An attacker can pretend to never receive the data packets, e.g., by always using old acknowledgment numbers to cause perpetual buffering. To mitigate this, NetWarden monitors its cache usage to detect signs of attacks. If a connection buffers data in the backstore excessively without making progress in the transfer (e.g., packets are never or very slowly acknowledged), NetWarden can simply shut down the connection using RST packets.

Excessive IPDs. In the common case, NetWarden maintains timestamps in sketches; but exact IPDs are exported to the slowpath if a connection has suspicious patterns. An adversary may also intentionally modulate packets to cause many timestamps to be sent to the slowpath, resulting in excessive communications. This would overwhelm the communication channel between the fastpath and the slowpath, interfering with the installation of new connections that needs to go through the same channel. To defend against this, NetWarden monitors the amount of suspicious connections and ensures that they are always under a pre-defined threshold; abnormally large counts would trigger alarms to the network operator and NetWarden would shut down all subsequent connections labeled as suspicious.

Connection table flooding. Aside from the backstores, the only other operation that would trigger software processing is new connection insertion. Upon seeing a new connection, NetWarden sends the new connection information to the control plane using "digests", so that the corresponding entry will be populated in the connection table. An adversary may maliciously generate many new connections to overwhelm the control plane and to occupy entries in this connection table. This is akin to a SYN flooding attack, against which well-established defenses exist [10]. In addition to relying on these defenses, NetWarden also rate limits the number of connections that an IP address can establish, and clears connections that have not been active for a long time.

5 Evaluation

Our evaluation of NetWarden is designed to answer four key research questions: a) how much overhead does NetWarden introduce? b) how effective is NetWarden in detecting and mitigating covert channels? c) how well can NetWarden preserve network performance when mitigating channels? and d) how well can NetWarden support real-world applications?

5.1 Prototype implementation

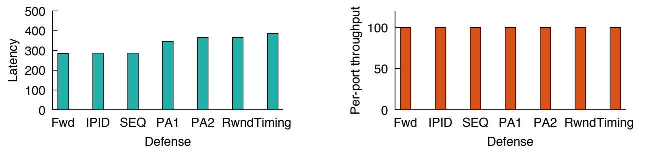
We have built our NetWarden prototype using ∼5500 lines of code. The fastpath contains 2500 lines of code in P4, and the slowpath contains 3000 lines of code in C and Python. Our prototype can defend against six types of network covert channels: a) a Type-I storage channel that embeds data into the IPID field, b) a Type-II storage channel that embeds data into the TCP sequence number, c) three Type-III storage channels, including the receiver window size channel, and two variants of partial ACK channels (one acknowledging sequence numbers contained in received packets, and another acknowledging any offsets in a packet), and d) covert timing channels.

The fastpath of our prototype runs in a hardware switch. Our switch has a hardware mechanism called "digests", which can compress per-flow data (e.g., IPDs) and send new connections to the switch control plane for installation. Therefore, we have implemented the IPD statistical checks and the logic for installing new connections directly in the switch control plane. The packet buffer, on the other hand, needs to receive and re-inject entire packets; in order to provision more bandwidth for packet buffering, we run this component in a server that is connected to the switch via a 25Gbps Ethernet cable.[1] This component buffers and proxies packets for covert channel defense, and it is the overall bottleneck of the system.

5.2 Experimental setup

We have conducted a series of experiments by deploying NetWarden to a Wedge 100BF-32X Tofino switch, which has 32× 100Gbps ports and can be programmed in P4. It is configured as a Top-of-Rack switch in our cluster, and one of these switch ports is connected via a 100Gbps-to-25Gbps breakout cable to the slowpath server. The server that hosts sensitive data is also directly connected to the NetWarden switch, and it communicates with remote clients over emulated wide-area network links with realistic latency, jitter, and loss rates. All machines in our experiments have a six-core Intel Xeon E5-2643 CPU, 16 GB RAM, 1 TB hard disk, and

[1] A recent work [43] shows that a more efficient approach would be to connect the P4 switch to the server using RDMA, which can achieve 1–2μs latency and 34Gbps throughput over a 40Gbps NIC. Our current prototype uses `libpcap` to capture Ethernet packets and is bottlenecked by this library; as a result, it can only achieve a fraction of the full bandwidth (25Gbps) the NIC can support. As future work, this cited work could be a drop-in replacement for our slowpath/fastpath communication as performance optimizations.



(a) Latency (nanoseconds) (b) Throughput (Gbps)

Figure 3: NetWarden incurs extra latency on the order of nanoseconds, and it achieves linespeed throughput.

they are installed with an Ubuntu 18.04 OS with the default TCP version (CUBIC). The attacker has full control over the sensitive server, and can modulate any outgoing packets to leak secret data. The attacker's goal is to exfiltrate a 2048-bit RSA key via covert channels.

Workloads. For a comprehensive evaluation, we have used three sets of workloads in our experiments. In Sections 5.3–5.4, we perform a set of microbenchmarks and overhead evaluations using synthetic traces as a "stress test" of NetWarden. In Sections 5.5–5.7, we adopt the widely used DCTCP workloads [13], which represent the traffic characteristics of a production-scale data center. The same workloads have been used in many previous projects for evaluation [12, 29, 34, 36, 38, 48, 55, 64]. In Section 5.8, we further evaluate NetWarden using a set of real-world applications, including Apache web server, Node.js, and FTP, to understand how well NetWarden can support complex systems.

5.3 Microbenchmarks

We start by performing a set of microbenchmarks using synthetic traces that are designed as a "stress test". This subsection focuses on measuring the performance of the fastpath, and the next subsection measures the fastpath/slowpath communication and the slowpath overheads.

Maximum number of connections. The first metric we have used is the maximum number of connections that NetWarden can support. Unlike software programs, where increasing the program size merely results in a somewhat slower program, P4 programs are mapped to the hardware by the compiler in an "all-or-nothing" flavor. The P4 compiler ensures that only programs that fit within the resource constraints would compile to the switch, and that such programs are guaranteed to run at linespeed. On the other hand, programs that exceed the maximum amount of available resources would be rejected at compilation time. Therefore, the maximum number of connections a P4 program can support is determined at compilation time rather than runtime. To measure this, we gradually increased the number of connections in NetWarden's connection table, which resulted in larger and larger program sizes, until the P4 compiler rejected the program due to insufficient switch resources. We found that the compiler successfully compiles and maps NetWarden to the switch up to 200 k connections. This is larger than the maximum num-

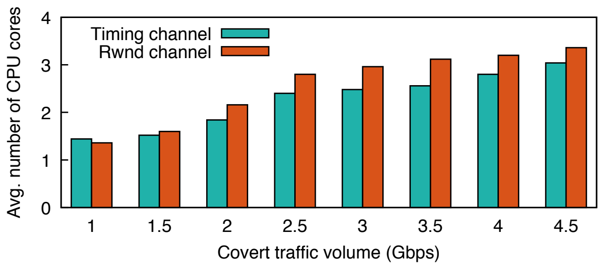


Figure 4: The compute overhead of NetWarden slowpath.

