Pacer: Comprehensive Network Side-Channel Mitigation in the Cloud

Aastha Mehta[1,2], Mohamed Alzayat[1], Roberta De Viti[1], Björn B. Brandenburg[1], Peter Druschel[1], and Deepak Garg[1]

[1]Max Planck Institute for Software Systems (MPI-SWS), Saarland Informatics Campus
[2]University of British Columbia (UBC)

Abstract

Network side channels (NSCs) leak secrets through packet timing and packet sizes. They are of particular concern in public IaaS Clouds, where any tenant may be able to colocate and indirectly observe a victim's traffic shape. We present Pacer, the first system that eliminates NSC leaks in public IaaS Clouds end-to-end. It builds on the principled technique of shaping guest traffic outside the guest to make the traffic shape independent of secrets by design. However, Pacer also addresses important concerns that have not been considered in prior work—it prevents internal side-channel leaks from affecting reshaped traffic, and it respects network flow control, congestion control and loss recovery signals. Pacer is implemented as a paravirtualizing extension to the host hypervisor, requiring modest changes to the hypervisor and the guest kernel, and only optional, minimal changes to applications. We present Pacer's key abstraction of a *cloaked tunnel*, describe its design and implementation, prove the security of important design aspects through a formal model, and show through an experimental evaluation that Pacer imposes moderate overheads on bandwidth, client latency, and server throughput, while thwarting attacks based on state-of-the-art CNN classifiers.

1 Introduction

Sharing resources is in the very nature of public Clouds. However, many side-channel leaks arise when mutually distrusting parties share hardware resources. Shared CPUs, cores, caches, and memory buses have all been exploited as side channels [25, 32, 43, 53, 59, 77–79]. As a result, side-channel leaks in Cloud environments are a growing concern for computer security research.

In this paper, we revisit a specific class of side-channel leaks—those arising from shared network elements—in the specific setting of public IaaS Clouds. These channels, called *network side channels* (NSCs), leak information via traffic shape (packet timing and packet size) even when packet payloads are encrypted. We argue below that such leaks ought to be a serious concern in public Clouds. We then describe key requirements for a practical, comprehensive defense that mitigates NSCs in public Clouds. Despite decades of work on mitigating NSCs, these requirements have not received much attention. We present Pacer, a new system that satisfies all the requirements and effectively mitigates NSCs in IaaS Clouds.

NSCs are a serious concern in Clouds. Prior work has shown that traffic shape is strongly correlated with secrets in many applications – traffic shape can reveal sensitive information about webpages [15, 17, 21, 26, 28, 29, 42, 63, 70], video streams [58], VoIP chats [73], users' keystrokes [62], and even private keys [11, 12]. Chen *et al.* [16] demonstrate that users' medical conditions, family income, and investments can be gleaned from the encrypted traffic of healthcare, taxation, investment, and web search services provided as software-as-a-service (SaaS) offerings.

While many of these attacks relied on direct access to the victim's traffic, more recent work has shown that an *unprivileged adversary* can also indirectly infer the victim's traffic shape by inducing contention with the victim's traffic at a shared network element and measuring resulting variations in the adversary's own traffic shape [6, 56, 58]. In fact, we were able to create such an indirect attack to recognize streamed videos with 96% accuracy using a CNN classifier (§A). Such indirect attacks are of particular concern in public (IaaS) Clouds as adversaries can rent virtual machines (VMs) and even colocate with a victim's VM at low cost [30, 31, 56]. Hence, NSCs *should be a significant concern* for security researchers, Cloud tenants and Cloud providers alike.

Requirements for mitigating NSCs. Any comprehensive mitigation of NSC attacks in an IaaS Cloud must satisfy the following requirements. **R1.** The mitigation must prevent leaks through all aspects of the shape of transmitted traffic, with provable guarantees. **R2.** In line with basic Cloud philosophy, the mitigation must allow for elastic (dynamically adaptive) sharing of network resources. **R3.** Although some overhead for mitigating a threat as strong as NSCs is unavoidable, the mitigation should still permit responsive,

client-facing services and not require excessive resources. **R4.** The mitigation should work with any guest VM and accommodate bursty network traffic, with minimal application changes. In other words, the mitigation should be *general*.

These requirements rule out many NSC mitigation techniques, specifically those that prevent leaks via either packet timing or packet size but not both (violates R1) [60,74,76], do not handle bursty traffic (violates R4) [21,74], rely on multipath routing [20] or adding "best-effort" noise without strong guarantees (violates R1) [36,47], hard, static bandwidth reservation for tenants including TDMA (violates R2) [68], or application code rewriting (violates R4) [47].

A general approach that can meet these requirements is to change the *shape the traffic* in a dedicated system component *outside the application* to make it independent of secrets. The final shape can be learnt by the shaping component adaptively [7,46], or the application can provide it to the shaping component [13,71]. Although this approach has been considered in prior work, the Cloud setting and the public Internet have additional practical requirements that have *not been considered* in prior work: **R5.** The traffic-shaping logic must take flow control and loss recovery of the network protocol into account (else information may leak via the presence of ACKs in the reverse direction), and it must respect network congestion signals (else it could destabilize the network). **R6.** The traffic-shaping component must be integrated with Cloud servers—as opposed to routers or middleboxes—to prevent colocated attacker from exploiting contention on servers' network interface (NIC). Consequently, the shaping component must be performance-isolated from secret-carrying Cloud tenants to prevent *internal side-channel leaks within the server* from affecting the reshaped traffic. To the best of our knowledge, no prior work on NSC mitigation satisfies all of R1–R6.

Our contribution: Pacer. The requirements R1–R6 pose significant design and engineering challenges for a secure and practical NSC mitigation solution. To our knowledge, Pacer is the first end-to-end system that mitigates NSCs comprehensively addressing all requirements. Pacer's contribution is twofold: a novel *cloaked tunnel* abstraction that shapes traffic between two guests on different hosts end-to-end, and a realization of this abstraction for IaaS Clouds.

Briefly, a cloaked tunnel shapes application traffic to provably make it independent of secrets at the traffic's origin (R1). This eliminates NSC's by design. The tunnel multiplexes multiple flows at fine granularity (R2). The tunnel works with all IaaS VMs and unmodified applications, although, to improve efficiency, applications may *optionally* interact with the tunnel to specify which traffic shapes should be used on their flows to improve efficiency (R3); this requires only small changes to applications (R4). (The tunnel is secure as long as applications pick shapes independent of secrets.) Finally, by design, the tunnel is isolated from guest applications (R6) and it takes network congestion, flow control, and loss recovery into account when shaping traffic (R5). The cloaked tunnel described above is a general abstraction that mitigates NSC leaks in any setting, not just Clouds.

In addition, Pacer implements a *paravirtualized* instance of the cloaked tunnel integrated with IaaS Cloud servers. Pacer relies on a hypervisor component, called HyPace, and a guest kernel module, called GPace, which interacts with HyPace to facilitate congestion management, loss recovery, and flow control during shaping (R5). HyPace and GPace implement a novel *masking* mechanism to ensure timely packet transmission independent of guest delays, thus achieving performance isolation from the guests (R6). Furthermore, HyPace implements a secure *batching* mechanism to amortize the high costs of masking and sustain ~7.6 Gbps line rate (R3). An experimental evaluation of our prototype on two IaaS applications—a medical information site and a video streaming service—shows that Pacer defeats powerful NSCs with moderate overhead.

Organization. We present the threat model, design challenges, and key ideas behind Pacer in §2. We describe the general cloaked tunnel abstraction, define its requirements and properties, and argue its security in §3. We describe Pacer's implementation of a paravirtualized instance of the tunnel in IaaS Cloud servers in §4. We discuss generation of efficient transmit schedules for traffic shaping in §5. We present our implementation and empirical evaluation of Pacer's performance overheads and security in §6. We discuss related work in §7 and conclude in §8. Additionally, we present NSC attacks under various setups in §A and a detailed evaluation of the security of Pacer's masking mechanism in §B. Finally, we build an abstract formal model capturing relevant aspects of Pacer's design and prove its security in §C.

2 Overview

As a running example, we use the scenario of a patient who consults a trusted, Cloud-hosted medical website MedWeb for diagnostic and therapy options, medical procedures, and care providers in their area. The patient wishes to keep their condition from employers, health insurers, and other parties for fear of discrimination. We show how Pacer can ensure the patient's privacy by hiding the content they retrieve from colocated tenants and other network observers, with minimal modifications to MedWeb, and with modest overhead in network bandwidth and response time.

2.1 Threat model

The *victim* in the public IaaS Cloud is a tenant executing arbitrary computations in one or more guest VMs, and serving a set of *trusted clients* that connect to its VMs using IPSec or a VPN (virtual private network) with pre-shared key au-

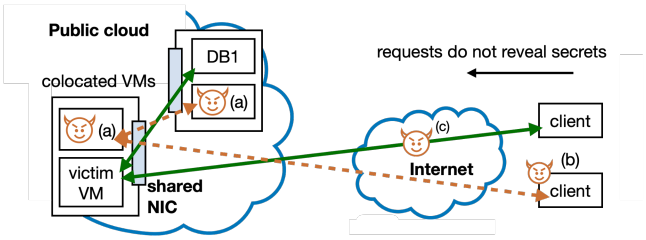

Figure 1: The adversary can (a) colocate with victim's VM or backend services in the Cloud, (b) control clients of its own VMs, and (c) use cross-traffic between any pair of these to infer the shape of the victim's traffic at shared network links.

thentication[1]. The MedWeb site, for instance, authenticates its registered clients using IPSec-PSK[2]. To serve a client request, the victim may invoke other Cloud backend services hosted on separate physical servers. The victim's goal is to protect its secrets; these secrets can be reflected in parameters of client requests (e.g., the name of a requested file), in the victim's internal state (e.g., which request handlers are cache-hot because they were recently accessed), or in the backend traffic.

Pacer's goal includes preventing NSC leaks of the victim's secrets to anyone able to rent other VMs in the Cloud. Prior work has shown that deliberate colocation with a victim VM is feasible [30, 31, 56]. Accordingly, we assume a strong adversary that may colocate 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. 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[3]. 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. The adversary controls network clients, which communicate with its VMs via the network. However, the adversary cannot break standard cryptography, break into the victim's VPN, impersonate/compromise the victim's clients, or connect to the instances of backend services used by the victim. While not the primary goal of our work, Pacer's design also protects against powerful adversaries who can directly observe the victim's traffic as well as delay, drop, and inject network packets (e.g., ISPs). Figure 1 summarizes Pacer's threat model.

Non-goals. Pacer addresses NSCs; we assume that microarchitectural side-channel leaks are mitigated by renting an entire server socket and the associated NUMA domain to the victim for exclusive use. Alternatively, Pacer can be combined with complementary work to mitigate side-channel leaks through other shared resources [10, 66]. The Cloud platform and provider are trusted.

Our focus is on protecting secrets within the content provided by a server. Hiding the identity of the service requested [21, 28, 29], the communication protocol used [22, 75], or the IP address [64] of the client are non-goals. Pacer can be combined with other techniques to address them. In our running example of MedWeb, the patient wishes to hide what specific disease, procedures and care facilities (s)he is interested in, not that (s)he is accessing a medical site and video service—most people do occasionally. Note also that hiding the patient's IP address alone (e.g., using Tor or a VPN) would be insufficient, because aspects of the content retrieved, e.g., the geographic location of care facilities the client retrieves, can reveal the patient's identity.

Prototype assumptions. Pacer's prototype additionally assumes that *clients' request* traffic reveals no secrets through its shape (its length, number of packets, or timing). In particular, the time of requests does not depend on any secrets or the actual completion times of previous responses. However, Pacer's design can support bidirectional traffic shaping, trivially by running the Pacer-enabled hypervisor and kernel, or a kernel with all of Pacer's functionalities on the client side.

2.2 Key ideas

Pacer avoids NSCs by ensuring the shape of the victim's network traffic is secret-independent. Ensuring secret-independence requires that: (i) The choice of traffic shape must not reveal secrets. For instance, if a constant rate of one 1.5kB packet per millisecond for 10 seconds is chosen to transmit a particular video, then this choice must not be specific to the video. (ii) If the actual packet transmission times deviate from the chosen shape, the deviations must not reflect application secrets. In our example, if the actual transmission time of a packet deviates from its precise expected time based on the rate, then this deviation must not reflect the CPU and memory consumption of the concurrent video processing in a way that may identify the video.

