Pacer: Network Side-Channel Mitigation in the Cloud

Aastha Mehta, Mohamed Alzayat, Roberta De Viti, Björn B. Brandenburg, Peter Druschel, Deepak Garg

Max Planck Institute for Software Systems (MPI-SWS)

Abstract

An important concern for many Cloud customers is data confidentiality. Of particular concern are potential data leaks via side channels, which arise when mutually untrusted parties contend on resources such as CPUs, caches, and network elements. In this paper, we present a principled solution for mitigating side channels that arise from shared network elements in a public Cloud. Our solution, Pacer, shapes the outbound traffic of a Cloud tenant to make it independent of the tenant's secrets by design. At the same time, Pacer permits traffic variations based on public (non-secret) aspects of the tenants' computation, thus enabling efficient sharing of network resources. Implementing Pacer requires modest changes to the guest OS and the hosting hypervisor, and only minimal changes to guest applications. Pacer allows guests to protect their secrets with bandwidth overhead close to the minimum possible given a workload partitioning based on public information. For instance, Pacer can hide a requested static document from a real document corpus in one of two size clusters at an average throughput and bandwidth overhead of 6.8% and 150%, respectively.

1. Introduction

Clouds provide elastic computing, communication, and storage to customers large and small on a pay-per-use basis. An important requirement for many customers is the confidentiality of Cloud-hosted data and computation. Trusted execution environments (TEEs), such as those offered by Azure confidential computing [1], provide hardware-based isolation under a strong threat model. However, a TEE alone does not prevent leakage via *side channels*.

Side channels arise when mutually distrusting parties share hardware resources. Such sharing is in the very nature of public Clouds; for instance, CPUs, cores, caches and memory buses may be shared among otherwise isolated tenants co-located on the same server in an infrastructure-as-a-service (IaaS) Cloud. All of these shared resources have been exploited as side channels [76, 56, 53, 9, 55, 52, 77, 44, 34, 78, 5, 25, 65, 80, 7, 27, 62].

Even if tenants rent dedicated servers or CPU sockets and use memory within their local NUMA domain only, they still share network elements: the server's network interface card (NIC), a top-of-the-rack switch, a router, or a bottleneck link. By generating cross-traffic on such shared elements and observing its delay, an adversarial tenant can infer the shape of a co-located victim's encrypted network traffic [6, 60, 59]. Thus, in public Clouds the barrier is significantly lowered for mounting network side-channel attacks, which would otherwise be limited to adversarial network operators or other parties with direct access to a victim's network traffic.

Extensive prior work has shown that the shape of encrypted network traffic can reveal rich information about the underying communication, including what a user is typing, what webpage is being visited and which video is being streamed, what phrases are being used in VoIP conversations, and even the private keys of communication partners [63, 31, 69, 60, 73, 13, 12]. Moreover, Chen *et al.* [17] demonstrate that users' medical conditions, family income, and investment secrets can be gleaned from the encrypted traffic of healthcare, taxation, investment, and web search services provided as software-as-a-service offerings.

In contrast to side channels via shared microarchitectural state and memory, side channels via shared network elements have not received much attention, particularly in the context of Cloud computing. In this work, we present a system called *Pacer*, which focuses on mitigating network side channels in the Cloud. Specifically, Pacer prevents leaks of a tenant's secrets to an adversary who can observe, directly or indirectly, the shape of the tenant's encrypted network traffic.

Side channels via shared network elements can be avoided by reserving dedicated network bandwidth at the physical layer, but this approach is at odds with the idea of dynamic resource sharing in a public Cloud and difficult to realize even within a datacenter. Other defenses rely on noise in the adversary's observations, which may arise naturally due to unrelated cross traffic or can be added explicitly to weaken the correlation between a tenant's traffic shape and its secrets [37]. However, relying on noise alone is risky, because we cannot safely assume that the noise is sufficient in level and entropy at all times to defend against an adversary with unknown sophistication and resources.

A more principled approach is to *shape* the victim's traffic at the source in a manner that completely decorrelates the traffic shape from secrets, so that an adversary cannot infer the victim's secrets despite observing its traffic. Shaping involves padding all outgoing messages to a secret-independent size and transmitting all packets at secret-independent times. Unfortunately, a naïve use of this approach requires that each tenant transmit traffic at the peak rate of its workload, because any workload-dependent variation could reveal secrets.

Pacer reconciles security and efficiency by allowing the shape of the tenant's network traffic to depend on public (non-private) but not private information. Thus, Pacer protects secrets while reducing bandwidth and latency overhead compared to the naïve shaping approach. Moreover, Pacer relies on a paravirtualization approach to deliver strong side-channel mitigation while retaining the flexibility of an IaaS Cloud. In the following, we sketch key aspects of Pacer's design.

Security Whenever the tenant produces network output, a transmit schedule (shape) for the traffic is selected based on public information. The tenant's guest operating system (OS) prepares network packets as usual. The IaaS hypervisor shapes traffic as specified by the schedule, transmitting dummy packets when the guest fails to prepare payload packets in time. The resulting traffic shape is *secret-independent by design*.

Efficiency A tenant partitions its workload based on public information. For each partition, a profiler samples the distribution of the guest's unmodified payload traffic shapes and computes a schedule. All traffic within the partition is then shaped according to this schedule. Profiling affects performance but not security, because any traffic schedule chosen independent of secrets hides those secrets.

Usability Pacer places modest requirements on the tenant. The tenant and its clients must run a guest OS that supports Pacer. The tenant application requires minimal annotations to identify traffic boundaries and workload partitions. From the Cloud provider's perspective, the system requires modest changes to the IaaS hypervisor.

Novelty As discussed in §8, prior work that mitigates network side channels either does not fully decorrelate application secrets from traffic shape (packet sizes, number and timing), does not integrate with an IaaS Cloud, or is inherently inefficient on bursty workloads. To our knowledge, Pacer is the first working system that is efficient, prevents all network side-channel leaks by design, and overcomes the challenges of integrating network side channel mitigation into an IaaS Cloud while requiring minimal changes to guests.

Contributions This paper contributes (i) a novel *cloaked tunnel* abstraction, which ensures that the shape of network traffic in the tunnel is independent of secrets (§3); (ii) a paravirtualized cloaked tunnel implementation for an IaaS Cloud that ensures full isolation from potentially secret-dependent computations and requires modest changes to hypervisor and guests (§4); (iii) a *gray-box profiler* that generates transmit schedules automatically from tenant execution traces with minimal support from the tenant application (§5); and finally, (iv) an experimental evaluation on video streaming and static content hosting services, which shows that strong mitigation of network side channels is possible with modest overhead (§6). We discuss extensions of Pacer in §7, present related work in §8, and conclude in §9.

2. Pacer overview

We begin with a discussion of Pacer's threat model and security goals, and present its key ideas in designing a secure and efficient network side-channel mitigation.

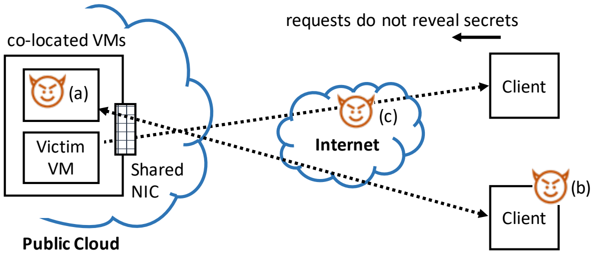

Figure 1: The adversary can (a) co-locate VMs with victim's VM in the Cloud, (b) control clients of its own VMs, (c) and use cross-traffic between any pair of these to infer the shape of the victim's traffic at shared network links.

2.1. Threat model

Pacer's goal is to prevent network side-channel leaks of a Cloud tenant's secrets to anyone able to rent other VMs in the Cloud. Prior work has shown that it is easy to attain co-location with a victim VM [59, 32, 33]. Accordingly, we assume a strong adversary that may co-locate its VMs with the victim's VM and indirectly infer the shape of the victim's outbound traffic by observing contention with its own cross-traffic. In particular, the adversary may use this method to infer the traffic shape of the victim at shared network elements in the common server, rack or datacenter. However, since our focus is only on side channels, we assume that the victim VM and its legitimate remote clients are uncompromised[1]. While not the goal of our work, Pacer's design also protects against powerful adversaries with the ability to directly observe the victim's traffic as well as delay, drop, and inject network packets.

In more detail, a tenant in a public IaaS Cloud, subsequently called the *victim*, executes in one or more guest VMs. The victim performs arbitrary computations but does not invoke other guest VMs or Cloud back-end services. The victim serves a set of trusted clients that connect to guest VMs from outside the Cloud via a secure virtual private network (VPN), or within the Cloud via a virtual Cloud network[2]. (Guests may require a second level of authentication to separate clients' privileges, but this is not relevant for Pacer's security.)

The victim's goal is to protect its secrets; these secrets can be reflected in parameters of client requests (i.e., secret inputs, such as the name of a requested file) or in the victim's internal state or code (e.g., which request handlers are cache-hot because they were recently accessed). The victim rents an entire server socket for exclusive use and uses main memory within the NUMA domain associated with this socket to mitigate micro-architectural side channel leaks to co-located tenants

[1] Our threat model and our solution are different from prior work on predictive mitigation for *covert-channel* leaks [8, 79], where remote clients may be adversarial and the victim itself may be compromised. We discuss this further in §8.

[2] In principle, Pacer can support multi-tier guests and also guests that provide an open service to untrusted clients, but these extensions are beyond the scope of this paper and not implemented in our prototype. §7 discusses the extension to multi-tier guests.

(micro-architectural side channels are not the focus of this work)[3]. The victim partitions its workload independent of secrets.