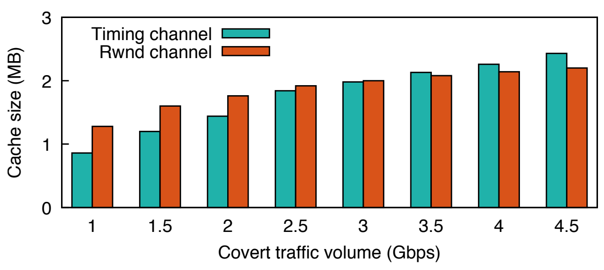


Figure 5: The memory overhead of NetWarden slowpath.

ber of active connections in typical ToR switches in Facebook frontend clusters (10k-100k) [54].

Latency. We then measured the extra latency of NetWarden, using a baseline system ("Fwd") that runs an "empty" P4 program that simply forwards packets without any other processing. Figure 3a shows the results. As we can see, Type-I/II storage channels incur the least amount of overhead, because their defenses simply perform header modifications or table lookups. Type-III storage channels have higher overhead, because they require keeping a larger amount of state for each connection and updating these states per packet.[2] Timing channels have the highest overheads because they have more complex logic for IPD computation, sketching, and pre-checks. Nevertheless, compared to the baseline program, NetWarden defenses lead to an extra delay from 3-101 nanoseconds. Since the RTT of a typical network path is on the order of milliseconds, this extra delay is negligible.

Throughput. Despite the slight latency increase, the pipelined nature of the switch hardware can hide latency per packet by parallelizing the processing. As Figure 3b shows, the throughput of NetWarden is stable at about 99.98Gbps per port across scenarios; the maximum bandwidth per port is 100Gbps.[3] These results are expected, because the P4 compiler guarantees that all programs that successfully compile would run at linespeed.

These microbenchmarks demonstrate that NetWarden can indeed process linespeed traffic with negligible overheads. This property alone already sets NetWarden apart from all existing covert channel defenses that run in software.

5.4 Fastpath/slowpath overheads

Next, we evaluate the overhead of the fastpath/slowpath interaction in the presence of different types of covert traffic, as well as the compute and memory overheads of the slowpath for packet caching and performance boosting. As discussed, the IPD statistical tests and new entry installation are performed in the switch control plane, and these operations happen very occasionally; the packet buffering and proxying defense resides in the slowpath on the server, which we have confirmed to be the scalability bottleneck. Therefore, the following measurements stress test this packet buffer.

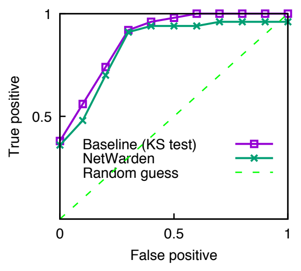
We gradually increased the amount of malicious traffic to trigger more and more processing in the packet buffer until it cannot keep up (i.e., incurs packet loss), and measured the maximum bandwidth for covert traffic. Since Type-I/II defenses are performed entirely in the fastpath, they do not incur any overheads at the slowpath. Also, we found that the new defenses we proposed in NetWarden against the partial ACK channels (PA1 and PA2) only lead to very small delay increase and do not need ACK boosting or slowpath involvement. Therefore, below, we show the results for the covert timing channel and the receive window size storage channel.

The fastpath/slowpath communication. This experiment measures the maximum amount of covert traffic the packet buffer can process. We started by testing whether the bottleneck comes from the slowpath processing speed, or the fastpath/slowpath communication. Our results show that the communication, not the slowpath logic itself, is the bottleneck. Whereas the Ethernet connection has a 25Gbps throughput, the `libpcap` packet capture utility was only able to sustain 4.5Gbps traffic without causing packet loss, both for the covert timing and storage channels. The maximum number of new flows per second NetWarden can sustain is 1200, which is larger than the medium flow arrival rate (500 new flows per second) reported by Facebook for popular services [60]. As mentioned, existing work [43] has shown that using an RDMA connection could achieve much higher bandwidth between the P4 switch and the server (34Gbps over 40Gbps NIC). This is an interesting optimization that we leave to future work.

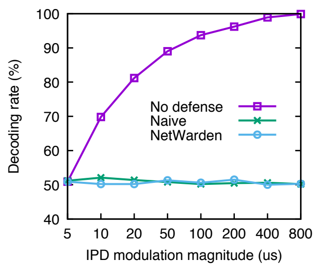
Slowpath overheads. We then measured the compute and memory overheads of the slowpath due to covert channel defense. Figure 4 shows the amount of CPU overheads at different volumes of covert traffic. As we can see, even with the maximum volume, NetWarden only uses roughly 3.5 out of 24 available CPU cores. This is good news, because it shows that the slowpath logic itself is not compute intensive. Therefore, if we adopt a higher-performance RDMA connection, NetWarden still has enough CPU resources to scale the slowpath throughput much further. Similarly, Figure 5 shows how the size of the packet cache used by NetWarden grows with

[2]PA1 (partial ACK channel variant 1) acknowledges arbitrary offsets in a packet; PA2 acknowledges exact packet boundaries, so it is more stealthy.

[3]This stress testing was performed using a hardware traffic generator in the switch, as software packet generators cannot saturate the switch linespeed.



(a) ROC curves



(b) Decoding success rates

Figure 6: NetWarden can detect and mitigate covert timing channels effectively. Its detection performance is similar as the KS test. When there is no defense, the channel decoding rate can achieve almost 100% when the IPD modulation is heavy (>400μs). When either the baseline defense (KS test+mitigation) or NetWarden is deployed, the decoding rate drops to ~50% (a random guess) for all levels of modulation.

covert traffic volume. In the worst-case scenario, the cache size is only 2.4MB, which is only a fraction of the available memory of the slowpath.

5.5 Mitigating covert channels

Next, we evaluate the effectiveness of NetWarden in detecting and mitigating covert channels using the DCTCP trace. Similar as [22], we assume the external accomplice is very close to the compromised machine; this gives the attacker advantage in achieving robust decoding.