Secret-independent traffic shapes. A strawman secure shaping strategy is to continuously transmit fixed-size packets at fixed intervals independent of the application's actual workload; in the absence of application payload, dummy packets are transmitted. However, this strategy is highly inefficient when the application workload is bursty. Pacer instead allows the shape to vary as long as the variations do not depend on secrets. Specifically, if a guest can partition its workload into classes based on public information, Pacer permits the use of a different, efficient traffic shape in each class. Returning to our example, suppose MedWeb streams videos about medical procedures in different resolutions. Then, the shape used to stream a video can vary by resolution, which only reveals

[1] Guests may require a second level of authentication to separate clients' privileges, but this is not relevant for Pacer's security.

[2] Pacer requires IPSec-PSK as the timing of the tenant's response to an unauthenticated client's connection attempt may be affected by tenant's concurrent processing of other clients' secrets, thus revealing these secrets.

[3] Prior work has shown that traffic shape can be inferred through such methods by an adversary colocated on the same physical machine as well as an adversary contending on a downstream network link [58]. Thus, renting dedicated physical machines is insufficient to mitigate NSCs.

the patient's available bandwidth but not the specific medical procedure being watched.

Gray-box profiling. Secret-independent traffic shaping requires understanding how a guest's secrets affect its network traffic. This information can be obtained by black-box profiling of guests, but this approach cannot reliably discover all dependencies and therefore is not secure. Program analysis, which could discover all dependencies in principle, does not scale well. Pacer instead relies on gray-box profiling, which requires no knowledge of a guest's internals beyond a *traffic indicator* provided by the guest. This indicator partitions the guest's possible network interactions independent of secrets, and can be used to profile the guest's network interactions and generate a transmit schedule for each partition (§5).

Paravirtualized cloaked tunnel support. To enforce traffic shapes, 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, called HyPace, initiates transmissions based on a schedule. A guest kernel module, GPace, shares state with HyPace for schedule installation and adaptation based on network congestion, loss recovery, and flow control. HyPace's and GPace's execution can experience interference from the guest due to side channels, so Pacer uses a novel idea—it *masks* any execution delays in HyPace and GPace that could influence actual packet transmission times (R6).

3 Cloaked tunnel

In this section, we describe an idealized realization of the *cloaked tunnel* abstraction and security properties, independent of a specific application setting, implementation, or placement of tunnel entry and exits.

We begin with a discussion of three high-level properties required in a practical, secure cloaked tunnel. These properties mirror the requirements R1, R5, and R6 from §1. **P1.** *Secret-independent traffic shape:* Requires that transmissions follow a schedule that does not depend on secrets (R1), and that actual transmissions within a schedule are not delayed by potentially secret-dependent computations (R6). **P2.** *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 (R5); 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. **P3.** *Congestion control:* The tunnel must react to network congestion (R5). Congestion control is needed for network stability and fairness, but does not reveal secrets since it reacts to network conditions, which themselves depend only on shaped and third-party traffic.

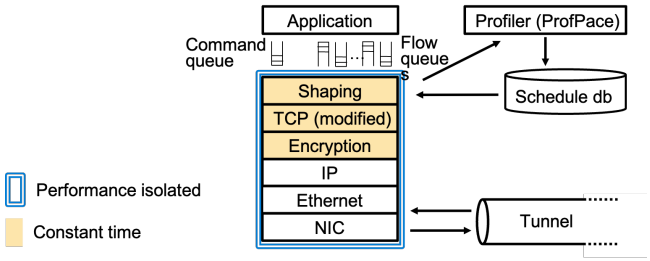

Figure 2: Cloaked tunnel (one endpoint)

3.1 Idealized tunnel design

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 stack consists of a *shaping* layer on top of a modified transport layer (e.g., TCP) on top of the encryption layer, e.g. IPSec. These layers rest on 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.[4]

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.

Overall, the shaping layer shapes each flow separately, and then multiplexes the shaped traffic of all flows onto the same physical links at the granularity of packets (R2 of §1).

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

Assumptions. The tunnel design presented here relies on some idealized assumptions, which are relaxed in the practical design of §4. To ensure that packet transmissions are not delayed by secret-dependent contributions (property P1), the design assumes that processing delays in the tunnel network stack are not influenced by secrets, even 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; and (ii) the execution of the tunnel network stack is *performance-isolated* from the application and any other computation.

Outbound data processing. A timestamp is taken whenever data is queued by the application; these timestamps and the

[4]We describe the tunnel in terms of TCP; however, another stack like QUIC [37] can also be used.

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 to indicate the amount of padding added.

Transport layer. The transport layer operates as normal, except for two tunnel-related modifications to satisfy R5: (i) When the congestion window closes, the transport layer signals the shaping layer to suspend the flow's transmit schedule until the 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. (ii) 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 outbound data processing 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 tunnel's other end.

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 schedule is associated with each flow's 5-tuple f and a traffic id *sid*, and can be of two types: *default* and *custom*. A default *sid* maps to a default schedule that is installed when the flow is created. This schedule acts as a template, which is instantiated automatically by the shaping layer whenever a 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 the arrival time of the packet that causes the schedule's instantiation.

A default schedule active on a flow can be optionally extended by a custom schedule in response to an application's *sid*. 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 shaping layer looks up the schedule associated with *sid* in the schedule database and associates it with flow f. Every custom schedule has a sufficiently large initial delay to allow the schedule to reach the tunnel endpoint before the first scheduled transmission, despite any queueing delays. Thus, the precise time of schedule extension remains unobservable to the adversary.

3.2 Tunnel security

The cloaked tunnel provides the following security property: *The shape of the traffic in the tunnel does not depend on secrets.* This property holds because our design ensures that each of the following is either independent of secrets or unobservable to a network adversary: **S1.** the chosen transmit schedules, **S2.** activations, pauses and resumes of transmit schedules, **S3.** timing of updates to active transmit schedules, **S4.** deviations from transmit schedules, and **S5.** transport-layer responses to transmitted packets.

S1 is secret-independent by assumption on how schedules are picked. **S2** and **S5** depend only on public network events (like congestion signals) by design. **S3** cannot be observed by a network adversary due to the delay at the beginning of custom schedules. **S4** is secret-independent because the network stack is performance-isolated from the application so there are no side-channel leaks of secrets *into* the network stack, and all processing on cleartext data in the network stack is constant-time so there are no internal secret-dependent delays *within* the network stack.

4 Pacer design

We now describe Pacer, a practical cloaked tunnel design in the context of a public IaaS Cloud. We first discuss constraints on the design space in the context of an IaaS Cloud. First, *the tunnel entry must be integrated with the IaaS server*. In an IaaS Cloud, colocated 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. Second, *shaping requires padding, which must be done at the transport layer to ensure it is unobservable*. Third, the conceptual tunnel design of §3 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.

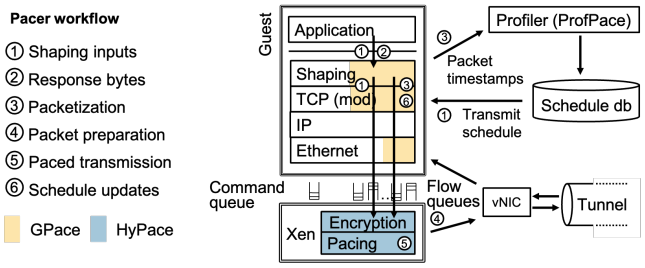

Figure 3: Pacer architecture and workflow.

One way to meet these requirements is to place the entire network stack in the hypervisor, performance-isolate it from guests, and implement it as constant time. However, this approach has many 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, it defeats NIC virtualization, such as SR-IOV, and requires guests and their network peers to use the network stack provided by the IaaS platform. Lastly, it complicates the hypervisor significantly.

Pacer architecture. Pacer addresses the tension outlined above using a paravirtualization approach. The responsibilities are divided between the hypervisor and the guest OS such 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) the guest OS changes are modest. We extend the 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 and workflow. 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* component plugs into Xen and provides these functions. The *GPace* component, a Linux kernel module, plugs into the guest OS and the OS of network clients that interact with the guest. It implements the cloaked tunnel in cooperation with HyPace.

The guest has direct access to a SR-IOV virtual NIC (vNIC) configured by the hypervisor, which it uses to receive but not to transmit packets. When the guest application receives a request, it sends a traffic indicator to GPace, which shares the indicator along with the flow's 5-tuple, TCP sequence number, congestion window, and crypto key in a per-flow datastructure to HyPace. HyPace instantiates a transmit schedule based on the indicator ①. When the application forwards response bytes to GPace ②, GPace splits the payload into MTU-sized packets with necessary padding, placing them in the per-flow structure, and timestamps the outgoing packets, sharing the flow's traffic profile with ProfPace ③. GPace also generates retransmission packets for both payload and dummy packets. At a scheduled transmit time, HyPace picks a payload packet or generates a dummy packet, encrypts and adds MACs to the packet, and places it in the NIC transmission queue ④. HyPace initiates NIC transmission after *masking* potentially secret-dependent delays in its execution ⑤. Additionally, it adapts the schedules in response to network events (e.g., congestion and retransmission) based on GPace's signals ⑥.

By generating dummy packets subject to congestion control and independently from the guest network stack, Pacer requires performance-isolation only for HyPace but not the guest. Overall, Pacer's security properties remain equivalent to those of the conceptual cloaked tunnel design (§3.2), as we discuss in §4.3.

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, congestion control, and retransmissions 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. Next, it initiates the transmission of the payload or dummy packet, reduces the congestion window accordingly, and initiates a retransmission timeout for the packet. Finally, in case of retransmissions (either due to a timeout on expected ACKs or due to receiving duplicate ACKs), HyPace extends the transmit schedule with a slot for every packet retransmitted by GPace. Unlike the generic tunnel, where the shaping occurs above the transport layer, this schedule extension is necessary to enable retransmissions. Note that extending schedules in response to retransmission events is secure because retransmissions occur only when there are packet losses in the network, which are publicly observable.

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 transmit schedule objects; the current TCP sequence number and the right edge of the congestion window; 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 and update the estimate whenever a new maximum is observed at any time during a system's execution. 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. (i) Our extensive empirical observations indicate that the instances in the tail of the event handler delay distributions tend to occur very infrequently and never in short succession[5]. As a result, the maximal delay for transmitting n packets in a single event handler activation does not increase much with n. Hence, we can amortize the overhead of masking handler delays over n packets. (ii) Actual transmission times can be delayed as long as the delay does not depend on secrets. Hence, 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 [3]. We plan to investigate NIC support in future work.

HyPace summary. HyPace is a minimal 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 (re-)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. At the same time, the batched transmission

[5] Without the knowledge of Intel CPU internals, it is difficult to determine the exact cause of the tail latencies, but their frequency suggests that they may be caused by system management interrupts.

design amortizes the high cost of masking and helps to sustain packet transmission throughput close to the NIC's line rate.

4.2 GPace

GPace is a kernel module that implements a cloaked tunnel jointly with HyPace[6]. 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. Furthermore, in case of a retransmission, GPace starts with retransmitting the first unacknowledged TCP sequence number. If this sequence number is for a dummy, GPace generates a dummy packet and sends it to HyPace, which eventually transmits it at a scheduled time.

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 used 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[7]. 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 and TCP timeouts in the guest, which is not performance-isolated. Thus, the delay between two causally related network events e_1 and e_2 must be made independent of actual processing delays in the guest, which may otherwise reveal secrets.

There are three relevant causally related pairs of events: 1) The arrival of the first packet of a request (e_1), which triggers the instantiation of a default schedule with start time equal to e_1's timestamp, and the subsequent transmission of a packet (e_2) according to the schedule, 2) An incoming ACK (e_1) that either causes a retransmission or opens the congestion window and triggers the next packet transmission (e_2), and 3) a network event (e_1) that sets a timer and subsequently causes a retransmission when the timer expires (e_2).

[6] On the client-side, GPace terminates the tunnel in the kernel.

[7] ACKs are not padded as Pacer does not need to hide client traffic shape. However, ACKs are paced to hide guest's interference with their transmission.

In each case, GPace uses masking to hide variability in the processing time between e_1 and e_2. Let ε be HyPace's epoch length, δ_{delay} be the guest OS's empirical *maximum* inbound packet- and timer-processing time, and $\delta = \varepsilon + \delta_{delay}$. Then, for (1): GPace requires that the initial response time of any default schedule be larger than δ; for (2): GPace ensures that e_2 is scheduled no earlier than δ after e_1; for (3): GPace ensures that e_2 is scheduled no earlier than δ after the timeout. These rules make the guest's actual processing time between causally related network events unobservable to the adversary.