Our focus is on server-side security; protecting client privacy is a non-goal. Moreover, we assume that client request traffic reveals no secrets through its shape (its length, number of packets, or timing)[4]. This implies that the time of requests does not depend on any secrets or the actual completion times of previous responses. A victim can design the content being served to meet this requirement, e.g., in the case of a web server by in-lining embedded objects, and using Javascript to decorrelate (in time) requests for embedded links and user requests that may depend on previously served content.

The adversary seeks to learn the victim's secrets. It controls one or more guest VMs in the same public IaaS Cloud and has the ability to co-locate with the victim's VMs in the same server, rack, or data center. The adversary controls network clients, which may communicate freely with the adversary's VMs. The adversary cannot access the victim's VPN or impersonate/compromise the victim's clients.

The adversary has access to all services available to IaaS guests, including the ability to time the transmission and reception of its own network packets with high precision. Figure 1 summarizes Pacer's threat model.

2.2. Challenges and key ideas

A principled approach to avoiding network side channels is to shape the network traffic so that it cannot reveal secrets. If done naïvely, shaping can be very costly in terms of bandwidth or latency when the payload traffic is bursty. Pacer exploits the following key ideas to reconcile security and efficiency.

Per-guest dynamic shaping Pacer shapes each guest's network traffic dynamically according to its prevailing workload, thus enabling dynamic sharing of the available network capacity among different guests for efficiency. This requires that the presence and time of *requests* reveal no secrets, as assumed in our threat model.

Secret-independent shaping Instead of insisting on a uniform traffic shape for all of a guest's network traffic, Pacer allows the shape to vary, as long as the variations don't depend on secrets. For instance, if the type of content being requested from a server (e.g., document vs. video) is not a secret, then a different traffic shape can be used for the two. This additional degree of freedom helps minimize overhead for variable network traffic while preventing leaks of secrets.

Gray-box profiling Dynamic traffic shaping requires an understanding of how a guest's secrets affect its network traffic. This information can be obtained via program analysis, but that is difficult to perform on arbitrary binaries running in a VM. Black-box profiling can be performed on arbitrary guests, but cannot reliably discover all dependencies and therefore is not secure. Pacer instead relies on gray-box profiling, which requires no knowledge of a guest's internals beyond an explicit *traffic indicator* from the guest. This indicator partitions the guest's possible network interactions independent of secrets and indicates the onset of a particular interaction. It is used by Pacer in two ways: (i) to profile the guest's network interactions and generate a transmit schedule for each partition, and (ii) to instantiate a transmit schedule for a network interaction. As long as a guest computes the indicator independent of secrets, the choice of indicator may affect performance but not security.

Paravirtualized cloaked tunnel support Pacer provides paravirtualized hypervisor support that enables guests to implement a cloaked network tunnel, while adding only a modest amount of code to the hypervisor. A performance-isolated shaping component in the hypervisor initiates transmissions based on a schedule. If no payload is available at the time of a packet's scheduled transmission, the shaping component transmits a dummy packet instead. To the adversary, this dummy is indistinguishable from a payload packet since the traffic is encrypted.

3. Cloaked tunnel

Pacer's key abstraction is a *cloaked tunnel*, which ensures that the shape of network traffic inside the tunnel is secret-independent, thus defending against adversaries who can observe, directly or indirectly, traffic inside the tunnel. In this section, we describe the tunnel and its security independent of a specific application setting, implementation, or placement of tunnel entry and exits. In §4, we evolve the design to work within the constraints of a realistic Cloud environment.

3.1. Requirements

We begin with the requirements for a cloaked tunnel. *Secret-independent traffic shape:* Transmissions must follow a schedule that does not depend on secrets; actual transmission times must not be delayed by potentially secret-dependent computations. *Unobservable payload traffic:* The traffic shape must not reveal, directly or indirectly, an application's actual time and rate of payload generation and consumption. This implies that flow control must not affect the traffic shape; that padded content must elicit the same response (e.g., ACKs) from receivers as payload data; and that packet encryption must encompass the padding. This in turn requires that padding be added at or above the transport layer, while encryption be done below the transport layer. *Congestion control:* The tunnel must react to network congestion. Congestion control is needed for network stability and fairness, but does not reveal secrets since it reacts to network conditions, which themselves depend only

[3] In principle, this assumption can be relaxed by combining Pacer with complementary work to mitigate side-channel leaks through shared memory buses, caches, and micro-architectural CPU state [67, 11].

[4] This assumption can be avoided by shaping client traffic. In principle, Pacer can support this by running a hypervisor on the client side, as discussed in §7.

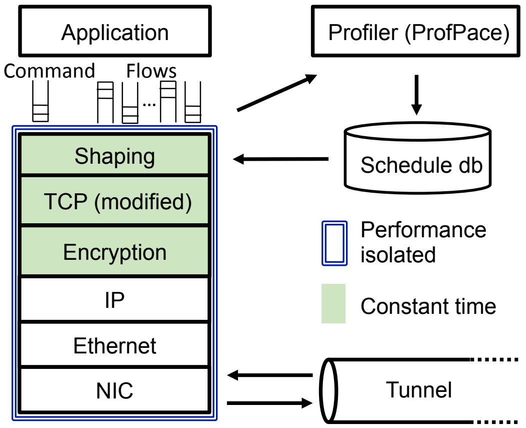

Figure 2: Cloaked tunnel (one endpoint)

on shaped and third-party traffic.

3.2. Architecture

Figure 2 shows the cloaked tunnel's architecture. The tunnel protocol stack runs on both tunnel endpoints. (Only one of two symmetric endpoints is shown in the figure.) The tunnel protocol stack consists of a *shaping* layer on top of a modified transport layer (e.g., TCP) on top of the encryption layer. These three layers rest on top of conventional IP and link layers. Each tunnel is associated with a flow identified by a 5-tuple of source and destination IP addresses and ports, and the transport protocol.[5]

The shaping layer initiates transmissions according to a schedule and pads packets to a uniform size. It interacts with applications via a set of shared, lock-free queues. The layer takes application data from a per-flow outbound queue and transmits it in the tunnel. It places incoming data from the tunnel into a per-flow inbound queue. Finally, it receives traffic indicators and per-flow cryptographic keys (to be used by the encryption layer) via a per-application command queue.

A separate, user-level *gray-box profiler* (ProfPace) analyzes timestamps and traffic indicators collected by the tunnel, and generates and updates transmit schedules in the schedule database. ProfPace is described in §5.

Assumptions The tunnel design presented in this section relies on some idealized assumptions, which will be relaxed in the practical design of §4. Specifically, we assume here that the processing delays in the tunnel network stack are not influenced by secrets directly or indirectly. This requires that: (i) the tunnel's layers—especially the shaping, transport, and encryption layers, which operate on cleartext data—execute in *constant time*, *i.e.*, they avoid data-dependent control flow and memory access patterns in their data path implementation; and (ii) the execution of the tunnel network stack is *performance-isolated* from the application and any other computation outside the stack.

Transmit schedules A transmit schedule is a finite series of times at which packets within a flow are transmitted. A schedule is typically associated with a type of packet train, e.g., a file transfer or the response to a service request. There is at most one active schedule on a flow at a time; successive schedules on the same flow are non-overlapping in time.

Outbound data processing A timestamp is taken whenever data is queued by the application; these timestamps and the recorded traffic indicators are shared with the gray-box profiler. The shaping layer retrieves a chunk of available data from the flow's outbound queue whenever a transmission is due on a flow according to the active schedule (if any) and TCP's congestion window is open (see transport layer below). The layer removes a number of bytes that is the minimum of (i) the available bytes in the queue, (ii) the receiver's flow control window (see transport layer below), and (iii) M, the network's maximal transfer unit (MTU) minus the size of all headers in the stack. If fewer than M bytes (possibly zero) were retrieved from the queue due to payload unavailability or flow control, the shaping layer pads the chunk to M bytes. It adds a header that indicates the number of padding bytes added.

Transport layer The transport layer operates as normal, except for two tunnel-related modifications: 1) When the congestion window closes, the transport layer signals the shaping layer to suspend the flow's transmit schedule until the congestion window reopens. Schedule suspension ensures network stability and TCP-friendliness, and does not leak information because it depends only on network conditions, which are visible to the adversary anyway. 2) Flow control is modified to make it unobservable to the adversary. The transport layer signals to the shaping layer the size of the flow control window advertised by the receiver. This window controls how much payload data is included in packets generated by the shaping layer (see above). The transport layer transmits packets irrespective of the flow control window, sending dummy packets while the window is closed, which are discarded at the other end of the tunnel.

The transport layer passes outbound packets to the encryption layer, which adds a message authentication code (MAC) keyed with the flow's key to a header and encrypts the packet with the flow's key. Finally, encrypted packets are passed to the IP layer, where they are processed as normal down the remaining stack and transmitted by the NIC.

Inbound packet processing Packets arriving from the tunnel are timestamped; the stamps are shared with the profiler. Packets pass through the layers in reverse order, causing TCP to potentially send ACKs. The encryption layer decrypts and discards packets with an incorrect MAC. The shaping layer strips padding and places the remaining payload bytes (if any) into the inbound queue shared with the application.

Schedule installation A transmit schedule must be installed on a flow before data can be sent via the tunnel. A guest application does so indirectly by sending traffic indicators.

[5] We describe the tunnel in terms of TCP; however, another stack like QUIC [38] can also be used.

The application provides the flow's 5-tuple f, a traffic id *sid* (which maps to a transmit schedule), and a type. The shaping layer looks up the schedule associated with *sid* in the schedule database and associates it with flow f.

There are two types of schedules: *default* and *custom*. A default schedule is installed when the flow is created. This schedule acts as a template, which is instantiated automatically by the shaping layer whenever an incoming packet arrives that indicates the start of a new network exchange (e.g., a GET request on a persistent HTTP connection), identified by the TCP PSH flag. The schedule starts at a time equal to the arrival time of the packet that causes the schedule's instantiation.