Timing channel detection. We measured the effectiveness of NetWarden's timing channel detector by launching a set of flows using the DCTCP workload, where half of the flows are benign and the other half are modulated by the attacker to leak data. The amount of modulation ranges from 1μs-100μs. The baseline detector would send all IPDs to the slowpath, which then performs the statistical tests. NetWarden, on the other hand, first performs pre-checks and only invokes the slowpath for suspicious flows. In both cases, we have adopted KS test as the statistical detector, as it has been shown to be effective by existing work [21]. We obtained a ROC (Receiver Operating Characteristics) curve for each detector by tuning its detection threshold and measuring its false positive and true positive rates at different operating points. As we can see from Figure 6a, NetWarden and the baseline detector have similar effectiveness in detecting timing channels.

Timing channel mitigation. We then measured the effectiveness of NetWarden in mitigating timing channels. We have tested scenarios where the IPD modulation ranges from 5-800μs. Heavy modulations will make the channel decoding rate higher, but they are also easier to detect. Light modulations are just the opposite. As shown in Figure 6b, without any mitigation ("no defense"), the attacker is able to leak data successfully when the modulated IPDs are larger than 20 μs, as the remote receiver can decode the covert message with high success rates (>80%). When the modulation is larger than 400μs, the decoding rate is almost always 100%. With either the performance-degrading defense ("Naïve") or NetWarden, we can destroy the channels and render the decoding close to random guesses (decoding rate: ~50%).

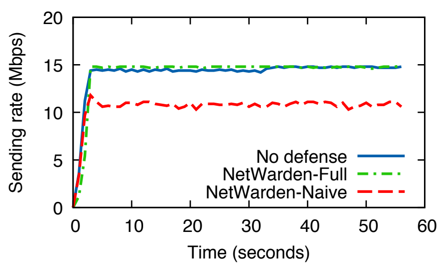
Covert storage channels. Next, we measured the effectiveness of NetWarden in defending against storage channels. Our baseline systems are a) "no defense", which represents the scenario where there is no NetWarden defense, and b) naïve defense, where we set header fields to conservative values. We found that, without any defense, the attacker can easily leak secret data via header fields within a few packets. The channel rates are 16 bits per packet for the IPID channel, 32 bits per flow for the TCP sequence number channel, 11 bits per packet for the partial ACK channel, and 16 bits per packet for the receive window size channel. We found that both NetWarden and naïve defenses have detected the covert channels, and that they have eliminated the IPID, TCP sequence number, and partial ACK channels; for the receive window size channel, both the naïve defense and NetWarden have reduced the channel rate to 2 bits per packet.

5.6 Performance preservation

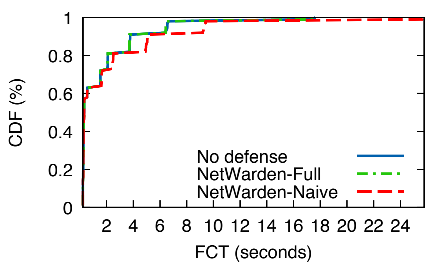
Next, we evaluate NetWarden's ability to preserve network performance while mitigating covert channels using the DCTCP trace. We have used the Linux `tc` tool to emulate realistic wide-area network links with jitter and loss rates, so that we can evaluate the ability of NetWarden to handle network "noise". In our experiments, we have tested different combinations of these parameters. Most of the results presented below are obtained under an average RTT of 10ms, path jitter of 1ms, and loss rate of 0.1%, unless explicitly stated otherwise. This setup closely mirrors the Service Level Agreement of a major ISP [8].

Our main metrics are the sending rates and the flow completion times (FCT) of the TCP flows under a) the "no defense" baseline, b) the performance-degrading countermeasures in NetWarden (NetWarden-Naïve), and c) full NetWarden with performance boosting (NetWarden-Full). An important note is that, a *truly* naïve defense that corresponds to the state of the art would be to perform the same defense techniques in software. These defenses would incur very high overheads just by processing the packets. The defenses labeled as "NetWarden-naïve" are already much more powerful than the actual software solutions—NetWarden enables them to run in programmable data planes with very low latency.

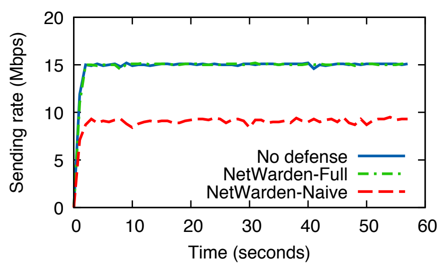
Covert timing channels. We start by evaluating covert timing channels. Figure 7a shows the sending rates over time for a long network transfer; we have enlarged the size of this flow in order to present the sending rate over a longer period of time. As we can see, if NetWarden only applies the naïve



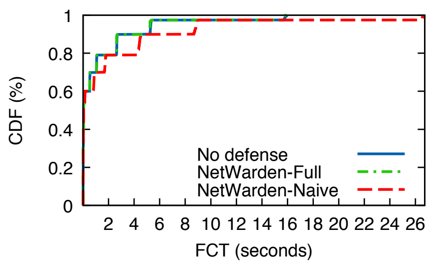
(a) Covert timing channel (sending rate)



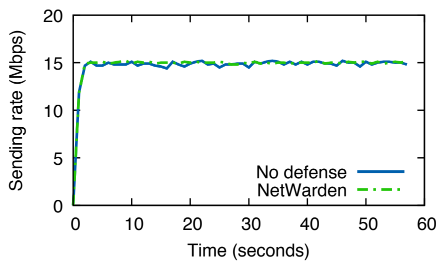
(b) Covert timing channel (FCT)



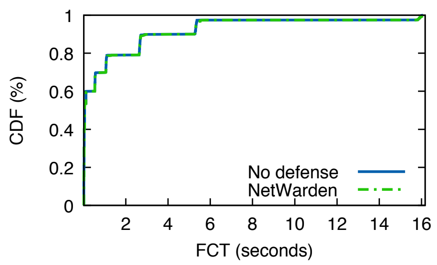
(c) Receive window channel (sending rate)



(d) Receive window channel (FCT)



(e) Partial ACK channel (sending rate)



(f) Partial ACK channel (FCT)

Figure 7: The performance-boosting defenses in NetWarden can preserve the performance of network transfers while mitigating network covert channels. The naïve defenses in NetWarden—although they are already much more advanced than the state-of-the-art defenses—still may cause significant performance degradation because they always take conservative countermeasures. NetWarden enables new defenses against the partial ACK channels due to its precise monitoring capability, and these defenses can preserve performance. The FCT (flow completion time) results are obtained using the DCTCP workloads; the sending rate results are obtained by enlarging the size of a representative flow to show the stable sending rate.

defense that simply disrupts timing modulation of covert traffic, this would cause a significant (25%) degradation in the average sending rate compared to the "no defense" baseline. In contrast, the performance-boosting techniques in NetWarden can achieve a very similar sending rate throughout the duration of this flow, because it carefully masks the perceived RTT increase using ACK boosting.