GPace summary. GPace is a Linux kernel module that implements both ends of a cloaked tunnel, using the paravirtualized support from HyPace. It handles padding in payload packets, shares outgoing packets with HyPace along with per-flow sequence numbers and congestion window state, signals HyPace on installation of a new transmit schedule or update of a transmit schedule, and masks processing delays between pairs of causally related network events.

4.3 Pacer security

We built an abstract formal model of HyPace, the guest and the network, covering essential details such as delays due to internal side channels and HyPace's schedule replacement. We formally proved that our design provides the standard, strong security property of *noninterference* [61]—adversaries learn nothing about guest secrets (in an information-theoretic sense) despite observing traffic shape. The formal model and the proof are presented in §C.

Here, we provide some intuitive justification of Pacer's security. First, Pacer's threat model rules out side-channel leaks to other co-located tenants through shared CPU state, caches, memory bandwidth and shared Cloud back-end services. Second, it is impossible to connect to the victim tenant as a (fake) client and elicit even one response packet because Pacer requires packet authentication with pre-shared keys and GPace silently ignores all unauthenticated packets. Third, the adversary cannot learn secrets by measuring the shape of the victim's traffic because, like the cloaked tunnel of §3, Pacer ensures that the shape of outgoing traffic does not depend on secrets. This holds because **S1**–**S5** from §3.2 are either unobservable or independent of secrets for Pacer as well. The only nontrivial difference is in the secret-independence of **S4**: while the cloaked tunnel relies on performance-isolation and a constant-time implementation of the network stack, Pacer relies on the empirical delay-masking mechanisms as above.

Among the empirical Pacer parameters, only δ_{xmit} and δ_{delay} are relevant for security; all others like the epoch length and batch size merely affect performance. If actual delays exceed these two parameters, the actual runtime of the transmit handler or the inbound packet/timer handlers could be exposed, which may be correlated with victim secrets.

However, to exploit this vector, a colocated adversary would have to first find a way to cause a delay in the exe-

cution of these handlers beyond what was observed during Pacer's systematic training phase for computing the masking delays. This is difficult because the adversary is unprivileged relative to handler executions in both the guest kernel and the hypervisor and, hence, limited in its ability to cause these handlers to preempt. Second, the adversary would have to extract the secret from the observed run time. This is difficult because the adversary does not generally know the nature of the correlation between the secret and the observed run time. The adversary cannot rely on statistical inference since it can observe only a single instance of a parameter violation (Pacer updates the parameter whenever a new maximum delay is observed). We discuss the security of Pacer's masking in §B.

5 Efficient transmission schedules

By default, Pacer can use the same transmit schedule for all of a guest's network traffic. This approach does not require any support from tenant applications 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 using public information, no information is leaked. Here, we discuss how tenants can safely partition their workload, and use automatically generated, efficient schedules for each workload partition.

5.1 Traffic indicators

To use custom schedules, a tenant needs to provide *traffic indicators*. These indicators are used by Pacer to instantiate schedules and, along with other logged information, can be used by *ProfPace* to produce transmission schedules automatically (ProfPace is explained later).

In more detail, traffic indicators are integer-valued events that a guest generates at appropriate points in its execution. The indicators serve two purposes: (i) They indicate the onset of a sequence of transmissions of the class corresponding to the integer *sid* argument. This information is used by Pacer to instantiate an appropriate transmission schedule for the sequence. (ii) They delimit semantically related packet exchanges within a network flow, e.g., a client request from the guest's corresponding response. The integer *sid* value of the indicator identifies the equivalence class of the exchange, e.g., a TCP handshake, a TLS handshake, or the workload partition to which the request and its response belong, such as the video resolution in case of MedWeb.

Instrumenting guests. Instrumenting guests to provide traffic indicators is straightforward. A guest that responds to client requests on the network, for instance, simply invokes an IOCTL call on the network socket before it sends the response. A client, on the other hand, calls IOCTL on a new socket to install a schedule before it sends a request. In §6, we describe how we instrumented Apache and the PHP applications we use to provide traffic indicators. Pacer ensures that the precise timing of the IOCTL call, which could reveal secrets, is not reflected in the start time of a transmission schedule. If the schedule is instantiated in response to a network request, then the schedule is anchored at the request's arrival time (see §3.1). Otherwise, the schedule is anchored at a fixed offset from a public event like the top of the hour.

5.2 Choosing workload partitions

The tenant provides a *sid* value with each indicator, which identifies the workload partition and enables Pacer to use an efficient schedule. For *performance*, the tenant's choice of *sid* values should partition the guest's network traffic into classes of similar shape. The lower the variance of shapes in each class, the less the padding required when a specific network response is generated, minimizing bandwidth overhead. Returning to video streaming in our running MedWeb example, there should be a different *sid* value for every resolution, and the application should provide this *sid* for all videos of this resolution.

For *security*, it is sufficient that the choice of *sid* does not depend on secrets. First, certain network interaction patterns are well-known and don't reveal secrets. For instance, a network server's traffic typically consists of a TCP handshake, a TLS handshake, and a variable number of requests and responses on the established connected, followed by a connection show-down. Using a different *sid* for each is safe. Second, the tenant may partition its request-response workload into equivalence classes, such that the chosen traffic shape reveals the class but not the specific object requested within a class. Returning to the MedWeb example, all videos with a given resolution may be considered a class, for which the same *sid* is used and therefore an efficient traffic shape (rate) for that class. If all videos are available in the same set of resolutions, then the resolution reveals no information about the content requested.

A tenant may choose to further partition its workload into clusters such that the cluster of a requested object is public, but not the specific object. We discuss clustering heuristics next.

5.3 Clustering

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 object to the largest one in the corpus, incurring a large overhead. Suppose now the guest can partition the corpus such that each partition contains objects of similar size, but revealing the partition of a requested object is not a secret as would be the case when each partition contains a sufficient large number of objects. Now, each object can be padded to the largest object in *its* cluster, which may reduce

overhead significantly without revealing which object within a partition is being requested.

Determining what clustering is sufficiently private for a specific content service given its corpus's size and popularity distribution is beyond the scope of this paper. We merely highlight here the large efficiency gains possible when clustering content with skewed size distributions.

We describe heuristic clustering algorithms for videos and static HTML documents that minimize overhead subject to a *given* privacy need, which is defined in terms of the minimum number of objects in any cluster.

Video clustering. We cluster videos according to the sequence and sizes of their 5s segments using the following algorithm. Note that dynamically compressed segments 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 a cluster the set of videos minimizing the average relative padding overhead per video, *i.e.*, $\frac{1}{c_i}\sum_{j=1}^{c_i}\sum_{k=1}^{l_i}\left(\frac{(s_k - s_{kj})}{s_{kj}}\right)$, 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 kth segment across all videos in the set (*i.e.*, $\max_{1\leq j\leq c_i}(s_{kj})$). 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, its videos are ignored for later rounds. The algorithm stops when all videos are clustered. If the last cluster has less than c videos, it is merged with the one before it.

Document clustering. Unlike videos, HTML documents contain a single data object. Therefore the algorithm clusters based on the single size parameter of documents, and the largest document in a cluster constitutes the cluster's ceiling.

More sophisticated clustering algorithms that account for distinct per-object privacy requirements (popularity) and overall privacy requirements are left to future work. We present overheads of clustering real videos and documents in §6.2.

5.4 ProfPace

The ProfPace gray-box profiler automatically generates a transmit schedule for each traffic class as follows. GPace (§4.2) logs the application-provided traffic indicators (§5) 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, a userspace process in the guest, analyzes these logs to compute transmit schedules. Specifically, ProfPace segregates the logs into network interaction segments and bins them by different values of traffic indicators, *sid*s. The set of observed segments in a bin are considered samples of the associated equivalence class of network interactions.

Library	Traffic indicator	Traffic class
Apache	Before listen()	TCP handshake
Apache	Before accept()	SSL handshake
Apache	Before last SSL handshake msg	HTTP requests
MediaWiki	After parsing requested page title	Page cluster
Video	After parsing video title/segment	Video cluster, segment
Video	Before connect() to memcache	TCP handshake
Video	Before get request to memcache	request (1 MTU)
Memcache	Before listen()	TCP handshake
Memcache	After parsing video title/segment	Video cluster, segment

Table 1: Locations of traffic indicator instrumentation

ProfPace characterizes the traffic shape for each class with a set of random variables: (i) the delay between the first incoming packet and the first response packet d_i, (ii) the delay between subsequent response packets d_s, and (iii) the number of response packets p. For each equivalence class of network interactions, 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 99th percentile of the initial delay d_i and the 90th percentile of the spacing among subsequent packets d_s. For the number of packets p, ProfPace generates the 100th percentile of the number of packets and further increments this value by a 10%. The choice of percentiles is determined empirically; the schedules thus generated incur minimal overheads on the peak throughput of web services and moderate overheads on the client response latencies at the cost of a small increase in bandwidth overheads.

Note that the choice of transmit schedules is 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 public and private information, as well as different guest load levels, so that the resulting profiles capture the full space of network traffic shapes.

Summary. Pacer enables tenants to optionally partition their network traffic into public classes, where each class is shaped differently using custom transmission schedules for efficiency. To use custom schedules, a tenant merely has to provide integer-valued traffic indicators at appropriate points in its execution. The indicators enable Pacer to instantiate efficient transmission schedules, and enable ProfPace to automatically generate efficient transmission schedules.

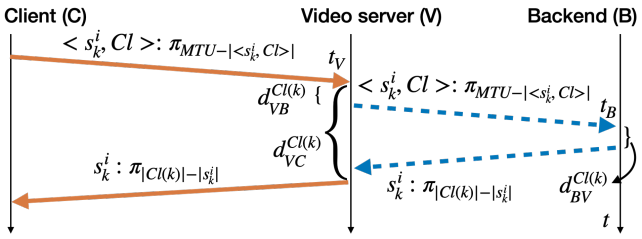

Figure 4: Traffic shaping in video service. s_k^i: k^{th} segment of i^{th} video. Cl: cluster id of the video with segment s_k^i. $|X|$: length of payload X. π_L: padding bytes of length L. t_V, t_B: arrival time of incoming request. $d_{VB}^{Cl(k)}, d_{BV}^{Cl(k)}, d_{VC}^{Cl(k)}$: initial delay for k^{th} segment in cluster Cl. In a **local** setup, shaping is done only on the paths shown by solid arrows.

6 Evaluation

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.

All experiments were performed on Dell PowerEdge R730 servers 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, which were connected to a single 10Gbps switch. The NIC was configured to export SR-IOV vNICs. We disabled hyperthreading, dynamic voltage and frequency scaling, and power management in the hosts, which helps to reduce variance in execution time and ensures consistent, repeatable results across different runs.

We run Xen 4.10.0 hypervisor on each host, which is assigned one of the CPU sockets and 40GB RAM. Up to two cores are configured to execute the HyPace transmit event handler in parallel; flows are partitioned statically among the HyPace cores. The guest runs an Ubuntu 16.04 LTS kernel (version 4.9.5, x86-64) in a VM with 8 cores and 64 GB RAM, and has access to a vNIC. The VCPUs of the guest VM were pinned 1-to-1 to cores on the second socket of the host CPU, and we used Xen's 'Null' scheduler [5] for VM scheduling. This is in line with our threat model, which assumes that guests rent dedicated CPU sockets. Network clients run a modified Ubuntu 16.04 LTS with GPace but no hypervisor.

We used Pacer to demonstrate NSC mitigation in the context of a video streaming service and a medical service. We use Pacer's traffic shaping to hide from an adversary the specific video or medical webpage requested from the respective service, and we evaluate the overhead incurred on client latencies and server throughput in the process. Both the services are hosted using Apache HTTP Server 2.4.33. Below, we first describe the services and then the modifications introduced in various applications to generate traffic indicators.

Video service. We wrote a custom video streaming server in PHP, which returns video segments in response to client requests. We evaluated two setups of the service: (i) **local:** with the videos hosted on the local VM disk, and (ii) **2-tier:** with the videos hosted in a memcached (v1.6.9) KVS backend. In the **2-tier** setup, we used one frontend and two replicated KVSes, each hosted in a VM on a separate server. The frontend randomly selects either KVS replica for serving each video segment. For each segment request and response, shaping is done only between the client and the video server in the **local** setup, while it is done along the entire network path between the client, the frontend and the KVS selected for the segment in the **2-tier setup** (Figure 4).