A default schedule active on a flow can be extended by a custom schedule in response to an application traffic indicator. For instance, a default schedule that allows a TLS handshake might be extended with one that is appropriate for the response to the first incoming network request. The new schedule can extend the currently active schedule only if the new schedule's prefix matches the prefix of the currently active schedule that has already been played out. Because the new schedule is anchored at the same time as the profile it extends, the time of a schedule extension is unobservable to the adversary. A traffic indicator that would require a custom profile that does not match the played-out prefix of the active schedule is ignored.

3.3. Tunnel security

Next, we justify the cloaked tunnel's security, summarized by property **S0:** *The shape of traffic in the tunnel does not depend on secrets.* Follows from S1–S7.

S1: *Transmit schedules are chosen based on public information.* By assumption about applications' choice of schedules.

S2: *The traffic in the tunnel is independent of the payload traffic.* Holds because packets are (i) padded and transmitted independently of the application's rate of payload generation and consumption; (ii) elicit a transport-layer response from the receiver independently of payload traffic; and (iii) the packet contents including headers that reveal padding are encrypted.

S3: *All packet transmissions follow a schedule.* Holds because shaping initiates transmissions according to a schedule.

S4: *Delays between scheduled and actual packet transmission times do not reflect secrets.* Follows from the fact that the tunnel stack, from the shaping layer down, is performance-isolated from any secret-dependent computation and layers that operate on cleartext are constant-time.

S5: *Transmit schedules are activated, paused, and re-activated at a secret-independent delay from any observable event that causally precedes the pausing or (re-)activation.* Holds because (i) pausing, reactivation, and instantiation of default schedules is performed within the performance-isolated tunnel stack; and (ii), by assumption, custom schedule installations that take immediate effect are not causally preceded by an observable event.

S6: *Transmit schedules are suspended and resumed only according to the network's congestion state.* Follows from the tunnel's transport layer congestion control mechanism.

S7: *Modifications of active transmit schedules do not reveal secrets.* Holds because the time of schedule replacement is unobservable to the adversary (matching prefix).

4. Pacer design

In the previous section, we described the conceptual design of a cloaked tunnel. In this section, we describe Pacer, a concrete and practical cloaked tunnel design in the context of a public IaaS Cloud.

We begin with a discussion of constraints on the design space in the context of a IaaS Cloud. First, *the tunnel entry must be integrated with the IaaS server*. In an IaaS Cloud, co-located tenants typically share the network link attached to the server and can therefore indirectly observe each others' traffic. Therefore, the tunnel entry must be in the IaaS server to ensure the attached link lies inside the tunnel. Secondly, *shaping requires padding, which must be done at the transport layer to ensure it is unobservable*. Thirdly, the conceptual tunnel design from the previous section requires that the *network stack is performance-isolated from secret-dependent computations and layers that deal with cleartext are constant-time*. All guest computation must be assumed to be secret-dependent in an IaaS server, suggesting that shaping should be implemented in the IaaS hypervisor, where it can be executed with dedicated resources and tightly controlled.

One way to meet these requirements would be to place the entire network stack into the hypervisor, performance-isolate it from guests, and implement it as constant time. This approach, however, has significant limitations. First, ensuring performance isolation for an entire network stack is technically challenging even in the hypervisor. Second, implementing the tunnel layers as constant time is not trivial. Third, the approach defeats NIC virtualization and instead requires that guests and their network peers use the network stack provided by the IaaS platform. Finally, it adds significant complexity to the hypervisor.

Pacer architecture Pacer addresses the tension outlined above using a paravirtualization approach. In Pacer, the hypervisor cooperates with the guest OS to implement the cloaked tunnel. The responsibilities are divided in such a way that *(i) the hypervisor can ensure tunnel security with only weak assumptions about a guest's rate of progress; (ii) the performance-isolated hypervisor component is small; (iii) required changes to the guest OS are modest*. Effectively, we extend the IaaS hypervisor to provide a *small set of functions that allows guests to implement a cloaked tunnel*, while guests retain the flexibility to use custom network stacks on top of a virtualized NIC.

Figure 3 shows Pacer's architecture. Unlike the strictly layered tunnel stack from §3, Pacer factors out a small set of functions that inherently require performance-isolation into the lowest layer, implemented in the IaaS hypervisor. The *HyPace*

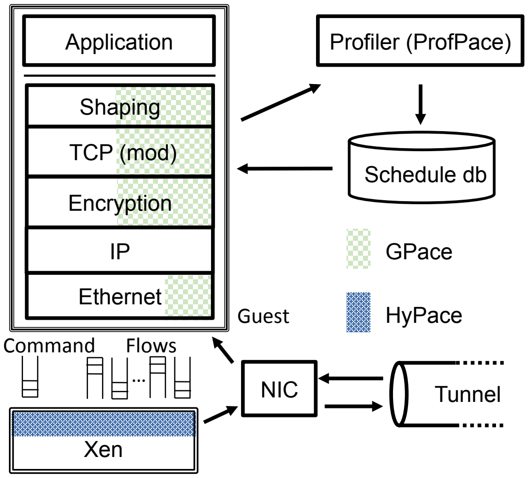

Figure 3: Pacer architecture

component plugs into Xen and provides these functions. The *GPace* component, a Linux kernel module, plugs into the guest OS and the OS of any network clients that interact with the guest. It implements the cloaked tunnel in cooperation with HyPace.

HyPace instantiates transmit schedules, encrypts and MACs packets, and initiates their transmissions, while masking potentially secret-dependent delays in its execution. It can generate padded (dummy) packets subject to congestion control independently from the guest network stack, thereby avoiding the need to performance-isolate the guest. GPace pads payload packets, and exposes each flow's congestion window, sequence number, and crypto key to HyPace. The guest has direct access to a virtual NIC (vNIC) configured by the hypervisor, which it uses to receive but not to transmit packets. Pacer's security properties remain equivalent to those of the conceptual cloaked tunnel design (§3.3).

4.1. HyPace

Similar to the shaping layer in the conceptual tunnel design, HyPace receives traffic indicators from applications (via GPace), instantiates template schedules in response to incoming packets (signaled by GPace), and initiates transmissions. To ensure tunnel security despite potentially secret-dependent delays in the guest, however, HyPace performs additional functions and there are differences, which we discuss next.

HyPace implements padding, encryption, and congestion control in cooperation with the guest. HyPace pauses a transmit schedule when a flow's congestion window closes and resumes the schedule when it reopens. When a transmission is due on a flow and the congestion window is open, HyPace checks whether the guest has queued a payload packet. If not, it generates a dummy packet with proper padding, transport header, and encryption, using the next available TCP sequence number and the flow key shared with the guest. Finally, it initiates the transmission of the payload or dummy packet and reduces the congestion window accordingly.

Interface with guests HyPace shares a memory region pairwise with each guest. This region contains a data structure for each active flow. The flow structure contains the following information: the connection 5-tuple associated with the flow; a sequence of transmission schedule objects; the current TCP sequence number *seq* and the right edge of the congestion window *cw*; the flow's encryption key; and, a queue of packets prepared for transmission by the guest. Each transmit schedule object contains the *sid* and a starting timestamp. HyPace and the guest use lock-free synchronization on data they share.

Packet transmission HyPace transmits packets according to the active schedule in the packet's flow. From a security standpoint, packets need not be transmitted at the exact scheduled times; however, any deviation between scheduled and actual time must not reveal secrets.

On general-purpose server hardware, it is challenging to initiate packet transmissions such that their timing cannot be influenced by concurrent, secret-dependent computations. Using hardware timers, events can be scheduled with cycle accuracy. However, the activation time and execution time of a software event handler is influenced by a myriad of factors. These may include (i) disabled interrupts at the time of the scheduled event; (ii) the CPU's microarchitectural, cache, and write buffer state at the time of the event; (iii) concurrent bus traffic; (iv) frequency and voltage scaling; and (v) non-maskable interrupts during the handler execution. Many of these factors are influenced by the state of concurrent executions on the IaaS server and may therefore carry a timing signal about secrets in those executions.

Masking event handler execution time HyPace masks hardware state-dependent delays to make sure they do not affect the actual time of transmissions. A general approach is as follows. First, we determine empirically the distribution of delays between the scheduled time of a transmission and the time when HyPace's event handler writes to the NIC's *doorbell register*, which initiates the transmission. We measure this distribution under diverse concurrent workloads to get a good estimate of its true maximum. We relax this estimate further to account for the possibility that we may not have observed the true maximum and call this resulting delay δ_{xmit}. Second, for a transmission scheduled at time t_n, we schedule a timer event at $t_n - \delta_{xmit}$. Third, when the event handler is ready to write to the NIC doorbell register, it spins in a tight loop reading the CPU's clock cycle register until t_n is reached and then performs the write. By spinning until t_n, HyPace masks the event handler's actual execution time, which could be affected by secrets.

Unfortunately, the measured distribution of event handler delays has a long tail. We observed that the median and maximum delay can differ by three orders of magnitude (tens of nanoseconds to tens of microseconds). This presents a problem: With the simple masking approach, a single core could at most initiate one transmission every δ_{xmit} seconds, making it infeasible to achieve the line rate of even a 10Gbps link. Instead, we rely on *batched transmissions*.

Batched transmissions The solution is based on two insights. First, instances in the tail of the event handler delay distribution tend to be independent. As a result, the maximal delay for transmitting *n* packets in a single event handler activation does not increase much with *n*. Therefore, we can amortize the overhead of masking handler delays over *n* packets. Second, actual transmission times can be delayed as long as the delay does not depend on secrets. Therefore, it is safe to batch transmissions.