Figure 7b further shows the CDF of the flow completion times for all tested flows in DCTCP. In aggregate, only applying the naïve strategies in NetWarden would negatively distort the performance characteristics of network transfers. The average FCT across all flows has increased by 9.8% compared to the "no defense" baseline. In the worse-case scenario, the FCT increase could be as much as 47.7%. The full version of NetWarden, on the other hand, causes a 0.06% deviation from the baseline and a worst-case degradation of 1.8%. These aggregate FCT results are consistent with what we observed for individual flows.

Covert storage channels. We then tested the Type-III covert storage channels.[4] Figure 7c shows the sending rate of a long flow under the defense against the receive window size channel. Using only the naïve defense, NetWarden always needs to set the window size to a smaller value, so it incurs a 40% drop in terms of sending rate as compared to the baseline. The full NetWarden, on the other hand, can counteract the penalty by enlarging the window size of certain packets to preserve performance; its sending rate at stable state is almost always the same as the baseline. Figure 7e shows the same experiment under the partial ACK channel defense.[5] NetWarden enables the ACK aggregation defense to run at very small extra latency, so the resulting defenses already achieves a similar performance as the "no defense" baseline.

Figures 7d and 7f show the CDFs of flow completion times for all DCTCP flows. The naïve defense in NetWarden against the receive window size channel increases the average FCT by 28.4%, whereas the full version of NetWarden only increases the average FCT by 0.4%. For the partial ACK defense, NetWarden only increases the FCT by 0.5%.

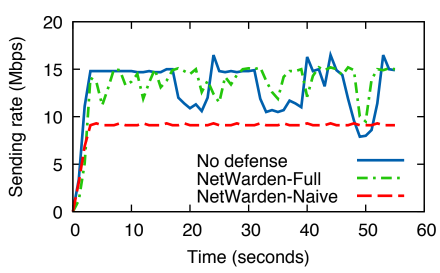
5.7 TCP variants

The above experiments use the default TCP version in Linux: CUBIC. Next, we test NetWarden with three more TCP variants to understand how well NetWarden can support other variants. We have configured the OS to run TCP Vegas, New Reno, and Westwood for long transfers, and these variants mainly differ in their congestion control signals and algorithms. Figures 8a, 8b, and 8c present the sending rates for each variant under timing channel defense.

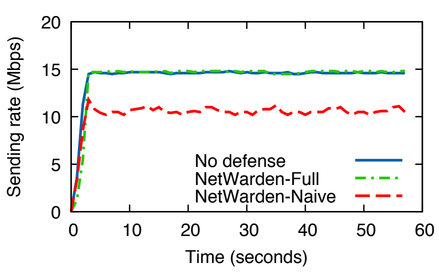
TCP Vegas reacts to delay variation, so we have tested

[4]Defenses against Type-I/II storage channels do not affect performance.

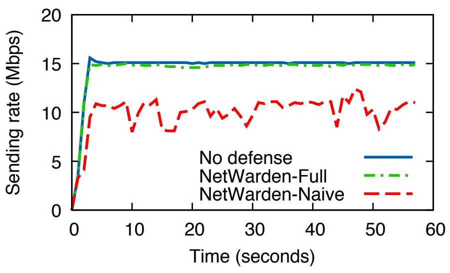
[5]PA1 and PA2 have similar results, and we show results for the latter.



(a) Vegas (20ms, 5ms, 0%)



(b) New Reno (10ms, 1ms, 0.1%)



(c) Westwood (15ms, 3ms, 0.1%)

Figure 8: NetWarden preserves the sending rates of the native transfers for TCP Vegas, New Reno, and Westwood. We have used different (RTT, jitter, loss) configurations to test a range of network conditions.

it with high jitter (5ms, 25% of RTT). As we can see from Figures 8a, the "no defense" baseline fluctuates with time because of this significant jitter. NetWarden exhibits similar fluctuations and deviates from the baseline only by 3.3% in terms of the average sending rate. Note that the fluctuations do not perfectly align with the baseline—this is expected because the jitter is random. The naïve defense, on the other hand, has a very different sending pattern. It has a 31.4% performance penalty; moreover, interestingly, the extra delay due to the defense has reduced the relative jitter, so its sending rate is quite stable and oblivious to the changing network conditions.

New Reno adjusts its sending rate based on packet loss, and we have tested it with a loss rate of 0.1%. As shown in Figure 8b, the high-level takeaways are similar as those in CUBIC. NetWarden experiences a 0.3% performance degradation, but the naïve defense has a penalty of 27.3%.

Westwood, on the other hand, adjusts its congestion window using the estimated bandwidth (obtained by RTT measurements) upon packet loss, so both delay and loss play a role. We have tested it using 3ms jitter and 0.1% packet loss rate. Figure 8c shows that NetWarden performs similarly as the "no defense" baseline, with 1.8% performance degradation. The sending rate of naïve defense, on the other hand, fluctuates over time. We found that this is because the extra delay incurred by the defense has caused occasional full sending windows, leading to a 31.9% performance degradation.

In summary, NetWarden can consistently preserve the performance of the transfer under each tested TCP variant. In contrast, the naïve defense cannot adjust to network conditions, and it always leads to performance penalty.

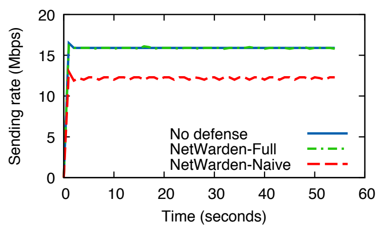
5.8 Complex applications

In the next set of experiments, we evaluate how well NetWarden can support complex, real-world applications, including unmodified versions of Apache HTTP server, Node.js, and FTP. Our most complex application, the Apache HTTP server, consists of 1.49 million lines of code. For our experiments, we have created workloads based on the distributions reported in Facebook [60] and HP [15]. We have performed uploads and downloads for more than 1000 times overall, and in all cases, these applications successfully processed the requests through NetWarden, showing that NetWarden can support complex applications smoothly with realistic workloads.