Medical service. The medical service is a single-tier application based on Mediawiki (v1.27.1) [1]. It stores the content of the medical pages in a database hosted locally on MySQL 5.7.16 and caches a copy of HTML pages generated from the content in a local file cache. In our experiment, all HTML pages are cached. When a client requests a page, Mediawiki queries the database for the page-specific metadata, retrieves the HTML page from the cache and returns it to the client.

Table 1 shows the code locations in the guest applications where we inserted 15 LoC each to generate traffic indicators. We identified and modified these sites manually; automating the instrumentation is possible but remains future work. No other changes were required to guest applications.

Evaluation overview. In the following subsections, we consider (i) microbenchmarks to determine HyPace and GPace configuration parameters; (ii) the tradeoff between spatial padding overhead and privacy possible due to Pacer's clustering, (iii) Pacer's impact on client latencies and server throughput in the context of two guest applications; and (iv) an empirical security evaluation of Pacer's implementation.

6.1 Microbenchmarks

We empirically select the maximum batch size B (number of packets to be prepared by a HyPace handler) in a suitable HyPace epoch length ε, and the parameters δ_{xmit} and δ_{delay} 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. In background, we ran large matrix multiplications on Xen's dom0 VM, which used ~12GB RAM and saturated the CPUs.

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. Epochs of length 30μs, 50μs, 100μs and 120μs could prepare 5, 14, 33 and 42 packets respectively, allowing HyPace to achieve 22%, 28%, 41% and 42% of the NIC line rate with one core. We set ε to 120μs for all HyPace handlers.

Based on these results, we run two parallel HyPace handlers

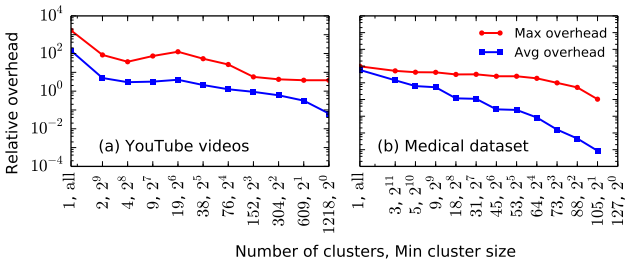

Figure 5: Relative padding overhead vs number of clusters and minimum cluster size for two corpuses representing real-world file size distributions (log-log scale).

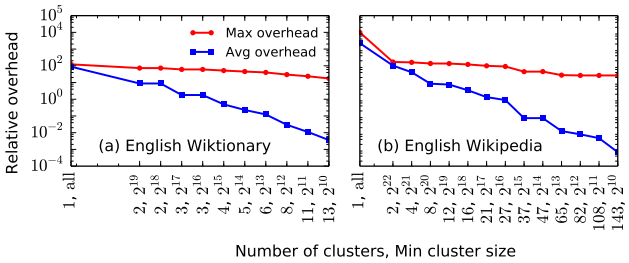

Figure 6: Relative padding overhead vs number of clusters and minimum cluster size (log-log scale) for (a) English Wiktionary and (b) English Wikipedia.

on two separate cores. In this configuration, we repeated our measurements and chose $B = 38$ packets and $\delta_{xmit} = 35\mu$s for each handler. Thus, with two HyPace handlers, Pacer sustains a line rate of 7.67 Gbps, which is 76.7% of the NIC's line rate.

δ_{delay} 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 set δ_{delay} to 20ms.

Note that only δ_{xmit} and δ_{delay} are security-relevant parameters, which we discuss in detail in §B. Epoch and batch size only affect performance.

6.2 Spatial padding overhead

We measure the tradeoff between spatial padding overhead and privacy guarantees when clustering content. The spatial padding overhead corresponds to the network bandwidth overhead for Pacer's traffic shaping.

We clustered two different datasets using algorithms described in §5.3: (i) a set of 1218 videos downloaded from YouTube (240p and 720p bitrate, max duration 4.2hr, median duration 7min, max size 468.7MB, median size 6.2MB), and (ii) a set of 6879 MedicineNet [2] medical web pages comprising diseases, procedures, medications, and supplements pages (max size 521.9KB, median size 75.2KB). Figure 5 shows the reduction in the average and maximum padding overhead with increasing number of clusters and decreasing

Technique	c_{min}	n_1	avg OH	max OH
Power of 2 [13]	1	1	0.512	0.999
Multiple of 100 [14]	1	219	0.001	0.002
Pacer ($c_{min} = 1$)	1	37	0.009	0.027
Pacer ($c_{min} = 8$)	8	0	0.002	0.989
Pacer ($c_{min} = 2206$)	2206	0	1.41	5.17

Table 2: Comparison of privacy and overheads in prior work and Pacer. c_{min}: size of the smallest cluster; n_1: number of clusters with a single element generated by each technique. Pacer's $c_{min} = 1$ is similar to [14] with rounding up to MTU.

minimum cluster size (i.e, the minimum number of objects in each cluster). Compared to the medical dataset, the overhead reduction is less for videos due to the multi-dimensional clustering needed 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.

We also compare Pacer's clustering with other shaping approaches described in the literature. Specifically, CS-BuFLO [13] and Tamaraw [14] round up each response to the nearest power of 2 and a multiple of some integer value (e.g., $L = 100$ in their paper), respectively. As can be seen from Table 2, rounding methods may still leave files with unique sizes in clusters of size 1, rendering the files immediately identifiable. With Pacer's clustering, the overheads are comparable even when generating clusters with more than 2200 files each. We observe similar results with videos. In fact, the rounding methods of prior work lead to nearly all the videos in the corpus being in clusters of size 1. Thus, rounding methods cannot guarantee privacy for all objects in the corpus, while Pacer's clustering can be configured based on desired privacy requirements and bandwidth constraints.

Clustering on larger corpuses. To understand the impact of padding on larger corpuses, we additionally ran our clustering algorithm on two wiki corpuses: (i) a 2016 snapshot of the English Wiktionary corpus (5,027,344 documents, max 521.9KB, median 4.7KB), and (ii) a 2008 snapshot of the English Wikipedia corpus (14,257,494 documents, max 14.3MB, median 83.5KB). Note that though 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 is irrelevant as it is encrypted during transmission anyway. We present the clustering results in Figure 6.

6.3 Macro experiments

Next, we measure the impact of Pacer's traffic shaping on the client response latencies and server throughput in the video service and medical document service. The client request payload is only a few bytes and, hence, is padded to one MTU by the GPace in the client's kernel. Furthermore, the client is open loop, *i.e.*, it transmits requests to the server at fixed

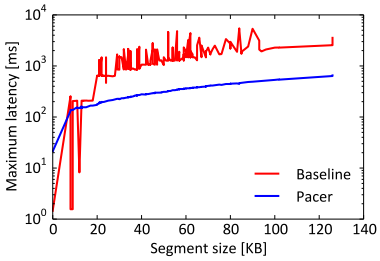

Figure 7: Download latency for a 10Mbps client

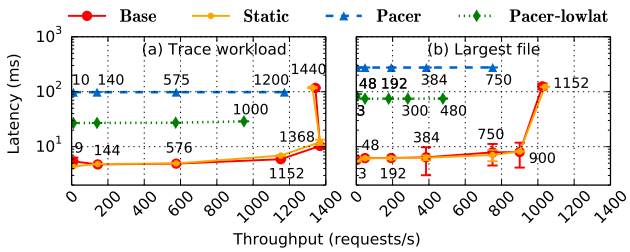

Figure 8: Medical service throughput vs latency

intervals independent of the server's prior responses. With Pacer support on clients (see §2.1), the shaping of client traffic will similarly ensure that request timing does not depend on the completion time of prior requests.

Video 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 5s of video each) in succession to fill 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 whenever space is available in the buffer. We measure the impact of traffic shaping on (i) the download latency for individual video segments, (ii) the initial delay until the video starts playing, and (iii) the frequency and duration of any pauses (video skipping) experienced by the player. We use a corpus of videos 1218 videos downloaded from YouTube in March 2018, which were clustered into 19 clusters with at least 64 elements each, yielding an average padding overhead of 4x. The client sequentially plays four randomly chosen videos for up to 5 min each.

We ran experiments for a client with high bandwidth (10Gbps) and with low bandwidth (10Mbps). The baseline segment download latency is <1ms on average, while the exact latency depends on the segment size. With Pacer, the download latency is dominated by the initial response latency in each segment's traffic shape at the video server, which is 30ms and 400ms in **local** and **2-tier** setup, respectively. Despite these overheads, there is no noticeable impact on the user experience for using Pacer for either client in either setup. Initial startup delays, i.e., the delay until a video starts playing, don't increase significantly, and there is no video skipping in any of the experiments. When serving 128 high bandwidth clients in **2-tier** setup, the maximum CPU utilization on the video server and the KVS increases respectively from 11.76% to 13.39% and 1.96% to 12.62% with Pacer.

Impact on 10Mbps clients. We also evaluated the effect of Pacer's shaping on bandwidth-constrained clients streaming videos. Here, Pacer's shaping also provides an opportunity to use domain knowledge to optimize schedules for better *performance*. Downloading the largest segment in our collection of 240p videos within its 5s deadline requires packets to be sent at an interval of max 20ms. Conservatively increasing the inter-packet spacing in the schedules to even 6ms allows downloading segments within 5s. However, for the 10Mbps clients, the paced schedule avoids losses and reduces the segment download latency significantly. Figure 7 shows the download latency for a 10Mbps client for different segment sizes in the baseline, and after applying Pacer's shaping with 6ms inter-packet spacing. These results are based on the **local** setup (§6). This schedule optimization does not affect security; it only utilizes Pacer to reduce network contention, a known benefit of traffic shaping.

Medical service. Next, we measured Pacer's impact on the throughput and response latency of the medical server. We use a corpus of static HTML pages downloaded from MedicineNet, a medical website [2], in August 2020. We used 3 clusters with at least 2048 elements each, which yields an average padding overhead of 142.8%. Modified wrk2 [4] based clients issue HTTPS GET requests for different pages concurrently and synchronously for 120s. Prior to the measurement, we ran the workload for 10s to warm up caches.

We selected 1,000 files from the 3 clusters in proportion 65%, 30%, and 5%, and used this as a workload trace. Each client requests files from the trace in a random order. For comparison, we also stressed the server with requests only to the largest file in the corpus (521.9KB). Figure 8 shows the throughput vs average latency for two insecure baselines and two schedule configurations of Pacer with varying number of concurrent client requests (denoted by the data point labels). The error bars show the standard deviations of the average latencies. **Base** corresponds to the baseline without any shaping, **Static** corresponds to a baseline where the HTML pages are statically padded but the response traffic is not paced, **Pacer** and **Pacer-lowlat** correspond to Pacer with schedules using the 99th and 80th %ile initial response latency, respectively.

The performance of **Static** is nearly the same as **Base** because the padding added is low for the trace workload and zero for the largest file, implying that there is not much difference in the two workloads. Unlike the baseline, Pacer's latency remains constant until the maximal throughput, because latency is determined only by the transmission schedule. Once the server is at capacity, it fails to serve additional requests and clients time out. With **Pacer-lowlat** a few requests (e.g., less than 50 out of 85K) timeout even at lower loads.

This is because **Pacer-lowlat** uses an aggressive schedule that does not account for the server's response latencies beyond the 80th %ile. We ignore these timeouts in the average latency and throughput measurements.

In the trace workload, **Pacer** incurs a 14.4% overhead on peak throughput with response latencies 10x-18x of the baseline. The figures reflect Pacer's total overhead, because they compare to a saturated baseline server. By comparison, **Pacer-lowlat** incurs a ~30% overhead on peak throughput with latencies 3x-5x of the baseline. Here, the throughput drops as the requests that are delayed beyond the 80th percentile latency timeout. Similar trends are also observed with the large file. This shows that latencies could be optimized with moderate additional overheads on response throughput.

The overheads on peak throughput are higher with the large file. Here, the baseline operates at over 40% of the line rate, and we believe that Pacer's performance in this challenging experiment is limited by the accuracy of transmit schedules, which can be improved substantially.