We divide time into *epochs*, such that all packet transmissions from an IaaS server scheduled in the same epoch, across all guests and flows, are transmitted at the end of that epoch. An event handler is scheduled once per epoch, it prepares all packets scheduled in the epoch, spins until the batch transmission time, and then initiates the transmission with a single write to the NIC's doorbell register.

Let us consider factors that could delay the actual packet transmission time once the spinning core issues the doorbell write. Reads were executed before the spin, so the state of caches plays no role. The write buffer should be empty after the spin. Interference from concurrent NIC DMA transfers reflects shaped traffic and is therefore secret-independent. Similarly, any delays in the NIC itself due to concurrent outbound or inbound traffic cannot depend on secrets. However, the doorbell write itself could be delayed by traffic on the memory bus, PCIe bus, or bus controller/switch.

Hardware interference and NIC support A remaining source of delays are concurrent bus transactions caused by potentially secret-dependent computations. We tried to detect such delays empirically and have not been able to find clear evidence of them. Nonetheless, such delays cannot be ruled out on general-purpose hardware. A principled way to rule out such interference would require hardware support.

For instance, a *scheduled packet transmission* function provided by the NIC would be sufficient. Software would queue packets for transmission with a future transmission time t. At time $t - \delta_{bus}$, the NIC DMAs packets into onboard staging buffers in the NIC. Here, δ_{bus} would be chosen to be larger than the maximal possible delay due to bus contention. At time t, the NIC would initiate the transmission automatically. With such NIC support, HyPace would prepare packets for transmission as usual, but instead of spinning until t_n it would immediately queue packets with $t = t_n$. Incidentally, NIC support for timed transmissions is also relevant for traffic management, and a similar "transmit on time stamp" feature is already available on modern smart NICs [2]. We plan to investigate NIC support in future work.

In summary, HyPace is a small component implemented in the hypervisor, which is performance-isolated from the guest and enables guests to implement a cloaked tunnel. HyPace's careful design masks any potentially secret-dependent delays in the transmission of packets, obviating the need for a constant-time implementation of any part of the tunnel's network stack or a performance-isolated guest network stack.

4.2. GPace

GPace is a Linux kernel module that implements a cloaked tunnel jointly with HyPace. On the client-side of a network connection, GPace extends the kernel to terminate the tunnel. GPace pads outgoing TCP segments to MTU size and removes the padding on the receive path. It modifies Linux's TCP implementation to share its per-flow congestion window and sequence number with HyPace, and to notify HyPace of retransmissions so that HyPace can extend the active schedule by one transmission. Unlike in the generic tunnel, where the shaping occurs above the transport layer, this schedule extension is necessary to allow for retransmission; it does not leak information because it depends only on network state.

Note that TCP's flow control window is not advertised to HyPace, causing HyPace to send dummies if the receiver's flow control window is closed, as required. GPace timestamps outbound data arriving from applications and inbound packets from the tunnel in the vNIC interrupt handler. All timestamps and recorded traffic indicators are consumed by the profiler (§5).

GPace allows applications to install session keys and provide traffic indicators on flows via IOCTL calls on network sockets. Recall that applications specify a flow, a traffic ID *sid* and a type as arguments when indicating traffic. GPace passes this information into the per-flow queue shared with HyPace, which uses the *sid* as an index to look up the corresponding transmit schedule in the database.

Packet processing With GPace, the guest OS generates TCP segments as usual, but pads them to the MTU size before passing them to the IP layer. Instead of queuing packets in the vNIC's transmit queue, GPace queues them in per-flow transmit queues shared with HyPace. The guest OS processes incoming packets as usual by accepting interrupts and retrieving packets directly from its vNIC.

Schedule (re-)activation delays Unlike the conceptual tunnel design, Pacer processes inbound network packets in the guest, which is not performance-isolated. Therefore, care must be taken to ensure that the time of activation or re-activation of a transmit schedule in response to an inbound packet does not reveal the guest kernel's execution time, which could depend on secrets. Schedule (re-)activation must occur at a defined, secret-independent delay from the event that causally precedes it, e.g., a packet arrival.

Let ε be HyPace's epoch length and δ_{recv} be the guest OS's empirical maximal inbound packet processing time. There are four such events to consider: 1) *The arrival of the first packet of a request.* GPace instantiates a default schedule with a start time equal to the packet's arrival time. To make sure the first transmission occurs in time, we require that the initial response time of any default schedule be larger than $\varepsilon + \delta_{recv}$. 2) *The arrival of an ACK that opens the congestion window.* GPace ensures the ACK does not enable a transmission that is scheduled within $\varepsilon + \delta_{recv}$ of the ACK's arrival. 3) *The arrival of an ACK that causes a retransmission.* GPace ensures the ACK does not enable a transmission that is scheduled less that $\varepsilon + \delta_{recv}$ from the ACK's arrival. 4) *A timeout that causes a retransmission.* GPace ensures the timeout does not enable a transmission that is scheduled within $\varepsilon + \delta_{recv}$ of the timeout. Here, we use δ_{recv} as a conservative upper bound on the delay of the timeout event handler.

These four rules make the guest's actual processing time for incoming packets and timeouts unobservable to the adversary.

4.3. Pacer security

We justify Pacer's overall security. Pacer's threat model rules out side channels via shared CPU state, caches, and memory bandwidth, as well as shared Cloud back-end services. Therefore, the adversary is limited to (i) trying to connect to the victim as a client and observe the timing and content of responses, or (ii) measuring the shape of the victim's traffic by observing packet delays on a shared network link.

Attack (i) is not possible because the adversary cannot elicit a response from the victim. Pacer relies on encryption and a MAC keyed with pre-shared keys and GPace silently ignores incoming packets that cannot be authenticated. Attack (ii) is unproductive, because the victim's incoming traffic shape is secret-independent by assumption and its outgoing traffic is shaped to be secret-independent. Next, we justify that the victim's outgoing traffic shape is indeed secret-independent *by design*. In other words, Pacer's tunnel has property S0 of the cloaked tunnel from §3.

S1 and S3 hold trivially, because the relevant behavior of Pacer is equivalent to the conceptual tunnel's. S2 holds because Pacer, like the conceptual tunnel, pads packets above the transport layer, encrypts packets below the padding layer, and makes flow control unobservable. S4 follows from GPace's rules on the pausing and (re-)activation of transmission schedules. S5 holds because HyPace's batch transmission mechanism masks the execution time of its transmission event handler. S6 holds because HyPace cooperates with GPace to pause and resume schedules in response to the network's congestion state. Even though a schedule can be extended in Pacer, S7 still holds because schedule extension happens only in response to a packet loss, which is a public event.

5. Generating schedules

By default, Pacer can use the same transmit schedule for all of a guest's network traffic. This approach does not require any guest support and is perfectly secure. In practice, however, tenants can significantly reduce bandwidth and latency overhead by using different schedules for different partitions of their workload. As long as those partitions are chosen by public information, no information is leaked.

In this section, we discuss ProfPace, Pacer's gray-box profiler that profiles guests and generates transmit schedules automatically by analyzing the guest's recorded network interactions and the explicit traffic indicators provided by guest application. For content-serving guest applications, we also analyze the application's content corpus to suggest a clustering that maximizes efficiency given the application's privacy needs.

5.1. Gray-box profiling

ProfPace analyzes the arrival times of incoming packets, the times at which the guest OS queues packets for transmission, and the traffic indicators from guest applications. The guest's traffic indicators serve three purposes. First, they delimit segments of semantically related sequences of inbound and outbound packets within a network flow. Second, the *sid* value indicated by the guest identifies segments of the same equivalence class, e.g., a TLS handshake, or a response to a request within a given workload partition. Recall that the application determines the *sid* based on public information; the *sid* therefore partitions the guest's network interactions by public information. Third, the indicators indicate the onset of a network interaction and therefore provide an opportunity to install the appropriate schedule.

At each site in the guest's application where a message is sent to the network, an IOCTL call is invoked that provides a traffic indicator *sid*. GPace logs the traffic indicators along with the arrival times of incoming packets and the times at which the guest OS queues packets for transmission, and shares the logs with ProfPace.

ProfPace bins the recorded network interaction segments by *sid*. The set of observed segments in a bin are considered samples of the associated equivalence class of network interactions. ProfPace characterizes each class by a set of random variables. Empirically, we have determined the following variables sufficient: The delay between the first incoming packet and the first response packet d_i; the time between subsequent response packets d_s; and, the number of response packets p. For each class, the profiler samples the distribution of these random variables from the segments in the associated bin.

Finally, ProfPace generates a transmit schedule for *sid* based on the sampled distributions of the random variables. Specifically, it generates a schedule with the 100th %-ile of the number of packets p, the 99th %-ile of the initial delay d_i, and the 90th %-ile of the spacing among subsequent packets d_s. We have determined empirically that this works well.

Recall that as long as applications choose *sid* values based on public information, transmit schedules are relevant only for performance not security. An inadequate schedule could increase delays and waste network bandwidth due to extra padding, but cannot leak secrets. For good performance, during profiling runs, the guests should sample the space of workloads with different values of the public and private information, as well as different guest load levels, so that the resulting profiles capture the space of network traffic shapes well. Next, we consider how a content serving guest can partition its workload to minimize overhead given its privacy needs.

5.2. Corpus analysis

Pacer minimizes overhead by allowing the shape of network traffic to reveal information deemed public by the guest. Towards this end, a guest may partition its workload so as to minimize padding overhead while ensuring that no secret information is revealed. For instance, consider a guest that serves a corpus of objects with a skewed size distribution. Using a single schedule for the entire corpus requires padding every requested object to the largest object in the corpus, incurring a large overhead. Suppose now that the guest can partition the corpus based on public information such that each partition contains objects of similar size; now, each object is padded to the largest object in *its* cluster, which may reduce overhead significantly without revealing which object within a partition is being requested. Below we briefly describe a clustering algorithm for videos and static HTML documents that can minimize overhead subject to a guest's privacy needs.