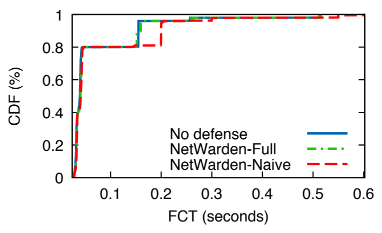
Figure 9 shows the FCT results, as well as the sending rates for several long transfers (with enlarged file sizes for each workload to show the stable rates). Different from the DCTCP trace, these application workloads [15,60] have smaller object sizes and some file transfers are too short to leak the secret data (or trigger defenses). For the transfers that did trigger the covert channel defense, the naïve defense in NetWarden has caused average FCT degradations of 12.2% (Apache), 12.3% (Node.js), and 7.8% (FTP); the worst-case degradations are 36.2% (Apache), 16.9% (Node.js), and 30.4% (FTP). In comparison, the full version of NetWarden has only caused average FCT degradations of 0.1% (Apache), 0.1% (Node.js), and 0.3% (FTP), and worst-case degradations of 2.8% (Apache), 2.7% (Node.js), and 3.0% (FTP).

For the partial ACK channel, the new defenses in NetWarden can also mitigate the channel while achieving a similar level of performance (FCT deviation: 0.1%-0.6%). An interesting finding for the receive window size channel is that, these applications are highly-optimized to process incoming requests as fast as possible; under the request loads our testbed was able to generate, the receive buffer size did not become a bottleneck for these applications. Therefore, we can see that NetWarden-Naïve already achieves a similar level of performance as the "no defense" baseline (FCT deviation: 0.2%-1.7%). As before, it is worth noting that a *truly naïve* defense that reflects the state of the art would need to intercept and process all packets in software.

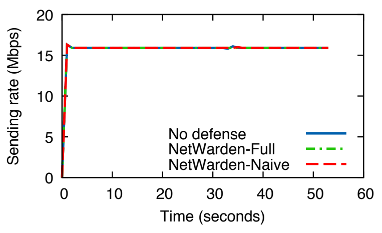
For the covert timing channel, we have similarly obtained ROC curves for NetWarden and the baseline KS test; and we have measured the decoding success rates under different levels of timing modulation. Figure 10a shows that NetWarden performs similarly as the baseline KS test on the tested applications. Figures 10b-10d further present, per application, the effectiveness of the mitigation. In all cases, NetWarden can disrupt the timing modulation and reduce the channel decoding nearly to a random guess.



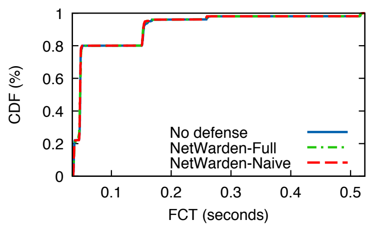
(a) Timing channel (sending rate; Apache)



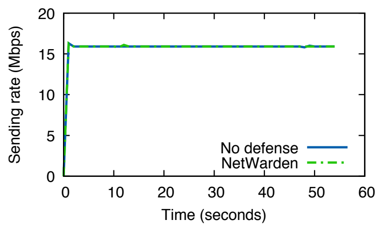
(b) Timing channel (FCT; Apache)



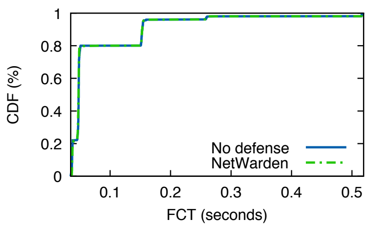
(c) Receive window (sending rate; Apache)



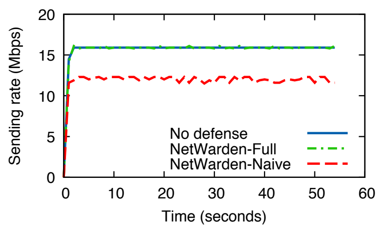
(d) Receive window (FCT; Apache)



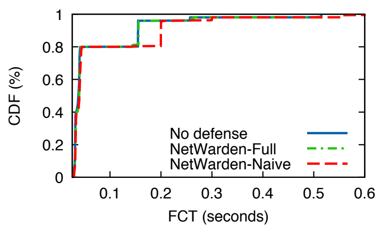
(e) Partial ACK (sending rate; Apache)



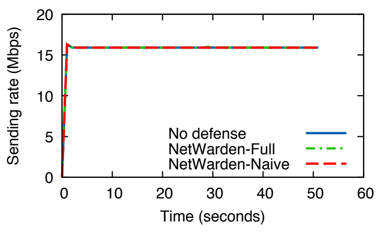
(f) Partial ACK (FCT; Apache)



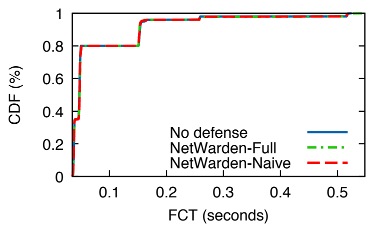
(g) Timing channel (sending rate; Node.js)



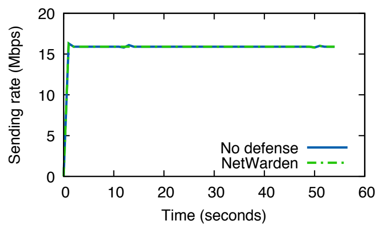
(h) Timing channel (FCT; Node.js)



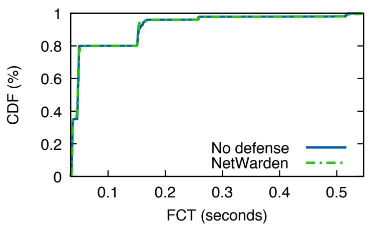
(i) Receive window (sending rate; Node.js)



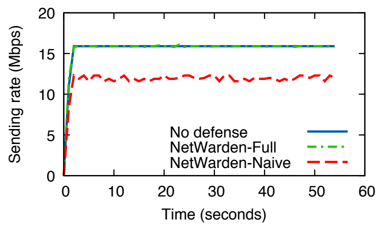
(j) Receive window (FCT; Node.js)



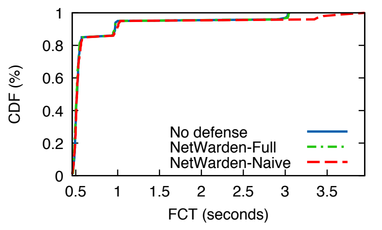
(k) Partial ACK (sending rate; Node.js)



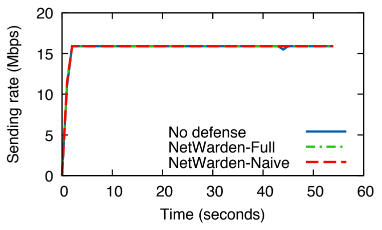
(l) Partial ACK (FCT; Node.js)