Pacer's costs are in bandwidth, CPUs, and memory. The bandwidth overhead (§6.2) depends on the application's data, workloads and the public inputs chosen for workload partitioning. The bandwidth overhead due to Pacer's clustering is comparable to that of prior work while offering stronger privacy. The CPU cost is in the two cores dedicated to HyPace (§6.1), and the increase in the guest CPU utilization due to shaping (§6.3). Pacer requires less than 20MB of additional main memory in the Xen hypervisor and less than 30MB of additional memory per HyPace core in each guest that uses Pacer. Cloud providers would likely charge their tenants for the added cost of NSC mitigation. In the case of a service like MedWeb, the tenants (health insurer or provider) would likely cover the cost from their own customers' premiums or subscriptions.

6.4 Security evaluation

Pacer is secure by design, as supported by a formal model and proof in §C. Nevertheless, as a sanity check and to validate our prototype implementation, we also empirically evaluated the security of Pacer's implementation using a powerful NSC attack. We streamed 4 videos from a single video cluster 40 times each, and collected the precise timestamps and sizes of packets transmitted in both directions using tcpdump at the video client. Thus, we grant the attacker direct access to the victim's traffic shape, which makes the attack more powerful than one launched by a colocated tenant. (However, as described in §A, the attack is effective even when launched by a colocated tenant when Pacer is not used.)

For Pacer, we trained a multi-feature CNN classifier using timeseries labeled with the video id and comprising of inter-packet intervals and sizes of packets in both directions between client and the server as the features. For the baseline, we trained the same classifier with just a single feature—the timeseries of inter-packet intervals in server's response packets. The classifier architecture is similar to that used by Schuster *et al.* [58, §7.2], except we used a dropout of 0.1 between the model's hidden layers and 64 epochs for training.

During classification, the classifier generates the probability of each label value for a given sample. In the baseline, the classification probability is more than 99% for each label. In Pacer, it is close to 25%, *i.e.*, the classifier's prediction of a video's label is no better than a random guess. We repeated the experiment with other video clusters and obtained similar results. Thus, we confirm empirically that, as expected, Pacer eliminates leaks through timing, sizes, and count of packets.

7 Related work

We compare to existing mitigation techniques and discuss related work with different threat models or goals.

(a) Mitigating NSCs in Clouds. Contention on individual shared links in a Cloud can be mitigated by time-division multiple access (TDMA) in a hypervisor [8, 34] as this eliminates the adversary VM's (and, in fact, every VM's) ability to observe a colocated victim's traffic at that link. However, an end-to-end mitigation against all network adversaries would require synchronous TDMA scheduling along the entire path of a tenant's traffic, which is inefficient especially when the payload traffic is bursty [68]. Statistical multiplexing, which only caps the total amount of data transmitted by a VM in an epoch, is insecure because the resources available to a flow depend on the bandwidth utilization of other flows [27].

Another approach restricts the adversary VM's ability to observe time [44, 48, 67]. StopWatch [41] replaces a VM's clock with virtual time based only on that VM's execution. To mitigate NSCs, each VM is replicated 3×, the replicas are colocated 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 colocated tenant. However, it requires a 3× increase in deployed Cloud resources. Deterland [76] also replaces VMs' real time with virtual time, but it does not address leaks due to NSCs as it delivers I/O events to VMs without delay. In contrast, Pacer *shapes* traffic by padding and pacing packets, which mitigates all NSCs with far less resource overhead.

Bilal *et al.* [9] generate multicast traffic to shape 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.

(b) Traffic shaping to mitigate NSCs. Pacer uses a standard technique [29, 73] to remove the dependence of packet *size* on secrets: it pads all packets to a fixed length. To make packet *timing* independent of secrets, a strawman is to send packets continuously at a *fixed* rate independent of the actual workload, inserting dummy packets when no actual packets exist [57, 62]. This either wastes bandwidth or incurs high

latencies when the workload is bursty. BuFLO [21] 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, it leaks the size of responses that take longer than the minimum time. Tamaraw [14], CS-BuFLO [13], and DynaFlow [46] 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 contrast, Pacer supports flexible traffic shape adaptation without leaking secrets. Moreover, as shown in §6.2, the bandwidth overhead of Pacer's clustering is comparable to CS-BuFLO's and Tamaraw's.

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. They cluster by simultaneously considering both packet sizes and timing from runtime network traces, and compute the shape based on the traces used in the clusters. Pacer instead first clusters based on static object sizes, and then computes traffic shapes for each cluster based on network traces of cluster objects. Pacer can also support clustering and shaping algorithms proposed by these systems. Traffic morphing [74] makes sensitive responses look like non-sensitive responses, but only shapes packet sizes and ignores packet timing. Pacer shapes all packet size and timing, and allows precise control over cluster sizes, thus eliminating all leaks by design.

(c) Predictive mitigation. Predictive mitigation [7, 80] mitigates network timing side-channel and covert channel leaks to an adversary who has compromised or authenticated as a legitimate client of the victim. Here, the adversary can distinguish real packets from dummies, so predictive mitigation cannot avoid a leak when the application fails to produce a packet in time for a scheduled transmission. In Pacer, the threat is from an adversary that only observes network traffic but does not communicate with the victim. Hence, Pacer can hide application delays by sending dummy packets. Both predictive mitigation and Pacer partition application workloads based on public inputs and precompute a traffic shape for each partition. However, a bad shape leaks information in predictive mitigation, but only affects performance in Pacer.

(d) Related work with other security goals. Herd [40], Vuvuzela [65], Karaoke [38], and Yodel [39] provide metadata privacy: they prevent information about who is communicating with whom from leaking via NSCs. Pacer's goal is different: it prevents sensitive data from leaking via NSC. To address its goal, in addition to shaping individual packet sizes and timing, Pacer shapes the lengths of application messages. Herd [40] and Yodel [39] focus on VoIP calls. Pacer can also be used to shape VoIP traffic. For instance, uniform pacing can be used for a maximum duration, which is picked before the call from a set of allowed durations. Only this maximum duration, but not the actual duration, will be leaked.

Format-Transforming Encryption (FTE) [22] and ScrambleSuit [72] use a tunnel abstraction to modify payload traffic to bypass a traffic censor's filters. However, unlike Pacer, they do not decorrelate the observable traffic shape from secrets. SkypeMorph [49] circumvents censors that inspect packet sizes and timing. It samples the inter-packet gap and the packet size from a fixed distribution, which mimics the distribution of some target protocol that the censor allows. SkypeMorph shapes traffic, but unlike Pacer it is not designed to ensure that the resulting shape does not reveal secret-dependent variations. Moreover, SkypeMorph transmits traffic *continuously* at the average transmission rate of the target protocol, which is inefficient for bursty traffic.

Oblivious computing systems [19, 23, 45] prevent accessed memory *addresses* or accessed database *keys* from depending on secrets, for which they rely on ORAM techniques. Pacer addresses the orthogonal problem of making packet size and timing independent of secrets, and relies on traffic shaping. Fletcher *et al.* [24] address *timing* leaks in ORAM accesses by pacing ORAM accesses. However, their pacing rate changes periodically based on the past actual request rate of the program, which may be secret-dependent and leak information.

(e) Other work. Some prior work [18, 51, 54] use performance-isolation techniques for performance predictability; Silo [33] implements traffic pacing to improve remote access latency; and MITTS [81] "shapes" memory traffic on CPU cores for performance and fairness. The goals and approaches are different from Pacer's. Richter *et al.* [55] propose to performance-isolate colocated tenants by modifying the NIC firmware. Pacer's traffic shaping can be implemented in NIC to provide strong isolation from the rest of the system in the face of microarchitectural side channels (§4.1).

8 Conclusions

Pacer is a comprehensive, provably-secure mitigation for NSC leaks in IaaS Clouds. It reshapes network traffic outside guest VMs to make packet timing and packet sizes independent of guest secrets. Pacer integrates with the host hypervisor to thwart attacks from colocated tenants, relies on paravirtualization to respect network flow control, congestion control, and loss recovery, and uses performance isolation and masking to nullify the effects of internal timing channels within the host. Pacer's end-to-end overheads are moderate.

Acknowledgments

We thank Lorenzo Alvisi, Bobby Bhattacharjee, Keon Jang, Antoine Kaufmann, Jonathan Mace, and the anonymous reviewers for their helpful feedback on earlier versions of this paper. This work was supported in by part by the European Research Council (ERC Synergy imPACT 610150) and the German Science Foundation (DFG CRC 1223).

References

[1] MediaWiki. https://www.mediawiki.org/wiki/MediaWiki_1.27. Accessed 31 Aug 2020.

[2] MedicineNet. https://www.medicinenet.com/script/main/hp.asp. Last accessed on 16 Sep 2020.

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

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

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

[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] 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.

[8] Andrew Beams, Sampath Kannan, and Sebastian Angel. Packet scheduling with optional client privacy. 2021.

[9] 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.

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

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Heyning Cheng and Ron Avnur. Traffic Analysis of SSL Encrypted Web Browsing, 1998.

[18] 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.

[19] 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.

[20] Wladimir De la Cadena, Asya Mitseva, Jens Hiller, Jan Pennekamp, Sebastian Reuter, Julian Filter, Thomas Engel, Klaus Wehrle, and Andriy Panchenko. TrafficSliver: Fighting Website Fingerprinting Attacks with Traffic Splitting. In *ACM Conf. on Computer and Communications Security (CCS)*, 2020.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] Xun Gong, Nikita Borisov, Negar Kiyavash, and Nabil Schear. Website Detection Using Remote Traffic Analysis. In *Privacy Enhancing Technologies Symposium (PETS)*, 2012.

[27] 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.

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

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

[30] 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.

[31] 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.

[32] 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.

[33] 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.

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

[35] Diederik P Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. http://arxiv.org/abs/1412.6980, 2014.

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

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] Xiapu Luo, Peng Zhou, Edmond WW Chan, Wenke Lee, Rocky KC Chang, and Roberto Perdisci. HTTPOS: Sealing Information Leaks with Browser-side Obfuscation of Encrypted Flows. In *Network and Distributed System Security Symposium (NDSS)*, volume 11, 2011.

[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] 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.

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

[54] 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.

[55] 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.

[56] 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.

[57] 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.

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

[59] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. *CoRR*, abs/1807.10535, 2018.

[60] Shawn Shan, Arjun Nitin Bhagoji, Haitao Zheng, and Ben Y Zhao. A Real-time Defense against Website Fingerprinting Attacks. *arXiv preprint arXiv:2102.04291*, 2021.

[61] Geoffrey Smith. Principles of Secure Information Flow Analysis. In Mihai Christodorescu, Somesh Jha, Douglas Maughan, Dawn Song, and Cliff Wang, editors, *Malware Detection*, volume 27 of *Advances in Information Security*, pages 291–307. Springer, 2007.

[62] Dawn Xiaodong Song, David Wagner, and Xuqing Tian. Timing Analysis of Keystrokes and Timing Attacks on SSH. In *USENIX Security Symposium*, 2001.

[63] 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.

[64] Paul Syverson, Roger Dingledine, and Nick Mathewson. Tor: The Second-Generation Onion Router. In *Usenix Security*, 2004.

[65] 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.

[66] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael M Swift. Scheduler-based Defenses against Cross-VM Side-channels. In *USENIX Security Symposium*, 2014.

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

[68] Bhanu Chandra Vattikonda, George Porter, Amin Vahdat, and Alex C Snoeren. Practical TDMA for Datacenter Ethernet. In *ACM European Conference on Computer Systems (EuroSys)*, 2012.

[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] Charles V Wright, Fabian Monrose, and Gerald M Masson. On Inferring Application Protocol Behaviors in Encrypted Network Traffic. *Journal of Machine Learning Research (JMLR)*, 7, Dec 2006.

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

[77] 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.

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

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

[80] 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.

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

A Network Side-Channel Attack

Here, we briefly describe a proof-of-concept NSC attack. To carry out such an attack, an adversary must be able to observe a victim's network traffic. An adversary with access to network elements like links, switches, or routers can observe the traffic *directly*. An adversary without direct access can still observe victim traffic *indirectly* if they can control attack traffic that shares bandwidth with the victim's traffic.

Indirect observation is impossible if *each network flow has exclusively reserved bandwidth*, as in time-division multiple access (TDMA), which ensures non-interference among flows. However, this approach prevents statistical multiplexing and is very inefficient for bursty traffic. On the other hand, when bandwidth is shared, then regardless of the queuing discipline, available bandwidth and queuing delays observed by one flow are influenced by concurrent flows. We demonstrate a simple attack where an adversary exploits the signals in the queueing delays for its own traffic to infer the victim's traffic shape.