However, we note that determining what information can be considered public and private in the context of a specific application and the corpus's size and popularity distributions may be challenging in general and beyond the scope of this paper. Our goal here is to highlight the large efficiency gains possible when clustering content with skewed size distributions. Specifically, we presents overhead results when clustering real videos and document corpuses in §6.1.

Video clustering We cluster videos according to the sequence and size of their 5-second segments using the following algorithm. Note that the dynamically compressed segments of a video differ in size. Initially, we over-approximate the shape of each video v_i by its maximal segment size $smax_i$ and its number of segments l_i. For each distinct video length l and each distinct maximal segment size s in the entire dataset, we compute the set of videos that are dominated by $\langle l, s \rangle$. A video v_i is dominated by $\langle l, s \rangle$ if $l_i \leq l$ and $smax_i \leq s$.

Let c be a desired minimum cluster size. Our algorithm works in rounds. In each round, we select every $\langle l, s \rangle$ dominating at least c videos, and we choose as cluster the set of videos minimizing the average padding overhead per video, i.e. $\frac{1}{c_i} \sum_{j=1}^{c_i} \sum_{k=1}^{l_i} \max_{1 \leq j \leq c_i}((s_k) - s_{kj})$, where c_i is the cardinality of the set of videos, l_i is the maximal length across all videos in the set and s_k is the maximal size of the k-th segment across all videos in the set. The sequence of segment sizes $\langle s_1, s_2, ..., s_{l_i} \rangle$ is the ceiling of the cluster c_i. Once a cluster is formed, videos in it are not taken into account in later rounds. The algorithm terminates when all videos are clustered. If the last cluster has less than c videos, it is merged with the one formed before it.

Document clustering We use a similar but simpler clustering algorithm to cluster static HTML documents. In contrast to videos, HTML documents contain only one data object. Therefore the clustering problem simplifies to one-dimensional clustering based on the single size parameter of individual documents, and the largest document in a cluster constitutes the cluster's ceiling.

6. Evaluation

In this section, we describe our implementation, and present results of an empirical evaluation of our Pacer prototype.

We implemented HyPace for Xen and GPace's Linux kernel module in 8,100 and ~15K lines of C, respectively. We imported 4,458 lines of AESNI assembly code from OpenSSL to encrypt packets in HyPace. We implemented ProfPace in 1,800 lines of Python and 1,200 lines of C. At each site in an application's code where a message is sent to the network, we add 15 LoC to send a traffic indicator via IOCTL to the guest kernel. We identified and modified these sites manually; automating the instrumentation is possible but remains future work. No other changes were required to guest applications.

All experiments were performed on Dell PowerEdge R730 server machines with Intel Xeon E5-2667, 3.2 GHz, 16 core CPU (two sockets, 8 cores per socket), 512 GB RAM, and a Broadcom BCM 57800 10Gbps Ethernet card. The NIC was configured to export virtual NICs (vNICs). We disabled hyperthreading, dynamic voltage and frequency scaling, and power management in the hosts. We used Xen hypervisor 4.10.0 on the hosts, and the 'Null' scheduler [4] for VM scheduling.

We assigned 40GB RAM and one of the CPU sockets to Xen; up to two cores are configured to execute the HyPace transmit event handler in parallel; flows are partitioned among the cores. The guest runs in a VM with 8 cores and 64 GB RAM, and has access to a vNIC. The VCPUs of the guest VM were pinned one-to-one to cores on the second socket of the host CPU. This is in line with our threat model, which assumes that guests rent dedicated CPU sockets. Pacer requires less than 10MB of additional main memory in the Xen hypervisor and less than 20MB of additional memory in each guest that uses Pacer. The guest runs an Ubuntu 16.04 LTS kernel (version 4.9.5, x86-64), and Apache HTTP Server 2.4. Network clients run Ubuntu 16.04 LTS without a hypervisor.

We evaluated Pacer's impact on client latencies and server throughput in the context of two guest applications: (i) a video streaming service, and (ii) a web service serving static documents. For video streaming, we wrote a custom video streaming server in PHP, which works like a simple file server serving video segments in response to client requests. The service serves videos from an ext4 file system on a VM disk. The document service is based on the Mediawiki server, version 1.27.1. Documents are stored in a database hosted locally on MySQL 5.7.16. We use a modified wrk2 [3] client to issue HTTPS GET requests for various pages to the document server.

With both services, we use real-world datasets, so that the videos or documents have real size distributions. For videos, we use a corpus of videos downloaded from Youtube in March

2018. For the document service, we use two different real data sets: static HTML pages of the English Wiktionary and static HTML pages of the English Wikipedia. Note that even though Youtube videos, Wiktionary pages and Wikipedia pages are not sensitive and may not need protection with a system like Pacer in practice, all that matters for our evaluation are the file sizes and size distributions. The content of the documents is completely irrelevant since the content is encrypted during transmission anyhow.

6.1. Spatial padding overhead

First, we measure the spatial padding overhead when clustering content as described in §5.2. This overhead corresponds to the network bandwidth overhead for Pacer's traffic shaping. We clustered three different datasets: (i) a set of videos downloaded from YouTube (1218 videos, max 468.7MB, median 6.2MB), (ii) a 2016 snapshot of the English Wiktionary corpus (5,027,344 documents, max 521.9kB, median 4.7kB), and (iii) a 2008 snapshot of the English Wikipedia corpus (14,257,494 documents, max 14.3MB, median 83.5kB). Figure 4 shows the reduction in the average and maximum padding overhead with increasing number of clusters and decreasing minimum cluster size (i.e, the minimum number of objects in each cluster). Compared to the two document datasets, the overhead reduction is less for videos due to the smaller dataset with a narrower range of video sizes, and due to the multi-dimensional clustering necessary for videos. Nonetheless, even clustering the corpuses into just two clusters leads to at least two orders of magnitude reduction in the average padding overhead. Moreover, by padding content to the cluster ceiling, Pacer trivially achieves the minimal bandwidth overhead possible given a workload partitioning.

6.2. Microbenchmarks

We empirically select a suitable HyPace epoch length ε, the maximum batch size B (number of packets to be prepared by a HyPace handler) in each epoch, and the parameters δ_{xmit} and δ_{recv} from §4. To this end, we ran multiple, 12-hour experiments with varying network workloads. We requested 100KB-sized documents from the document server using concurrent clients. As background workload, we ran large matrix multiplications on dom0 on the server. The workloads were configured to drive CPUs to near 100% utilization, and had a total working set of ~12GB of RAM.

To determine δ_{xmit}, ε and B, we measured the cost of preparing batches of packets for transmission in HyPace. Over many observations in the presence of the background load described above, we first determined the number of packets that can be safely prepared with different epoch lengths with a single HyPace handler. Within epochs of length 30μs, 50μs, 100μs and 120μs, the number of packets was 5, 14, 33, and 42, respectively, which allows HyPace to achieve 22%, 28%, 41% and 42% of the NIC line rate with a single core. We configured ε to be 120μs for all HyPace handlers.

Based on these results, we use two parallel HyPace handlers running on two separate cores. In this configuration, we repeated our measurements and chose B = 38 packets and δ_{xmit} = 35μs for each handler. δ_{recv} is independent of the number of HyPace threads, and its average and maximum values observed across all experiment configurations were 3.9ms and 15.8ms, respectively. We conservatively configured δ_{recv} to 20ms.

6.3. Video service

Next, we measure the impact of Pacer's traffic shaping on the video streaming service. We wrote a Python streaming client that simulates a MPEG-DASH player: when a user requests a video, the client initially fetches six segments (covering 5 seconds of video each) in succession to fill up a local buffer. After reaching 50% of the initial buffer (rebuffering goal), the player starts consuming the segments from the buffer. The client fetches subsequent segments sequentially whenever space is available in the buffer. With Pacer, the player does not request a segment until the transfer of the previous segment has finished, including any padding; otherwise, the timing of the client's request would reveal the actual size of the previous segment. We measure the impact of traffic shaping on: (1) the initial delay until the video starts playing; (2) the frequency and duration of any pauses (video skipping) experienced by the player; (3) the download latency for individual video segments. The client sequentially plays four videos, randomly chosen among 1218 videos, for up to 5 minutes each. The videos were clustered into 19 clusters with at least 61 elements each. We ran experiments for a client with high bandwidth (10Gbps) and with low bandwidth (10Mbps).

Transmit schedules generated by ProfPace seek to download each segment as fast as the available bandwidth allows, which is how the baseline system works. With these schedules, there is no noticeable impact on the user experience for using Pacer. Initial startup delays don't increase significantly, and there is no video skipping in any of the experiments. When serving 128 clients, the maximum CPU utilization on the server increases from 3.73% to 6.26% with Pacer.

It is interesting that Pacer's shaping also provides an opportunity to use domain knowledge to optimize schedules for better *performance*. We know that downloading the largest segment in our collection of 240p videos within its 5 second deadline requires 550kbps. Therefore, we can conservatively modify the inter-packet spacing in the schedules to 6ms, corresponding to a bandwidth of about 2Mbps. For clients with 10 Mbps bandwidth this optimization avoids losses and reduces the segment download latency significantly, but we omit the full results due to space constraints. This schedule optimization does not affect security; it only takes advantage of Pacer to reduce network contention, a known benefit of traffic shaping.