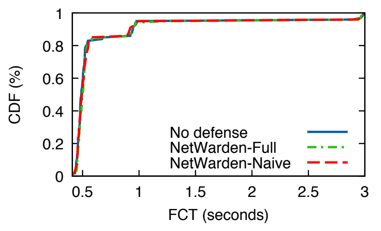
(m) Timing channel (sending rate; FTP)



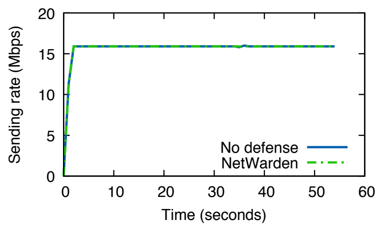
(n) Timing channel (FCT; FTP)



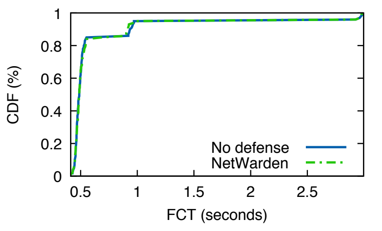
(o) Receive window (sending rate; FTP)



(p) Receive window (FCT; FTP)

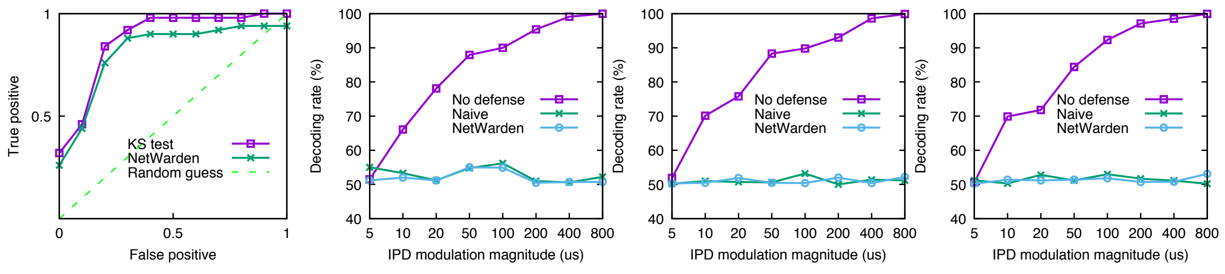


(q) Partial ACK (sending rate; FTP)



(r) Partial ACK (FCT; FTP)

Figure 9: NetWarden can support complex applications and workloads (Apache web server, Node.js server, and FTP) smoothly with minimal performance disturbance. The high-level takeaways are similar as those in the DCTCP trace.



(a) ROC curves (All) (b) Decoding rates (Apache) (c) Decoding rates (Node.js) (d) Decoding rates (FTP)

Figure 10: NetWarden can detect and mitigate covert timing channels for real-world, complex applications.

5.9 Self defense

Last but not least, we evaluate how well NetWarden can identify and block malicious traffic that is intended as attacks to the slowpath.

Bufferbloat attacks. In this attack, the adversary causes a large amount of buffered packets (e.g., by never acknowledging their receipt) in the slowpath. In contrast, for normal connections, the buffered packets would be removed by ACKs roughly one RTT after. Therefore, NetWarden uses a self-defense technique where it monitors the cache usage for each connection, and proactively resets connections whose cache size grows beyond a pre-defined threshold. Figure 11a shows the slowpath memory usage under such an attack that started five seconds after the connection was established. As we can see, without self defense, the adversary can cause the cache size to grow very quickly. The self defense in NetWarden can recognize such abnormal patterns and reset this connection.

Excessive IPD attacks. In this attack, the adversary modulates the packet timing to send a large amount of IPD data to overwhelm the communication channel between the slowpath and the fastpath. The self defense in NetWarden enforces upperbounds on the maximum number and rate of "covert timing channel" connections; it raises an alarm if too many covert channels are identified and shuts down the malicious flows. Figure 11b shows how the attack affects normal user flows at different attack strengths (as measured by the number of IPDs per second). Without any defense, the excessive IPDs would quickly overwhelm the slowpath/fastpath communication; as a result, normal users cannot establish new connections, because they need to be sent to the control plane via digests for entry installation (Section 4.2). A large percentage of them are dropped when the attack strength is high. Using the self defense technique, NetWarden can block these excessive IPDs and protect normal user connection establishments.

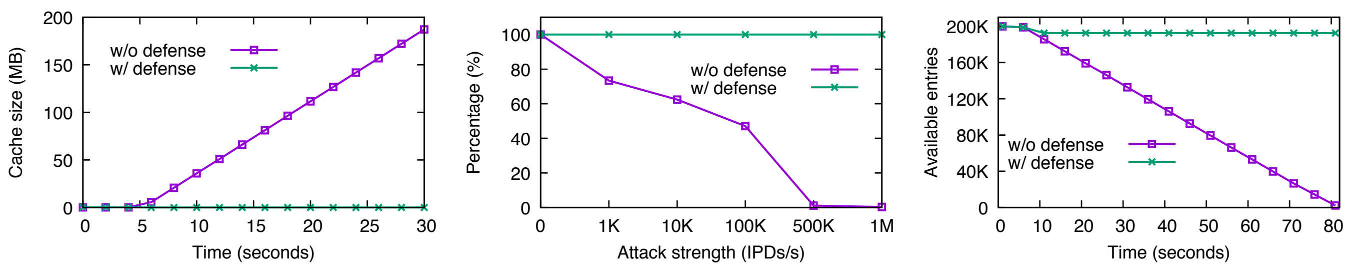
Connection table flooding attacks. Here, an adversary can launch a large number of connections to flood the connection monitoring table, which can support a maximum of 200 k active connections (Section 5.3). The self defense in NetWarden enforces a rate limit for the maximum number of connections that an IP address can establish. The NetWarden control plane also periodically scans this connection table (using a switch feature that identifies the ages of connection entries) and removes inactive flows. Figure 11c shows that, without defense, the available space in the connection monitoring table decreases quickly; eventually, the connection monitoring table becomes fully occupied by the attacker's flows, so that normal users cannot establish new flows any more. The self defense can effectively limit the amount of entries that a single user can occupy.

6 Related Work

Normalizers. Normalizers aim to eliminate ambiguities in protocol payloads, which can lead to attacks when they are interpreted inconsistently by intrusion detection systems and end hosts. Example attacks have been demonstrated with inconsistent TTL values [35], retransmitted TCP segments [62, 65, 66], among others. The key approach is to normalize traffic payload into a deterministic stream of bytes that is interpreted consistently. However, even deterministic payload streams can contain covert channels.