Experimental setup. We set up two VMs, a victim and an attack VM, on two separate sockets of a Dell PowerEdge R730 server machine (S_1). The VMs use Xen's virtualized network stack; thus all traffic is routed through the netback driver and the TCP stack in dom0 of the hypervisor. We configure S_1's shared NIC with a bandwidth of 1Gbps, and the hierarchical token bucket (HTB) queueing discipline. We further create two separate HTB traffic classes for (i) the attack traffic, and (ii) the victim traffic and rest of the traffic through the host. We configure the attack traffic to have a lower priority than all other traffic. This is a reasonable assumption as an attacker can always lower the priority for its traffic.

The victim hosts a custom video streaming service on top of Apache, which servers video segment files in response to client requests. A custom video client runs on a second server (S_2) and requests the video segments sequentially over HTTPS. The attack VM runs a UDP client that sends short payloads (56 bytes) to a UDP server on a third machine (S_3), which logs the packet arrival timestamps and echoes the packets back to the attack client. S_2 and S_3 have 10Gbps NICs and all machines are connected with a 10Gbps switch; thus the bottleneck link is the shared NIC at S_1. The attack client maintains a send window of 4500 packets (computed based on the bandwidth-delay product for the NIC), which ensures that some attack packets are always queued at the bottleneck link without overflowing the queue.

We streamed 10 videos at 720p resolution from a YouTube dataset (a detailed description of the dataset is given in §6.2) for up to 30 segments. Segments take less than 0.02s to download, and segments within a video are requested at an interval of 5s. We streamed each video 150 times. During each video stream, we log the series of arrival timestamps of the adversarial client's packets at the adversarial server. We label each time series of the adversary's packet arrival timestamps with the id of the video streamed by the victim. Thus, we have 1500 time series of adversary's packet arrival timestamps with 10 distinct labels.

Analysis. We aggregated each time series into windows of 50ms, and generated a time series of the adversary's transmitted packet count in each window. The packet count is the number of packet arrival timestamps recorded in each time window. Finally, we normalized each packet count time series using min-max normalization.

Next, we implemented a CNN classifier to train on the time series of normalized packet counts. Figure 9 shows the architecture of our classifier, which consists of three convolution layers, a max pooling layer, and two dense layers. We use a dropout of 0.2 between each pair of hidden layers of the classifier as shown in the figure. We train the classifier with an

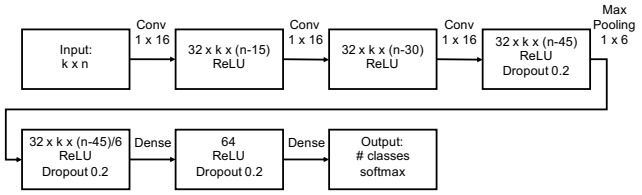

Figure 9: CNN architecture. k: the number of features used. n: the number of elements of one time series, which is the total time of the time series divided by the window size (50ms).

Adam optimizer [35], categorical cross-entropy error function, and with input batches of 64 samples. Our CNN classifier is similar to the one built by Schuster *et al.* [58, section 7.2], with the difference that we used a dropout of 0.2 between the model's hidden layers and 64 epochs for training.

We implemented the classifier using Tensorflow 2 API and with the Keras frontend. We used 70% of the time series data for each label (video) for training and the remaining for evaluating the classifier. The classifier achieves an overall precision and recall of 81.8% each, and an accuracy of 96.4%.

Additional attack setups. We performed a similar, but simpler attack on two additional setups: (i) with the host, S_1's NIC configured in SR-IOV, exposing vNICs to the victim and attacker VMs, and (ii) in a commercial IaaS Cloud provider platform. In both cases, we were able to show that victim transfers of large files generate a large signal on an attacker's cross-traffic that is visible in a timeseries plot even to the naked eye. These attacks are not surprising, since any queueing policy that allows a tenant to use network bandwidth not currently used by other tenants that share a link permits NSCs.

Our experiment confirms prior work [6,11,12,29,52,58,62,69,73] and shows that a network side-channel attack can identify videos in a collection with good accuracy. While an attack in a production environment faces additional challenges like achieving colocation with the victim, prior work has shown that it is easy to attain colocation [30,31,56]. Hence, cloud tenants that require strong confidentiality have to consider that NSCs are a realistic threat.

B Security of masking mechanisms

Recall from §4.1 that Pacer relies on four parameters whose values are empirically determined: the epoch length, the packet transmission batch size, HyPace's interrupt handler masking delay (δ_{xmit}), and GPace's inbound packet- and timer-processing masking delay (δ_{delay}). Of these, only the last two parameters are security-relevant. In this section, we discuss experiments to demonstrate that (i) masking is necessary (without it, the actual runtime of the handler tasks are observable, which could be correlated with guest secrets), and (ii) our empirically computed thresholds are effective at masking these timing leaks. We do this by analyzing Pacer's handlers

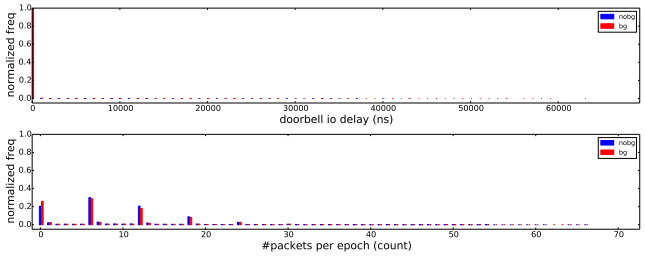

Figure 10: HyPace delays and batch size without masking.

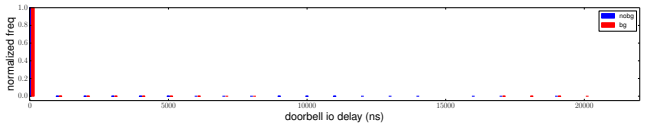

Figure 11: HyPace delays after applying masking.

under two extreme configurations: no background workload (**nobg**) and with *heavy* background load (**bg**).

To demonstrate that masking is necessary, we measure if there is any difference in HyPace execution due to background load in the *absence of masking*. Figure 10 top and bottom plots respectively show the distributions of delays in HyPace's doorbell writes (from the time of the scheduled interrupt handler) and the number of packets that can be prepared within an 120μs epoch in the two configurations. Both the distributions are based on 24-hour experiments, with about 550 million epochs involving some packet transmissions. First of all, in absence of masking, the exact HyPace delay and batch size of every single epoch is observable. Each pair of delay and batch size may be correlated with specific secrets; thus the observations could leak the secrets. Additionally, the distributions of delays and batch sizes are influenced by the background workload. As the figure shows, the maximum HyPace delay observed (top plot) is 65μs and 44μs in **bg** and **nobg** configurations, respectively and the maximum batch size (bottom plot) is 66 and 65 packets, respectively. These observations show that the adversary can potentially affect HyPace timing in the absence of masking to induce leaks. Hence, masking these delays is essential for security.

Next, we repeat the experiment with masking enabled. Figure 11 shows the observed delays of HyPace's doorbell writes when masking with δ_{xmit} set to 35μs. As can be seen, all handler execution times were masked in these experiments.

For full disclosure, in earlier unrelated experiments, we had observed a small number of epochs (e.g., less than 20 out of 550 million) where HyPace's delays exceeded δ_{xmit} by up to 5μs. Such overruns, if they occurred in practice, would be mitigated quickly by the automatic adjustment of δ_{xmit} (§4.1).

To summarize, masking is necessary and effective. In the unlikely case of HyPace delays exceeding our δ_{xmit} of 35μs, the adversary would be able to observe execution times that may be correlated with victim secrets. However, Pacer's automatic adjustment of δ_{xmit} denies the adversary an opportunity

to repeat an observation. Combined with the adversary's challenge to induce increasing delays in the execution of the privileged handlers, it seems impossible for an adversary to cause a sufficient number of repeated overruns necessary to infer a victim's secrets.

Our empirical observations and security arguments for δ_{delay} are similar to the above, and we omit those details.

C Formal Model and Proof of Security

We build a formal model to prove formally that HyPace is secure. In particular, it makes packet timing independent of any application secrets. The model assumes that the masking delays are never exceeded.

Our model has the following actors:

[-] Guest. The guest VM using HyPace's services. Denoted G. G is modeled as a state machine with an internal state σ_{G} that may contain secrets. Our goal is to keep these secrets confidential from the other actors. G's state machine reacts to incoming network events and, in turn, generates events to which HyPace reacts.

[-] HyPace. Denoted H. HyPace is modeled as a state machine with internal state σ_{H}. H's state machine reacts to events generated by the guest G and produces outgoing network events to which the environment reacts.

[-] The environment, which comprises everything outside the server, including the network and the clients. Denoted E. E is modeled as a state machine with internal state σ_{E}. This state machine reacts to HyPace's outgoing network events and produces incoming network events to which the guest reacts.

For simplicity of exposition, we assume that there is a fixed set of N flow names, $\texttt{flow}_1, \ldots, \texttt{flow}_N$. Not all of these may be active at any time, but we assume that HyPace always has a profile for each of them. When a flow is not active, its (default) profile causes no packets to be sent. We use f and its decorated variants as meta-variables that range over $\texttt{flow}_1, \ldots, \texttt{flow}_N$.

Event queues The three causal interactions $\mathsf{E} \to \mathsf{G}$, $\mathsf{G} \to \mathsf{H}$, and $\mathsf{H} \to \mathsf{E}$ are mediated by three event queues (producer-consumer queues), written Q_{G}, Q_{H} and Q_{E}, respectively. A queue contains *timestamped pending events* that have been generated by the queue's producer (E state machine for Q_{G}), but not yet handled by the queue's consumer (G for Q_{G}). The subscript on a queue indicates its consumer. We describe these queues next.

Q_{G}: This queue is a set of tuples of the form $(\mathsf{Ti}_i, \mathsf{Ei}_i, \mathsf{f}_i)$ meaning that incoming network event Ei_i that occurred at time Ti_i on flow f_i is still pending for the guest. Incoming network events represent incoming packets (including new client requests), network congestion signals and indicators of packet loss (e.g., network stack timeouts). The exact structure of these events is irrelevant here and, hence, kept abstract.

$$\mathsf{Q}_{\mathsf{G}} \quad ::= \quad (\mathsf{Ti}_1, \mathsf{Ei}_1, \mathsf{f}_1), \ldots, (\mathsf{Ti}_n, \mathsf{Ei}_n, \mathsf{f}_n)$$

Q_{H}: This queue actually consists of two subqueues – the *profile update subqueue*, written $\mathsf{Q}^u_{\mathsf{H}}$, and the *packet subqueue*, written $\mathsf{Q}^p_{\mathsf{H}}$ – and a time value $\mathsf{Te}_{\max}$.

$$\mathsf{Q}_{\mathsf{H}} \quad ::= \quad (\mathsf{Q}^u_{\mathsf{H}}, \mathsf{Te}_{\max}, \mathsf{Q}^p_{\mathsf{H}})$$

$\mathsf{Q}^u_{\mathsf{H}}$ contains profile updates. It is actually a key-value store, keyed by flows. For every flow $\texttt{flow}_i$ ($i \in \{1 \ldots N\}$), the value is a set U_i of pending updates on that flow.

$$\begin{array}{lcl} \mathsf{U} & ::= & (\mathsf{Tu}_1, \mathsf{Eu}_1, \mathsf{Te}_1), \ldots, (\mathsf{Tu}_m, \mathsf{Eu}_m, \mathsf{Te}_m) \\ & & \text{-- sorted ascending by } \mathsf{Te}_i \\ \mathsf{Q}^u_{\mathsf{H}} & ::= & \texttt{flow}_1 \mapsto \mathsf{U}_1, \ldots, \texttt{flow}_N \mapsto \mathsf{U}_N \end{array}$$

U contains update tuples of the form $(\mathsf{Tu}_i, \mathsf{Eu}_i, \mathsf{Te}_i)$ meaning that the profile update described by Eu_i was queued by the guest at time Tu_i, but *should be effective at time* Te_i. Eu_i may replace the existing profile at the application's request, or update/start the profile in response to an incoming network event (e.g., starting a new profile in response to a new request, pausing or resuming a profile in response to congestion signals, or extending a profile in response to packet retransmissions).