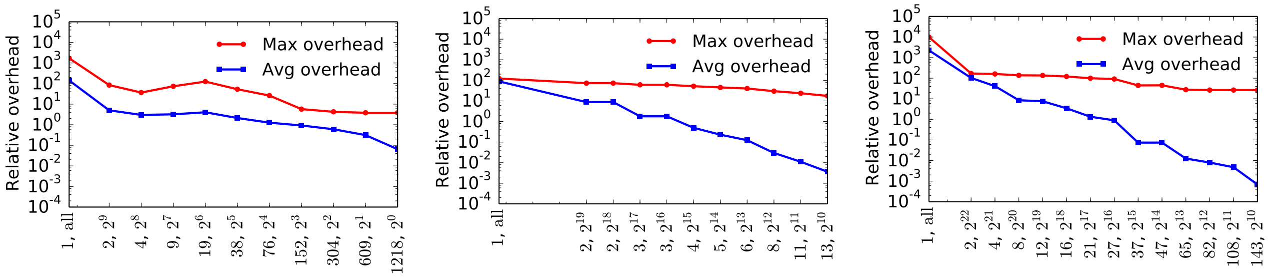

Figure 4: Relative padding overhead vs number of clusters and minimum cluster size (log-log scale) for three corpuses representing real-world file size distributions: (a) Youtube videos, (b) English Wiktionary, and (c) English Wikipedia

6.4. Document server

Next, we measured Pacer's impact on the throughput of the document server serving the English Wiktionary corpus. Clients request different pages concurrently and synchronously for a period of 120s. Prior to the measurement, we ran the workload for 10s to warm up the caches. We used a coarse-grained clustering of the corpus with one cluster of 5,010,856 files up to 12KB in size and another with the remaining 16,488 files, which yields an average padding overhead of 150%. We requested the largest file from each cluster several times with varying number of concurrent clients. Additionally, we ran a trace where clients request a total of 1,000 files randomly chosen from both clusters.

Figure 5 shows the throughput vs average latency for the baseline and Pacer. The error bars show the standard deviations of the average latencies. We used the request trace workload ($\text{Trace}_{\text{Base}}$ and $\text{Trace}_{\text{Pacer}}$ correspond to baseline and Pacer respectively) and, for comparison, we also stressed the server with requests only to the largest file in the corpus (521.9KB) ($\text{Large}_{\text{Base}}$ and $\text{Large}_{\text{Pacer}}$ for baseline and Pacer).

Unlike the baseline, Pacer's latency remains constant until the maximal throughput, because latency is determined by the initial response delay in the transmission schedule. Once the server is at capacity, it fails to serve additional requests and clients time out. Pacer's latencies are higher than the baseline's because the profiler generates conservative schedules based on all samples observed during profiling. Nevertheless, the latencies remain within hundreds of milliseconds, and could be optimized substantially using different schedules for different load conditions.

Pacer incurs a 6.8% and 30% overhead on peak throughput for the trace workload and the large file, respectively. These figures reflect Pacer's total overhead, because they compare to a saturated baseline server. With the large file, the baseline operates at over 40% of the line rate. We believe that this result is limited by the accuracy of transmit schedules, which can be improved substantially.

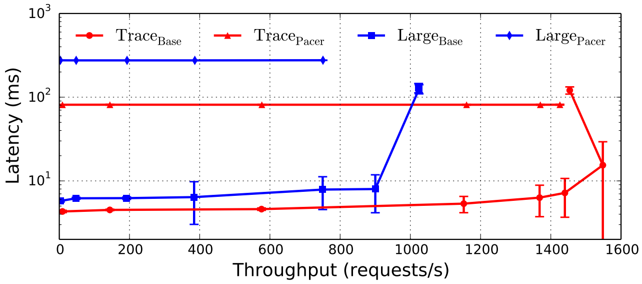

Figure 5: Document server throughput vs latency

7. Extensions

In this section, we discuss extensions to Pacer's design, to provide network side-channel mitigation for additional classes of applications. We are exploring the design and evaluation of these extensions in our ongoing work.

Interactive services Pacer currently shapes only the server-side traffic of an application, and assumes that the shape of client traffic reveals no secrets. This is sufficient for services with limited client-server interaction such as file downloading and video streaming. However, in more interactive applications, a client's request may depend on previous responses from the server. In such cases, the client request and even the shape of the client traffic may reveal secrets and must, therefore, be paced and padded. To handle such applications, Pacer's traffic-shaping support can be extended to clients by modifying the client browser or OS, or by running a Pacer-enabled hypervisor on the client. A transmit schedule for the application would cover an *entire interaction sequence* between the client and server, from the first request of the client to the last response of the server. We are currently working on this extension.

Multi-tier services Our current prototype of Pacer supports single-tier services, where a single server processes client requests without relying on other backend servers. Multi-tier services can be supported by running Pacer on all servers (frontend and backend) and extending transmission schedules to cover the entire communication graph of a request. However, in a multi-tier service, it may be challenging to determine the best-fitting communication graph from just the initial request.

So, we anticipate that, as in our current single-tier design, we would start processing a request using a default profile, which will be replaced with a custom profile after the structure of the remaining communication becomes clear. This is secure as long as the already played part of the default profile at the point of the replacement is a prefix of the new profile.

Dynamic content In applications like VoIP, the amount of traffic in each direction during a session depends on user actions. In these cases, the structure of the transmission schedule will be application-specific, but we believe that the structure can be determined for many applications. For instance, in VoIP sessions, the duration of VoIP calls can be bucketized, and for any given duration, traffic can be transmitted for that duration at a uniform rate. The granularity of buckets determines the trade-off between security and transmission-volume overhead, while the rate of transmission determines the trade-off between call quality and bandwidth overhead.

8. Related work

We discuss prior work on attacks using network side channels, compare to existing mitigation techniques, and also discuss technically related work with other threat models or completely different goals.

8.1. Attacks using network side channels

Network side-channel attacks can be launched by observing the total number and sizes of packets [18, 64, 21, 17], their timing [16, 28], and more coarse-grained information, such as burst lengths, the frequency of bursts, burst volumes [22, 73, 70], and a combination of such features [30, 43, 61]. Attacks have been shown to discover what a user is typing over SSH [63], which websites a user is visiting [31, 69], what videos are being streamed [60], the contents of live conversations [73] and private keys [13, 12], even with end-to-end encryption and techniques like onion routing in place [54].

Within the context of Cloud computing, the setting of this paper, Ristenpart *et al.* [59] and Inci *et al.* [32, 33] show that targeted attacks can be carried out by first attaining co-residency with a desired victim, and then exploiting side channels, including contention on I/O ports [59, Section 8.3]. Agarwal *et al.* [6] demonstrate a coarse-grained attack based on change in aggregate bandwidth consumption in a Cloud environment.

These attacks demonstrate the need for a solution such as Pacer that makes network traffic of a (Cloud-resident) victim independent of secrets.

8.2. Traffic-shaping systems to mitigate network side channels

Dependence of packet *size* on secrets can be eliminated by padding all packets to a fixed length [31, 73]. This standard technique is also used by Pacer. Making packet *timing* independent of secrets is substantially harder. A straw man is to send packets continuously at a *fixed* rate independent of the actual workload, inserting dummy packets when no actual packets exist [61, 63]. However, this either wastes bandwidth or incurs high latencies when the workload is bursty.

BuFLO [22] reduces this overhead by shaping response traffic to evenly-spaced *bursts* of a fixed number of packets for a certain minimum amount of time after a request starts. However, this leaks the size of responses that take longer than the minimum time. Tamaraw [15], CS-BuFLO [14], and DynaFlow [47] pad each response to some factor of the original size, such as the nearest power of 2. They offer no control over how many objects end up with the same traffic shape. In fact, a given traffic shape may correspond to a *single* sensitive object, which would be afforded no protection at all. In contrast to these systems, Pacer allows precise control over cluster size.

Additionally, CS-BuFLO and DynaFlow adapt traffic shapes based on the application's actual transmission rate. However, this rate may depend on application secrets. Thus, the choice of the traffic shape in these systems may leak secrets. In contrast, Pacer allows traffic shape adaptation based only on public inputs, so the traffic shape is secret-independent.

Traffic morphing [74] makes sensitive responses look like non-sensitive responses, but only shapes packet sizes and ignores packet timing. Walkie-Talkie [71], Supersequence [70], and Glove [50] cluster responses, and generate a traffic shape for the cluster that envelopes each response in the cluster. However, the traffic is shaped only for packet size and inter-packet spacing but not for the server response latency. This allows leaks via the application's delay up to the first response packet. In contrast, Pacer shapes packet size, inter-packet spacing, and the server response latency, thus eliminating all leaks by design.

Additionally, the systems described above do not defend against interference between the shaping component (enforcement) and the rest of the application stack. Hence, they would allow for potential leaks if integrated directly within the Cloud host, which is required in our context (§4). Pacer, on the other hand, is carefully designed and implemented to mask all secret-dependent delays in its pacing component (HyPace), as described in §4.1.

8.3. Mitigating network side channels in Clouds

Contention on NICs in a Cloud can be mitigated by time-division multiple access (TDMA) in a hypervisor [36] as this eliminates the adversary VM's (and, in fact, every VM's) ability to observe a co-located victim's traffic. However, this approach is inherently inefficient when the payload traffic is bursty. Statistical multiplexing, which only caps the total amount of data transmitted by a VM in an epoch, is fundamentally insecure because the resources available to a flow depend on the bandwidth utilization of other flows [29].

Another defense is to restrict the adversary VM's ability to observe time [68, 48, 45]. StopWatch [42] replaces a VM's clock with virtual time based only on that VM's execution. To mitigate network side channels, each VM is replicated 3×, the replicas are co-located with different guests, and each interrupt is delivered at a virtual time that is the median of the 3 times. This prevents a guest from consistently observing I/O interference with any co-located tenant. However, it also requires a 3× increase in deployed Cloud resources. In contrast, Pacer mitigates network side-channel leaks with far less resource overhead. Deterland [75] also replaces VMs' real time with virtual time, but it does not address leaks due to network side channels as it delivers I/O events to VMs without delay.