Network covert channels. Covert timing [20, 21, 32, 46, 49, 61, 67] and storage [11, 24, 33, 37, 41, 51, 59] channels have been a longstanding problem in the security community. Existing work has developed active wardens, which inspect network traffic, identify covert (timing or storage) channels, and modify the traffic to mitigate them [24, 27, 47]. However, most existing wardens are only proof-of-concept systems that are hard to deploy due to their inefficiency. To the best of our knowledge, NetWarden is the first practical defense against network covert channels. A different line of work [44, 71] has discovered the existence of covert channels due to the use of OpenFlow-based SDN controllers. They have also considered countermeasures against these covert channels. The scenario and threat model of NetWarden are closer to those of active wardens, which aim to detect and mitigate covert channels in network traffic originating from compromised hosts.

Programmable data planes. Programmable data planes have been used for a wide variety of *networking* tasks, such as network measurement [56, 70], monitoring [30, 63], and application offloading [25]. Only nascent work exists that leverages programmable data planes for network security [42, 53]. The closest to our work is a workshop paper [69], but it does not contain a full system design or evaluation.



(a) Bufferbloat attacks (b) Excessive IPD attacks (c) Connection table flooding attacks

Figure 11: The self defenses in NetWarden can successfully identify malicious traffic patterns and block them.

7 Conclusion

Network covert channels have been a longstanding threat to systems that host sensitive data. Existing defenses only work as proof-of-concept solutions, not only because they need to process every single packet in software, but also because of the performance drops due to channel mitigation. We have presented NetWarden, a system that can defend against network covert channels leveraging emerging switch hardware. It is the first system that can mitigate network covert channels in high-speed traffic while preserving performance. NetWarden achieves this by coupling defenses that degrade performance with new defenses that boost performance, neutralizing its overall performance impact. Our evaluation shows that NetWarden incurs low overheads, and that it can effectively mitigate covert timing and storage channels with minimum performance disturbance.

8 Acknowledgments

We thank the anonymous reviewers for their valuable feedback; we also thank Adam Morrison and Srinivas Narayana for their insightful comments on earlier drafts of this paper. This work was partially supported by NSF grants CNS-1942219 and CNS-1801884.

References

[1] Barefoot Tofino. `https://www.barefootnetworks.com/technology/#tofino`.

[2] Broadcom Trident 4 delivers disruptive economics for enterprise data center and campus networks. `https://www.globenewswire.com/news-release/2019/06/11/1866927/0/en/Broadcom-Trident-4-Delivers-Disruptive-Economics-for-Enterprise-Data-Center-and-Campus-Networks.html`.

[3] Common Criteria for IT security evaluation (ISO/IEC 15408). `https://csrc.nist.gov/glossary/term/Common-Criteria-for-IT-Security-Evaluation`.

[4] Information Technology Security Evaluation Criteria (ITSEC). `http://www.iwar.org.uk/comsec/resources/standards/itsec.htm`.

[5] Intel FlexPipe. `https://www.intel.com/content/www/us/en/products/network-io/ethernet/switches.html`.

[6] Netronome Agilio. `https://www.netronome.com/products/agilio-cx/`.

[7] The NetWarden code repository. `https://github.com/jiarong0907/NetWarden`.

[8] NTT service level agreement (SLA). `https://www.us.ntt.net/support/sla/network.cfm`.

[9] The P4 language repositories. `https://github.com/p4lang`.

[10] TCP SYN cookies. `https://etherealmind.com/tcp-syn-cookies-ddos-defence/`.

[11] C. Abad. IP checksum covert channels and selected hash collision. Technical report, iUniversity of California, Los Angeles, 2001.

[12] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav, and G. Varghese. Conga: Distributed congestion-aware load balancing for datacenters. In *Proc. SIGCOMM*, 2014.

[13] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data center TCP (DCTCP). In *Proc. SIGCOMM*, 2010.

[14] M. Allman, V. Paxson, and E. Blanton. TCP congestion control. RFC 5681, 2009.

[15] E. Anderson. Capture, conversion, and analysis of an intense NFS workload. In *Proc. FAST*, 2009.

[16] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proc. OSDI*, 2010.

[17] A. Belozubova, A. Epishkina, and K. Kogos. Random delays to limit timing covert channel. In *Proc. EISIC*, 2016.

[18] J. Border, M. Kojo, J. Griner, G. Montenegro, and Z. Shelby. Performance enhancing proxies intended to mitigate link-related degradations. RFC 3135, 2001.

[19] L. S. Brakmo, S. W. O'Malley, and L. L. Peterson. TCP Vegas: New techniques for congestion detection and avoidance. In *Proc. SIGCOMM*, 1994.

[20] S. Cabuk. *Network covert channels: Design, analysis, detection, and elimination*. PhD thesis, Purdue University, 2006.

[21] S. Cabuk, C. E. Brodley, and C. Shields. IP covert timing channels: Design and detection. In *Proc. CCS*, 2004.

[22] A. Chen, W. B. Moore, H. Xiao, A. Haeberlen, M. Sherr, C. Shields, and W. Zhou. Detecting covert timing channels with time-deterministic replay. In *Proc. OSDI*, 2014.

[23] G. Cormode and S. Muthukrishnan. An improved data stream summary: The count-min sketch and its applications. *J. Algorithms*, 55(1):58–75, Apr. 2005.

[24] D. M. Dakhane and P. R. Deshmukh. Active warden for TCP sequence number base covert channel. In *Proc. ICPC*, 2015.

[25] H. T. Dang, D. Sciascia, M. Canini, F. Pedone, and R. Soulé. NetPaxos: Consensus at network speed. In *Proc. SOSR*, 2015.

[26] Department of Defense. Trusted Computer System Evaluation Criteria (TCSEC). (DoD 5200.28-STD), 1985.

[27] G. Fisk, M. Fisk, C. Papadopoulos, and J. Neil. Eliminating steganography in Internet traffic with active wardens. In *Proc. IH*, 2002.

[28] S. Floyd, T. R. Henderson, and A. V. Gurtov. The NewReno modification to TCP's fast recovery algorithm. RFC 3782, 2004.

[29] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat. Exploiting a natural network effect for scalable, fine-grained clock synchronization. In *Proc. NSDI*, 2018.

[30] M. Ghasemi, T. Benson, and J. Rexford. Dapper: Data plane performance diagnosis of TCP. In *Proc. SOSR*, 2017.

[31] S. Gianvecchio and H. Wang. Detecting covert timing channels: An entropy-based approach. In *Proc. CCS*, 2007.

[32] S. Gianvecchio, H. Wang, D. Wijesekera, and S. Jajodia. Model-based covert timing channels: Automated modeling and evasion. In *Proc. RAID*, 2008.