It is assumed that the guest chooses Te_i sufficiently after Tu_i to allow the event to propagate to HyPace despite any processing delays (we explicitly model this assumption later). It is essential that Te_i *be independent of secrets*. For an update in response to an incoming network event, Te_i can be set to the timestamp of the incoming network event plus the maximum (empirical) propagation delay of the network stack. For a profile update initiated by the application, the application can set Te_i to the end time of the current profile minus the propagation delay of the application and network stack. If the propagation delay of the application is secret-dependent, it can be bucketized into public buckets and the next higher bucket boundary can be used instead.

Note that U is an ordered list, not a set. It is sorted in increasing order of Te_i. HyPace also applies updates in this order. We often treat $\mathsf{Q}^u_{\mathsf{H}}$ as an indexed vector, writing $\mathsf{Q}^u_{\mathsf{H}}[\mathsf{f}]$ for the update tuples of flow f.

Q_{H} contains a timestamp $\mathsf{Te}_{\max}$, which is the highest effective time of any update event that has been added to $\mathsf{Q}^u_{\mathsf{H}}$ in the past. This means that if $(\mathsf{Tu}, \mathsf{Eu}, \mathsf{Te}) \in \mathsf{U}_i$, then $\mathsf{Te} \leq \mathsf{Te}_{\max}$.

The packet subqueue $\mathsf{Q}^p_{\mathsf{H}}$ contains network packets queued by the guest for transmission. These packets should be encrypted. The details of this queue are irrelevant for our model, so we leave it abstract.

Q_{E}: This queue is a set of tuples of the form $(\mathsf{To}_i, \mathsf{Eo}_i, \mathsf{f}_i)$ meaning that the outgoing network packet Eo_i is generated on flow f_i by HyPace at time To_i. The Eo_i represents the encrypted payload, so its structure is irrelevant.

$$\mathsf{Q}_{\mathsf{E}} \quad ::= \quad (\mathsf{To}_1, \mathsf{Eo}_1, \mathsf{f}_1), \ldots, (\mathsf{To}_n, \mathsf{Eo}_n, \mathsf{f}_n)$$

State Next, we describe the internal states of the environment, guest and HyPace:

σ_E: The environment's state σ_E is a pair of a private component and a remaining component, written $\sigma\mathsf{pri}_E$ and $\sigma\mathsf{pub}_E$, respectively. The private component $\sigma\mathsf{pri}_E$ is held only by the clients of the guest VM being protected, while $\sigma\mathsf{pub}_E$ represents the remaining state of the clients, the network and any other actors. We do not specify these any further, but there are constraints on how they evolve.

$$\sigma_E \quad ::= \quad (\sigma\mathsf{pri}_E, \sigma\mathsf{pub}_E)$$

σ_G: The guest's state σ_G is similarly a pair of a private component and a remaining component, written $\sigma\mathsf{pri}_G$ and $\sigma\mathsf{pub}_G$, respectively. We do not specify these any further, but there are constraints on how they evolve.

$$\sigma_G \quad ::= \quad (\sigma\mathsf{pri}_G, \sigma\mathsf{pub}_G)$$

σ_H: HyPace's state σ_H consists of a map from flows to the current active profiles on them. We use the notation Φ for a profile. The exact structure of profiles is irrelevant for the security argument, so we keep it abstract.

$$\sigma_H \quad ::= \quad \mathtt{flow}_1 \mapsto \Phi_1, \ldots, \mathtt{flow}_N \mapsto \Phi_N$$

Auxiliary state There is also some auxiliary state that is not associated to any specific component. This state is just the current (global) time, written Tg.

$$\mathsf{Tg} \quad ::= \quad \text{Current global time}$$

Overall state (Configuration) The overall state of the system, also called a *configuration* and denoted $\mathcal{C}$, consists of the internal states of HyPace, the guest and the environment, the three event queues, and the auxiliary state.

$$\mathcal{C} \quad ::= \quad (\sigma_E, \sigma_G, \sigma_H, Q_G, Q_H, Q_E, \mathsf{Tg})$$

C.1 System evolution

The overall state (configuration) evolves over time through transitions. We write $\mathcal{C} \rightsquigarrow \mathcal{C}'$ to say that the configuration $\mathcal{C}$ transitions to $\mathcal{C}'$ in a single step.

A transition happens when one of the agents acts on events pending for it. Without loss of generality, we assume that only HyPace's actions cause the global time Tg to jump forward, and this jump is exactly the length of one epoch, which we denote δ_e.

$$\delta_e \quad ::= \quad \text{Length of epoch}$$

Also w.l.o.g., the agents act in order: HyPace, environment, guest, and repeat. Technically,

$$\rightsquigarrow \;\triangleq\; \rightsquigarrow_E \;;\; \rightsquigarrow_G \;;\; \rightsquigarrow_H$$

$$\begin{array}{lcl}
Q_E| & \triangleq & \{(\mathsf{To}, f) \mid (\mathsf{To}, \mathsf{Eo}, f) \in Q_E\} \\
Q_{E1} \sim Q_{E2} & \triangleq & Q_{E1}| = Q_{E2}| \\
\\
Q_G| & \triangleq & \{(\mathsf{Ti}, f) \mid (\mathsf{Ti}, \mathsf{Ei}, f) \in Q_G\} \\
Q_{G1} \sim Q_{G2} & \triangleq & Q_{G1}| = Q_{G2}| \\
\\
U| & \triangleq & [(\mathsf{Eu}_i, \mathsf{Te}_i) \mid (\mathsf{Tu}_i, \mathsf{Eu}_i, \mathsf{Te}_i) \in U] \\
Q^u_H| & \triangleq & \{(f \mapsto U|) \mid (f \mapsto U) \in Q^u_H\} \\
Q^u_{H1} \sim Q^u_{H2} & \triangleq & Q^u_{H1}| = Q^u_{H2}| \\
\\
\sigma_E| & = & \sigma\mathsf{pub}_E \text{ for } \sigma_E = (\sigma\mathsf{pri}_E, \sigma\mathsf{pub}_E) \\
\sigma_{E1} \sim \sigma_{E2} & \triangleq & \sigma_{E1}| = \sigma_{E2}| \\
\\
\sigma_G| & = & \sigma\mathsf{pub}_G \text{ for } \sigma_G = (\sigma\mathsf{pri}_G, \sigma\mathsf{pub}_G) \\
\sigma_{G1} \sim \sigma_{G2} & \triangleq & \sigma_{G1}| = \sigma_{G2}|
\end{array}$$

Figure 12: Equivalence of states and queues

where $\rightsquigarrow_E$, $\rightsquigarrow_G$ and $\rightsquigarrow_H$ represent steps of the environment, guest and HyPace, respectively, and the semicolon means relation composition. In the following we describe $\rightsquigarrow_E$, $\rightsquigarrow_G$ and $\rightsquigarrow_H$ one by one.

C.1.1 Environment acts

The environment acts by consuming a subset of events in the queue Q_E, processing them to update its internal state and adding new events to the queue Q_G. We model the environment as an abstract function F_E that takes as input the current queue Q_E, the environment's internal state σ_E (which contains the private component $\sigma\mathsf{pri}_E$ and the remaining component $\sigma\mathsf{pub}_E$) and the current global time Tg. It outputs a new internal state $\sigma_E{}'$, an updated queue $Q_E{}'$ (which should be a subset of Q_E) and a set of events $Q_G{}''$ that are added to Q_G by $\rightsquigarrow$.

$$F_E(Q_E, \sigma_E, \mathsf{Tg}) = (Q_E{}', Q_G{}'', \sigma_E{}')$$

Using F_E, we define the transition rule (env) for the environment (Figure 13). Note the index E in $\rightsquigarrow_E$, which indicates that this is the environment's transition.[8]

The function F_E can be arbitrary (we don't assume that we know what the network and the clients do), but it is subject to an important security assumption, which we describe here.

(Clients don't break secrecy explicitly) Clients get to see the payloads of incoming messages, which may depend on secrets and also have access to their existing private state $\sigma\mathsf{pri}_E$. In our threat model, we explicitly trust clients to not leak either of these into timing. In the formal model, this is specified by a constraint on F_E. Specifically, if we consider two input

[8] The way to read a rule $\frac{A}{B}$ is that *if* A holds *then* B holds.

Assumptions

(1) $\mathsf{Q_{E1}} \sim \mathsf{Q_{E2}}$ and $\sigma_{E1} = \sigma_{E2}$ and
$\mathsf{F_E}(\mathsf{Q_{E1}}, \sigma_{E1}, \mathsf{Tg}) = (\mathsf{Q_{E1}}', \mathsf{Q_{G1}}'', \sigma_{E1}')$ and
$\mathsf{F_E}(\mathsf{Q_{E2}}, \sigma_{E2}, \mathsf{Tg}) = (\mathsf{Q_{E2}}', \mathsf{Q_{G2}}'', \sigma_{E2}')$
$\Rightarrow$
$\mathsf{Q_{E1}}' \sim \mathsf{Q_{E2}}'$ and
$\mathsf{Q_{G1}}'' \sim \mathsf{Q_{G2}}''$ and
$\sigma_{E1}' \sim \sigma_{E2}'$

Transition

$$\frac{\mathsf{F_E}(\mathsf{Q_E}, \sigma_E, \mathsf{Tg}) = (\mathsf{Q_E}', \mathsf{Q_G}'', \sigma_E')}{(\sigma_E, \sigma_G, \sigma_H, \mathsf{Q_G}, \mathsf{Q_H}, \mathsf{Q_E}, \mathsf{Tg}) \rightsquigarrow_E (\sigma_E', \sigma_G, \sigma_H, \mathsf{Q_G} \cup \mathsf{Q_G}'', \mathsf{Q_H}, \mathsf{Q_E}', \mathsf{Tg})}\text{env}$$

Figure 13: Assumptions and transition of the environment

queues $\mathsf{Q_{E1}}$ and $\mathsf{Q_{E2}}$ that differ only in payloads (but agree on timing), and two states σ_{E1} and σ_{E2} that differ only in the private components $\sigma\mathsf{pri}_{E1}$ and $\sigma\mathsf{pri}_{E2}$ (but agree in the non-private components), then the output queues $\mathsf{Q_{E1}}'$ and $\mathsf{Q_{E2}}'$ should differ only in the payloads (similarly for $\mathsf{Q_{G1}}''$ and $\mathsf{Q_{G2}}''$), and the output states σ_{E1} and σ_{E2} can differ only in the private components.

Formally, we define $\mathsf{Q_{E1}} \sim \mathsf{Q_{E2}}$ to mean that $\mathsf{Q_{E1}}$ and $\mathsf{Q_{E2}}$ agree on the flows and timestamps of events. Similarly, we define $\mathsf{Q_{G1}} \sim \mathsf{Q_{G2}}$. Finally, we define $\sigma_{E1} \sim \sigma_{E2}$ to mean that σ_{E1} and σ_{E2} agree on the non-private components. These definitions are shown in Figure 12. We then make the assumption (1) in Figure 13, which captures exactly the intuition described in the previous paragraph.

Note. A real $\mathsf{F_E}$ would also have the following properties, but we do not need these properties for security, so we do *not* assume them. We show these properties just for completeness.

(Causality) $\mathsf{F_E}$ should depend only on past events in $\mathsf{Q_E}$, i.e., those that occurred before the current time Tg. Formally, we assume that

$$\mathsf{F_E}(\mathsf{Q_E}, \sigma_E, \mathsf{Tg}) = \mathsf{F_E}(\mathsf{Q_E}|_{\leq \mathsf{Tg}}, \sigma_E, \mathsf{Tg})$$

Here, $\mathsf{Q_E}|_{\leq \mathsf{Tg}}$ denotes the subset of $\mathsf{Q_E}$ containing events whose timestamps are no more than Tg.

$$\mathsf{Q_E}|_{\leq \mathsf{Tg}} \triangleq \{(\mathsf{To}, \mathsf{Eo}, \mathsf{f}) \mid (\mathsf{To}, \mathsf{Eo}, \mathsf{f}) \in \mathsf{Q_E} \text{ and } \mathsf{To} \leq \mathsf{Tg}\}$$

(Non-modification of past outputs) $\mathsf{F_E}$ should not output events in the past, i.e., $\mathsf{Q_G}''$ should not contain any events with timestamps Tg or lower.

$$\forall (\mathsf{Ti}, \mathsf{Ei}, \mathsf{f}) \in \mathsf{Q_G}''.\ \mathsf{Ti} > \mathsf{Tg}$$

(Non-consumption of future inputs) $\mathsf{F_E}$ should not consume input events from the future, i.e., $\mathsf{Q_E}'$ should agree with $\mathsf{Q_E}$ on *future* events, i.e.,

$$\mathsf{Q_E}|_{>\mathsf{Tg}} = \mathsf{Q_E}'|_{>\mathsf{Tg}}$$

Here, $\mathsf{Q_E}|_{>\mathsf{Tg}}$ denotes the subset of $\mathsf{Q_E}$ containing events whose timestamps are strictly greater than Tg. It is defined analogous to $\mathsf{Q_E}|_{\leq \mathsf{Tg}}$.