Bilal *et al.* [10] address leaks via the *pattern* of queries to different backend nodes in multi-tier stream-processing applications in a Cloud, but they do not consider leaks due to packet size and timing, which is what Pacer focuses on.

8.4. Predictive mitigation

Predictive mitigation [8, 79] mitigates network timing side channels and covert channels, but in a threat model fundamentally different from Pacer's. In Pacer, the threat is from a co-located tenant or a network adversary that cannot compromise (or authenticate as) legitimate clients of the victim. In contrast, in predictive mitigation the threat is from legitimate (or authenticated) clients of the victim who may have been compromised. In this setting, the adversary can always distinguish real packets from dummy packets, so predictive mitigation does not rely on dummy packets and uses a fundamentally different shaping strategy: At each scheduled transmission, the enforcement mechanism transmits a packet *only if* the application has actually provided one. Otherwise, nothing is transmitted and an information leak (up to 1 bit) is incurred due to the absence of a packet on the wire. After any such leak, the schedule is *adjusted* using a prediction algorithm (based only on public inputs) to reduce the chances of a leak in the future. In contrast, owing to its different threat model, Pacer is able to send a dummy packet when the application does not provide a real packet before a scheduled transmission. This prevents leaks completely.

Nonetheless, Pacer and predictive mitigation share a key technical idea: Both partition application workloads based on public inputs and compute a traffic shape for each partition ahead of time. The difference is that a badly chosen shape for a partition can leak information in predictive mitigation, but it only affects performance in Pacer.

Finally, the prototype implementation of predictive mitigation does not prevent or mask interference between the pacing logic and the application, which may result in timing leaks from the application to the paced traffic. Hence, that implementation cannot be used in our IaaS context without significant changes (§4).

8.5. Related work with other security goals

Metadata-privacy systems Herd [41], Vuvuzela [66], Karaoke [39], and Yodel [40] provide metadata privacy: they prevent information about who is communicating with whom from leaking via network side channels. However, these systems do not address leaks via metadata such as the lengths of application messages or calls. Pacer's goal is to prevent sensitive data from leaking via network side channels, which is fundamentally different from that of the above systems. To address its goal, in addition to shaping the sizes and timing of individual packets, Pacer shapes the lengths of application messages. Further, although metadata-privacy systems and Pacer share some underlying techniques such as the use of fixed size packets and dummy packets to shape traffic, Pacer additionally masks interference between the application and HyPace, thus preventing any leak of sensitive data via timing channels.

Censorship circumvention systems Systems like Format-Transforming Encryption (FTE) [23], SkypeMorph [49] and ScrambleSuit [72] use a tunnel abstraction similar to Pacer's conceptual design (§3) to modify payload traffic to bypass a traffic censor's filters. This goal is different from Pacer's goal of decorrelating observable traffic from secrets, and the tunnel's design depends on the assumptions about the censor's filters. For example, FTE [23] circumvents filters that use only deep-packet inspection (DPI), but not packet size or timing information. Hence, FTE's tunnel does not shape the traffic at all and offers no protection against threats that Pacer defends against.

ScrambleSuit [72] seeks to bypass censors that may inspect packet sizes and timing. It shapes packet size within the tunnel to a distribution picked ahead of time, independent of any secrets. Consequently, it decorrelates packet sizes from secrets. However, it only weakly obfuscates packet timing by adding a bounded random delay to every payload packet. This does not hide long (secret-dependent) inter-packet gaps, and may leak information, unlike Pacer.

SkypeMorph [49] has goals similar to ScrambleSuit. It extends traffic morphing [74] to sample both the inter-packet gap and the packet size from a fixed distribution, which mimics the distribution of some target protocol that the censor is assumed to allow. Dummy traffic is sent when the application does not produce sufficient traffic in time. While this approach could securely mitigate network side-channel leaks as well, it transmits traffic *continuously* at the average transmission rate of the target protocol. This either wastes bandwidth or causes significant latency overhead when the payload traffic is bursty. In contrast, Pacer allows adapting the transmission rate for every request based on public parameters of the request, thus limiting overheads on both bandwidth and latency.

Additionally, unlike Pacer, the implementations of SkypeMorph and ScrambleSuit do not mask interference between the application and the pacing component and, hence, both implementations could suffer from leaks when integrated with the Cloud server.

Oblivious computing Oblivious computing systems [20, 24, 46] generally prevent accessed memory *addresses* or accessed database *keys* from depending on secrets. Pacer addresses the orthogonal problem of making packet size and timing independent of secrets. As such, the techniques used in the two lines of work are completely different — oblivious computing generally relies on ORAM techniques, while Pacer relies on traffic shaping. However, Fletcher *et al.* [26] address *timing* leaks in ORAM accesses by pacing ORAM accesses. While this is superficially similar to Pacer's pacing of general network traffic, the pacing mechanism used by Fletcher *et al.* changes the pacing rate periodically based on the past actual request rate of the program, which may be secret-dependent. In contrast to Pacer's design, this leaks information.

8.6. Related work with non-security goals

Some prior work [57, 51, 19] uses techniques similar to those used by Pacer to isolate co-located tenants to make their *performance* predictable (not for security goals). Silo [35] implements traffic pacing in the hypervisor like Pacer. However, Silo's goal is to improve remote access latency, not information security, and, hence, its pacing logic is very different. MITTS [81] "shapes" memory traffic on CPU cores for performance and fairness, whereas Pacer shapes network traffic for security, so the goals and approaches are also very different. Richter *et al.* [58] propose to performance-isolate co-located tenants by modifying the NIC firmware. Pacer's traffic shaping can be similarly implemented even in the NIC. This would provide strong isolation of the pacing logic from the rest of the system in the face of micro-architectural side channels.

9. Conclusions

We have presented and evaluated Pacer, a cloaked tunnel implementation for an IaaS Cloud. Pacer shapes a tenant's network traffic to be independent of secrets, thereby eliminating network side-channel leaks to adversaries who can achieve colocation in the same server, rack, or datacenter, or otherwise observe victim's traffic. Pacer reduces overhead by allowing the tenant's traffic shape to depend on non-secret (public) attributes of its workload. A key technical challenge in Pacer is implementing its pacing component within the IaaS hypervisor. This is necessary to thwart attacks from co-located tenants and difficult because it requires masking the timing effects of interference from secret-dependent concurrent computation in tenant VMs. Experiments with video and document servers show that Pacer achieves a bandwidth overhead that is minimal given the tenant's workload partitioning, tolerable delay even with a simple, automatic profiler, and generally has modest runtime overhead. Pacer's design can be generalized to cover client traffic, multi-tier systems, and general VPNs, but the details of these remain future work.

References

[1] Azure confidential computing. `https://azure.microsoft.com/en-us/solutions/confidential-compute/`.

[2] NapaTech SmartNIC, Feature Overview Data Sheet. `https://www.napatech.com/support/resources/data-sheets/napatech-smartnic-feature-overview/`.

[3] wrk2: A constant throughput, correct latency recording variant of wrk. `https://github.com/giltene/wrk2`.

[4] Xen Null scheduler. `https://patchwork.kernel.org/patch/9669405/`.

[5] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Cryptographers' Track at the RSA Conference*, 2007.

[6] Yatharth Agarwal, Vishnu Murale, Jason Hennessey, Kyle Hogan, and Mayank Varia. Moving in next door: Network flooding as a side channel in cloud environments. In *Intl. Conf. on Cryptology and Network Security (CANS)*, 2016.

[7] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Fine grain Cross-VM Attacks on Xen and VMware are possible! In *IEEE Intl. Conf. on Big Data and Cloud Computing (BDCLOUD)*, 2014.

[8] Aslan Askarov, Danfeng Zhang, and Andrew C Myers. Predictive black-box mitigation of timing channels. In *ACM Conf. on Computer and Communications Security (CCS)*, 2010.

[9] Daniel J Bernstein. Cache-timing attacks on AES. `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`, 2005.

[10] Muhammad Bilal, Hassan Alsibyani, and Marco Canini. Mitigating network side channel leakage for stream processing systems in trusted execution environments. In *ACM Intl. Conf. on Distributed and Event-based Systems (DEBS)*, 2018.

[11] Benjamin A Braun, Suman Jana, and Dan Boneh. Robust and efficient elimination of cache and timing side channels. *arXiv preprint arXiv:1506.00189*, 2015.

[12] Billy Bob Brumley and Nicola Tuveri. Remote timing attacks are still practical. In *European Symposium on Research in Computer Security (ESORICS)*, 2011.

[13] David Brumley and Dan Boneh. Remote timing attacks are practical. *Computer Networks*, 48(5), 2005.

[14] Xiang Cai, Rishab Nithyanand, and Rob Johnson. Cs-buflo: A congestion sensitive website fingerprinting defense. In *Workshop on Privacy in the Electronic Society (WPES)*, 2014.

[15] Xiang Cai, Rishab Nithyanand, Tao Wang, Rob Johnson, and Ian Goldberg. A systematic approach to developing and evaluating website fingerprinting defenses. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[16] Xiang Cai, Xin Cheng Zhang, Brijesh Joshi, and Rob Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *ACM Conf. on Computer and Communications Security (CCS)*, 2012.

[17] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *IEEE Symposium on Security and Privacy (SP)*, 2010.

[18] Heyning Cheng and Ron Avnur. Traffic analysis of ssl encrypted web browsing, 1998.

[19] Ron Chi-Lung Chiang, Sundaresan Rajasekaran, Nan Zhang, and H. Howie Huang. Swiper: Exploiting virtual machine vulnerability in third-party clouds with competition for I/O resources. *IEEE Trans. on Parallel and Distributed Systems (TPDS)*, 26(6), 2015.