[33] J. Giffin, R. Greenstadt, P. Litwack, and R. Tibbetts. Covert messaging through TCP timestamps. In *Proc. PET*, 2002.

[34] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. M. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can jump them! In *Proc. NSDI*, 2015.

[35] M. Handley, C. Kreibich, and V. Paxson. Network intrusion detection: Evasion, traffic normalization and end-to-end protocol semantics. In *Proc. USENIX Security*, 2001.

[36] K. He, E. Rozner, K. Agarwal, Y. Gu, W. Felter, J. Carter, and A. Akella. AC/DC TCP: Virtual congestion control enforcement for datacenter networks. In *Proc. SIGCOMM*, 2016.

[37] A. Hintz. Covert channels in TCP and IP headers. *Presentation at DEFCON*, 2002.

[38] K.-F. Hsu, R. Beckett, A. Chen, J. Rexford, P. Tammana, and D. Walker. Contra: A programmable system for performance-aware routing. In *Proc. NSDI*, 2020.

[39] S. Jero, E. Hoque, D. Choffnes, A. Mislove, and C. Nita-Rotaru. Automated attack discovery in TCP congestion control using a model-guided approach. In *Proc. NDSS*, 2018.

[40] C. Jin, D. X. Wei, and S. Low. FAST TCP: Motivation, architecture, algorithms, performance. *IEEE/ACM Trans. on Networking*, 14:1246–1259, 2006.

[41] E. Jones, O. Le Moigne, and J.-M. Robert. IP traceback solutions based on time to live covert channel. In *Proc. ICON*, 2004.

[42] Q. Kang, L. Xue, A. Morrison, Y. Tang, A. Chen, and X. Luo. Programmable in-network security for context-aware BYOD policies. In *Proc. USENIX Security*, 2020.

[43] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan. Generic external memory for switch data planes. In *Proc. HotNets*, 2018.

[44] R. Krösche, K. Thimmaraju, L. Schiff, and S. Schmid. I DPID it my way!: A covert timing channel in software-defined networks. In *Proc. Networking*, 2018.

[45] B. Lampson. A note on the confinement problem. *Communications of the ACM*, 16:613–615, 1973.

[46] K. S. Lee, H. Wang, and H. Weatherspoon. PHY covert channels: Can you see the idles? In *Proc. NSDI*, 2014.

[47] G. Lewandowski, N. B. Lucena, and S. J. Chapin. Analyzing network-aware active wardens in IPv6. In *Proc. IH*, 2006.

[48] Y. Li, R. Miao, C. Kim, and M. Yu. FlowRadar: A better netflow for data centers. In *Proc. NSDI*, 2016.

[49] X. Luo, E. W. W. Chan, and R. K. C. Chang. TCP covert timing channels: Design and detection. In *Proc. DSN*, 2008.

[50] X. Luo, E. W. W. Chan, and R. K. C. Chang. CLACK: A network covert channel based on partial acknowledgment encoding. In *Proc. ICC*, 2009.

[51] X. Luo, E. W. W. Chan, R. K. C. Chang, and W. Lee. A combinatorial approach to network covert communications with applications in web leaks. In *Proc. DSN*, 2011.

[52] G. R. Malan, D. Watson, F. Jahanian, and P. Howell. Transport and application protocol scrubbing. In *Proc. INFOCOM*, 2000.

[53] R. Meier, P. Tsankov, V. Lenders, L. Vanbever, and M. Vechev. NetHide: Secure and practical network topology obfuscation. In *Proc. USENIX Security*, 2018.

[54] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu. SilkRoad: Making stateful layer-4 load balancing fast and cheap using switching ASICs. In *Proc. SIGCOMM*, 2017.

[55] B. Montazeri, Y. Li, M. Alizadeh, and J. Ousterhout. Homa: A receiver-driven low-latency transport protocol using network priorities. In *Proc. SIGCOMM*, 2018.

[56] S. Narayana, A. Sivaraman, V. Nathan, P. Goyal, V. Arun, M. Alizadeh, V. Jeyakumar, and C. Kim. Language-directed hardware design for network performance monitoring. In *Proc. SIGCOMM*, 2017.

[57] P. Peng, P. Ning, and D. S. Reeves. On the secrecy of timing-based active watermarking trace-back techniques. In *Proc. SP*, 2006.

[58] I. Rhee, L. Xu, S. Ha, A. Zimmermann, L. Eggert, and R. Scheffenegger. CUBIC for fast long-distance networks. RFC 8312, 2018.

[59] C. H. Rowland. Covert channels in the TCP/IP protocol suite. *First Monday*, 2(5), 1997.

[60] A. Roy, H. Zeng, J. Bagga, G. Porter, and A. C. Snoeren. Inside the social network's (datacenter) network. In *Proc. SIGCOMM*, 2015.

[61] G. Shah, A. Molina, M. Blaze, et al. Keyboards and covert channels. In *Proc. USENIX Security*, 2006.

[62] U. Shankar and V. Paxson. Active mapping: Resisting NIDS evasion without altering traffic. In *Proc. SP*, 2003.

[63] J. Sonchack, O. Michel, A. J. Aviv, E. Keller, and J. M. Smith. Scaling hardware accelerated network monitoring to concurrent and dynamic queries with *flow. In *Proc. ATC*, 2018.

[64] E. Vanini, R. Pan, M. Alizadeh, P. Taheri, and T. Edsall. Let it flow: Resilient asymmetric load balancing with flowlet switching. In *Proc. NSDI*, 2017.

[65] G. Varghese, J. A. Fingerhut, and F. Bonomi. Detecting evasion attacks at high speeds without reassembly. In *Proc. SIGCOMM*, 2006.

[66] M. Vutukuru, H. Balakrishnan, and V. Paxson. Efficient and robust TCP stream normalization. In *Proc. SP*, 2008.

[67] X. Wang and D. S. Reeves. Robust correlation of encrypted attack traffic through stepping stones by manipulation of inter-packet delays. In *Proc. CCS*, 2003.

[68] W. Wu and B. Ford. Deterministically deterring timing attacks in deterland. In *Proc. TRIOS*, 2015.

[69] J. Xing, A. Morrison, and A. Chen. NetWarden: Mitigating network covert channels without performance loss. In *Proc. HotCloud*, 2019.

[70] N. Yaseen, J. Sonchack, and V. Liu. Synchronized network snapshots. In *Proc. SIGCOMM*, 2018.

[71] M. Zhang, G. Li, L. Xu, J. Bi, G. Gu, and J. Bai. Control plane reflection attacks in SDNs: New attacks and countermeasures. In *Proc. RAID*, 2017.