[20] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. Obladi: Oblivious serializable transactions in the cloud. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[21] George Danezis. Traffic Analysis of the HTTP Protocol over TLS. `http://www0.cs.ucl.ac.uk/staff/G.Danezis/papers/TLSanon.pdf`, 2009.

[22] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *IEEE Symposium on Security and Privacy (SP)*, 2012.

[23] Kevin P Dyer, Scott E Coull, Thomas Ristenpart, and Thomas Shrimpton. Protocol misidentification made easy with format-transforming encryption. In *ACM Conf. on Computer and Communications Security (CCS)*, 2013.

[24] Saba Eskandarian and Matei Zaharia. An oblivious general-purpose SQL database for the cloud. *CoRR*, abs/1710.00458, 2017.

[25] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *IEEE/ACM Intl. Symposium on Microarchitecture (MICRO)*, 2016.

[26] Christopher W Fletchery, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. Suppressing the oblivious ram timing channel while making information leakage and program efficiency trade-offs. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

[27] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 2016.

[28] Xun Gong, Nikita Borisov, Negar Kiyavash, and Nabil Schear. Website detection using remote traffic analysis. In *Privacy Enhancing Technologies Symposium (PETS)*, 2012.

[29] Xun Gong and Negar Kiyavash. Quantifying the information leakage in timing side channels in deterministic work-conserving schedulers. *IEEE/ACM Trans. on Networking (TON)*, 24(3), 2016.

[30] Jamie Hayes and George Danezis. k-fingerprinting: A robust scalable website fingerprinting technique. In *USENIX Security Symposium*, 2016.

[31] Andrew Hintz. Fingerprinting websites using traffic analysis. In *Conf. on Privacy Enhancing Technologies (PETS)*, 2002.

[32] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! cross-vm rsa key recovery in a public cloud. *IACR Cryptology ePrint Archive*, 2015(1-15), 2015.

[33] Mehmet Sinan İnci, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Efficient, adversarial neighbor discovery using logical channels on microsoft azure. In *Annual Conf. on Computer Security Applications (ACSAC)*, 2016.

[34] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing–and Its Application to AES. In *IEEE Symposium on Security and Privacy (SP)*, 2015.

[35] Keon Jang, Justine Sherry, Hitesh Ballani, and Toby Moncaster. Silo: Predictable message latency in the cloud. In *ACM Conf. on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[36] Sachin Kadloor, Negar Kiyavash, and Parv Venkitasubramaniam. Mitigating timing side channel in shared schedulers. *IEEE/ACM Trans. on Networking (TON)*, 24(3), 2016.

[37] Paul Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Advances in Cryptology – CRYPTO*, 1996.

[38] Adam Langley, Alistair Riddoch, Alyssa Wilk, Antonio Vicente, Charles Krasic, Dan Zhang, Fan Yang, Fedor Kouranov, Ian Swett, Janardhan Iyengar, et al. The quic transport protocol: Design and internet-scale deployment. In *ACM Conf. on Special Interest Group on Data Communication (SIGCOMM)*, 2017.

[39] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Karaoke: Distributed private messaging immune to passive traffic analysis. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[40] David Lazar, Yossi Gilad, and Nickolai Zeldovich. Yodel: strong metadata security for voice calls. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2019.

[41] Stevens Le Blond, David Choffnes, William Caldwell, Peter Druschel, and Nicholas Merritt. Herd: A scalable, traffic analysis resistant anonymity network for VoIP systems. In *ACM Conf. on Special Interest Group on Data Communication (SIGCOMM)*, 2015.

[42] Peng Li, Debin Gao, and Michael K Reiter. Stopwatch: a cloud architecture for timing channel mitigation. *ACM Trans. on Information and System Security (TISSEC)*, 17(2), 2014.

[43] Shuai Li, Huajun Guo, and Nicholas Hopper. Measuring information leakage in website fingerprinting attacks and defenses. In *ACM Conf. on Computer and Communications Security (CCS)*, 2018.

[44] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (SP)*, 2015.

[45] Weijie Liu, Debin Gao, and Michael K Reiter. On-demand time blurring to support side-channel defense. In *European Symposium on Research in Computer Security (ESORICS)*, 2017.

[46] Jacob R Lorch, Bryan Parno, James Mickens, Mariana Raykova, and Joshua Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[47] David Lu, Sanjit Bhat, Albert Kwon, and Srinivas Devadas. Dynaflow: An efficient website fingerprinting defense based on dynamically-adjusting flows. In *Workshop on Privacy in the Electronic Society (WPES)*, 2018.

[48] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *Intl. Symposium on Computer Architecture (ISCA)*, 2012.

[49] Hooman Mohajeri Moghaddam, Baiyu Li, Mohammad Derakhshani, and Ian Goldberg. Skypemorph: Protocol obfuscation for tor bridges. In *ACM Conf. on Computer and Communications Security (CCS)*, 2012.

[50] Rishab Nithyanand, Xiang Cai, and Rob Johnson. Glove: A bespoke website fingerprinting defense. In *Workshop on Privacy in the Electronic Society (WPES)*, 2014.

[51] Diego Ongaro, Alan L Cox, and Scott Rixner. Scheduling I/O in virtual machine monitors. In *ACM SIGPLAN/SIGOPS Intl. Conf. on Virtual Execution Environments (VEE)*, 2008.

[52] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *The Cryptographers' Track at the RSA Conf. on Topics in Cryptology (CT-RSA)*, 2006.

[53] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. `http://www.cs.bris.ac.uk/Publications/pub_info.jsp?id=1000625`, 2002.

[54] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. Website fingerprinting in onion routing based anonymization networks. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2011.

[55] Colin Percival. Cache missing for fun and profit. In *BSDCan*, 2005.

[56] Peter Pessl, Daniel Gruss, Clementine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium*, 2016.

[57] Xing Pu, Ling Liu, Yiduo Mei, Sankaran Sivathanu, Younggyun Koh, Calton Pu, and Yuanda Cao. Who Is Your Neighbor: Net I/O Performance Interference in Virtualized Clouds. *IEEE Trans. on Services Computing*, 6(3), 2013.

[58] Andre Richter, Christian Herber, Stefan Wallentowitz, Thomas Wild, and Andreas Herkersdorf. A Hardware/Software Approach for Mitigating Performance Interference Effects in Virtualized Environments Using SR-IOV. In *IEEE Intl. Conf. on Cloud Computing (CLOUD)*, 2015.

[59] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds. In *ACM Conf. on Computer and Communications Security (CCS)*, 2009.

[60] Eran Tromer Roei Schuster, Vitaly Shmatikov. Beauty and the Burst: Remote Identification of Encrypted Video Streams. In *USENIX Security Symposium*, 2017.

[61] T Scott Saponas, Jonathan Lester, Carl Hartung, Sameer Agarwal, Tadayoshi Kohno, et al. Devices that tell on you: Privacy trends in consumer ubiquitous computing. In *USENIX Security Symposium*, 2007.

[62] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. Netspectre: Read arbitrary memory over network. *CoRR*, abs/1807.10535, 2018.

[63] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *USENIX Security Symposium*, 2001.

[64] Qixiang Sun, Daniel R. Simon, Yi-Min Wang, Wilf Russell, Venkata N. Padmanabhan, and Lili Qiu. Statistical identification of encrypted web browsing traffic. In *IEEE Symposium on Security and Privacy (SP)*, 2002.

[65] Leif Uhsadel, Andy Georges, and Ingrid Verbauwhede. Exploiting hardware performance counters. In *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 2008.

[66] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Symposium on Operating Systems Principles (SOSP)*, 2015.

[67] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M Swift. Scheduler-based defenses against cross-vm side-channels. In *USENIX Security Symposium*, 2014.

[68] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *ACM workshop on Cloud Computing Security Workshop*, 2011.

[69] Pepe Vila and Boris Köpf. Loophole: Timing attacks on shared event loops in chrome. In *USENIX Security Symposium*, 2017.

[70] Tao Wang, Xiang Cai, Rishab Nithyanand, Rob Johnson, and Ian Goldberg. Effective attacks and provable defenses for website fingerprinting. In *USENIX Security Symposium*, 2014.

[71] Tao Wang and Ian Goldberg. Walkie-talkie: An efficient defense against passive website fingerprinting attacks. In *USENIX Security Symposium*, 2017.

[72] Philipp Winter, Tobias Pulls, and Juergen Fuss. Scramblesuit: A polymorphic network protocol to circumvent censorship. In *ACM Workshop on Privacy in the Electronic Society (WPES)*, 2013.

[73] Charles V Wright, Lucas Ballard, Scott E Coull, Fabian Monrose, and Gerald M Masson. Spot me if you can: Uncovering spoken phrases in encrypted VoIP conversations. In *IEEE Symposium on Security and Privacy (SP)*, 2008.

[74] Charles V. Wright, Scott E. Coull, and Fabian Monrose. Traffic morphing: An efficient defense against statistical traffic analysis. In *Network and Distributed System Security Symposium (NDSS)*, 2009.

[75] Weiyi Wu and Bryan Ford. Deterministically deterring timing attacks in deterland. *arXiv preprint arXiv:1504.07070*, 2015.

[76] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (SP)*, 2015.

[77] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*, 2014.

[78] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: a timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering*, 7(2), 2017.

[79] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. Predictive Mitigation of Timing Channels in Interactive Systems. In *ACM Conf. on Computer and Communications Security (CCS)*, 2011.

[80] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *ACM Conf. on Computer and Communications Security (CCS)*, 2012.

[81] Yanqi Zhou and David Wentzlaff. Mitts: Memory inter-arrival time traffic shaping. *ACM SIGARCH Computer Architecture News*, 44(3), 2016.