C.1.2 Guest acts

The guest acts by consuming events from $\mathsf{Q_G}$ to update its internal state and to produce events in the queue $\mathsf{Q_H}$ (including both its subqueues $\mathsf{Q_H^u}$ and $\mathsf{Q_H^p}$). We model the environment as an abstract function $\mathsf{F_G}$.

$$\mathsf{F_G}(\mathsf{Q_G}, \mathsf{Te}_{\max}, \mathsf{Q_H^p}, \sigma_G, \mathsf{Tg}) = (\mathsf{Q_G}', \mathsf{Q_H^u}'', \mathsf{Te}_{\max}', \mathsf{Q_H^p}', \sigma_G')$$

$\mathsf{F_G}$ takes as input the incoming network event queue $\mathsf{Q_G}$ (recall that this queue is populated by the environment), the current maximum update effective time $\mathsf{Te}_{\max}$, the current packet queue $\mathsf{Q_H^p}$, the guest's current state σ_G and the current time Tg. It outputs an updated input queue $\mathsf{Q_G}'$, a set of profile update key-values $\mathsf{Q_H^u}''$ to add to the hypervisor's update queue, a new $\mathsf{Te}_{\max}'$, an updated packet queue $\mathsf{Q_H^u}'$, and a new guest state σ_G'.

Using $\mathsf{F_G}$, we define the transition rule (guest) for the guest (Figure 14). The index G in $\rightsquigarrow_G$ indicates that this is the guest's transition.

The function $\mathsf{F_G}$ can be arbitrary (meaning that the enforcement is almost black-box), but it is subject to some causality and security assumptions.

(Guest does not break secrecy explicitly) The guest should distinguish private from public state, but it may have timing leaks. Specifically, the descriptions of profile updates (denoted Eu_i) it queues for the hypervisor must not depend on the guest's secret state or the payloads of incoming packets, which may also be secret-dependent. Similarly, the times at which these updates become effective (determined by Te_i) should be secret-independent. However, the time at which the update events are queued (denoted Tu_i) may depend on secrets due to timing leaks. Also, the packets the guest queues to send (i.e., the subqueue $\mathsf{Q_H^p}$) may depend on secrets. These packets are encrypted anyhow.

To formalize this, we define notions of equivalence $\sim$ of the guest state σ_G (the public components, but not the private components, must coincide) and the update event queue $\mathsf{Q_H^u}$ (Figure 12). We then assume (2) from Figure 14, which formalizes the intuition of the previous paragraph. Observe that $\mathsf{Q_{H1}^u}'' \sim \mathsf{Q_{H2}^u}''$ in (2) correctly imposes no restrictions on Tu_is as these may depend on secrets. Additionally, (2) imposes no restrictions at all on the packet queues ($\mathsf{Q_{H1}^p}'$ and $\mathsf{Q_{H2}^p}'$), as these queues may also be secret-dependent.

Assumptions

(2) $\mathsf{Q_{G1}} \sim \mathsf{Q_{G2}}$ and $\sigma_{\mathsf{G1}} \sim \sigma_{\mathsf{G2}}$ and
$\mathsf{F_G}(\mathsf{Q_{G1}}, \mathsf{Te_{max1}}, \mathsf{Q}^p_{\mathsf{H1}}, \sigma_{\mathsf{G1}}, \mathsf{Tg}) =$
$(\mathsf{Q_{G1}}', \mathsf{Q}^{u\,\prime\prime}_{\mathsf{H1}}, \mathsf{Te_{max1}}', \mathsf{Q}^{p\,\prime}_{\mathsf{H1}}, \sigma_{\mathsf{G1}}')$ and
$\mathsf{F_G}(\mathsf{Q_{G2}}, \mathsf{Te_{max2}}, \mathsf{Q}^p_{\mathsf{H2}}, \sigma_{\mathsf{G2}}, \mathsf{Tg}) =$
$(\mathsf{Q_{G2}}', \mathsf{Q}^{u\,\prime\prime}_{\mathsf{H2}}, \mathsf{Te_{max2}}', \mathsf{Q}^{p\,\prime}_{\mathsf{H2}}, \sigma_{\mathsf{G2}}')$
$\Rightarrow$
$\mathsf{Q_{G1}}' \sim \mathsf{Q_{G2}}'$ and
$\mathsf{Q}^{u\,\prime\prime}_{\mathsf{H1}} \sim \mathsf{Q}^{u\,\prime\prime}_{\mathsf{H2}}$ and
$\sigma_{\mathsf{G1}}' \sim \sigma_{\mathsf{G2}}'$

(3) $\mathsf{F_G}(\mathsf{Q_G}, \mathsf{Te_{max}}, \mathsf{Q}^p_{\mathsf{H}}, \sigma_{\mathsf{G}}, \mathsf{Tg}) =$
$(\mathsf{Q_G}', \mathsf{Q}^{u\,\prime}_{\mathsf{H}}, \mathsf{Te_{max}}', \mathsf{Q}^{p\,\prime}_{\mathsf{H}}, \sigma_{\mathsf{G}}')$
$\Rightarrow I_{\mathsf{delay}}(\mathsf{Q}^{u\,\prime}_{\mathsf{H}})$
where
$I_{\mathsf{delay}}(\mathsf{Q}^u_{\mathsf{H}}) \triangleq \forall(\mathsf{f} \mapsto \mathsf{U}) \in \mathsf{Q}^u_{\mathsf{H}}.\ \forall(\mathsf{Tu}_i, \mathsf{Eu}_i, \mathsf{Te}_i) \in \mathsf{U}.\ \mathsf{Tu}_i \leq \mathsf{Te}_i$

(4) $\mathsf{F_G}(\mathsf{Q_G}, \mathsf{Te_{max}}, \mathsf{Q}^p_{\mathsf{H}}, \sigma_{\mathsf{G}}, \mathsf{Tg}) =$
$(\mathsf{Q_G}', \mathsf{Q}^{u\,\prime\prime}_{\mathsf{H}}, \mathsf{Te_{max}}', \mathsf{Q}^{p\,\prime}_{\mathsf{H}}, \sigma_{\mathsf{G}}')$
$\Rightarrow I_{\mathsf{emax}}(\mathsf{Q}^{u\,\prime\prime}_{\mathsf{H}}, \mathsf{Te_{max}}')$
where
$I_{\mathsf{emax}}(\mathsf{Q}^u_{\mathsf{H}}, \mathsf{Te_{max}}) \triangleq \forall(\mathsf{f} \mapsto \mathsf{U}) \in \mathsf{Q}^u_{\mathsf{H}}.\ \forall(\mathsf{Tu}_i, \mathsf{Eu}_i, \mathsf{Te}_i) \in \mathsf{U}.\ \mathsf{Te}_i \leq \mathsf{Te_{max}}$

(5) $\mathsf{F_G}(\mathsf{Q_G}, \mathsf{Te_{max}}, \mathsf{Q}^p_{\mathsf{H}}, \sigma_{\mathsf{G}}, \mathsf{Tg}) =$
$(\mathsf{Q_G}', \mathsf{Q}^{u\,\prime\prime}_{\mathsf{H}}, \mathsf{Te_{max}}', \mathsf{Q}^{p\,\prime}_{\mathsf{H}}, \sigma_{\mathsf{G}}')$
$\Rightarrow \forall(\mathsf{f} \mapsto \mathsf{U}') \in \mathsf{Q}^{u\,\prime\prime}_{\mathsf{H}}.\ \forall(\mathsf{Tu}'_i, \mathsf{Eu}'_i, \mathsf{Te}'_i) \in \mathsf{U}'.\ \mathsf{Te_{max}} < \mathsf{Te}'_i$

Transition

$$\frac{\begin{array}{c}\mathsf{Q_H} = (\mathsf{Q}^u_{\mathsf{H}}, \mathsf{Te_{max}}, \mathsf{Q}^p_{\mathsf{H}}) \\ \mathsf{F_G}(\mathsf{Q_G}, \mathsf{Q}^p_{\mathsf{H}}, \sigma_{\mathsf{G}}, \mathsf{Tg}) = (\mathsf{Q_G}', \mathsf{Q}^{u\,\prime\prime}_{\mathsf{H}}, \mathsf{Te_{max}}', \mathsf{Q}^{p\,\prime}_{\mathsf{H}}, \sigma_{\mathsf{G}}') \\ \mathsf{Q_H}' \leftarrow (\mathsf{Q}^u_{\mathsf{H}} \circledast \mathsf{Q}^{u\,\prime\prime}_{\mathsf{H}}, \mathsf{Te_{max}}', \mathsf{Q}^{p\,\prime}_{\mathsf{H}})\end{array}}{\begin{array}{c}(\sigma_{\mathsf{E}}, \sigma_{\mathsf{G}}, \sigma_{\mathsf{H}}, \mathsf{Q_G}, \mathsf{Q_H}, \mathsf{Q_E}, \mathsf{Tg}) \\ \rightsquigarrow_{\mathsf{G}} (\sigma_{\mathsf{E}}, \sigma_{\mathsf{G}}', \sigma_{\mathsf{H}}, \mathsf{Q_G}', \mathsf{Q_H}', \mathsf{Q_E}, \mathsf{Tg})\end{array}}\ \text{guest}$$

Note: $\circledast$ is the merge operation on (sorted) lists, lifted pointwise to key-value tuples pointwise on keys.

Figure 14: Assumptions and transition of the guest

```
function update(Φ, V)
Φ_out ← Φ
foreach (Eu_i, Te_i) ∈ V :
    Φ_out ← F^u_H(Φ, Eu_i, Te_i)
return Φ_out
```

```
function update_prof(Φ, U, Tg)
i ← min_j{U[j] = (Tu, _, _) and Tu > Tg}
U_curr ← U[..(i − 1)]
U_rest ← U[i..]
Φ_out ← update(Φ, U_curr|)
return (Φ_out, U_rest)
```

Figure 15: The functions update and update_prof that model HyPace's profile update logic

(Propagation delays are respected) Next, we formalize the fiat assumption that the guest accounts for propagation delays correctly. For this, we define a property $I_{\mathsf{delay}}(\mathsf{Q}^u_{\mathsf{H}})$ on profile update subqueues, and assume that this property holds of the output $\mathsf{Q}^{u\,\prime\prime}_{\mathsf{H}}$ of $\mathsf{F_G}$ (assumption (3) in Figure 14). $I_{\mathsf{delay}}(\mathsf{Q}^u_{\mathsf{H}})$ simply says that for any update tuple $(\mathsf{Tu}_i, \mathsf{Eu}_i, \mathsf{Te}_i)$ in $\mathsf{Q}^u_{\mathsf{H}}$, $\mathsf{Tu}_i \leq \mathsf{Te}_i$, meaning that the time at which the update gets queued (Tu_i) is no more than the intended effective time Te_i.

(Guest queues updates in increasing order) The new updates the guest queues ($\mathsf{Q}^{u\,\prime\prime}_{\mathsf{H}}$) should have effective times after $\mathsf{Te_{max}}$, and the new $\mathsf{Te_{max}}'$ should be an upper bound on the effective times in $\mathsf{Q}^{u\,\prime}_{\mathsf{H}}$. We formalize these as assumptions (4) and (5) in Figure 14.

Note. Any real guest would also have the following additional properties. These properties are not necessary for security, so we do not assume them. We mention them just for completeness.

(Causality) $\mathsf{F_G}$ should only depend on past events in $\mathsf{Q_G}$, i.e., those that occurred before Tg.

(Non-modification of past outputs) $\mathsf{F_G}$ should not output events in the past, i.e., $\mathsf{Q}^{u\,\prime}_{\mathsf{H}}$ should only contain events with timestamps greater than Tg.

(Non-consumption of future inputs) $\mathsf{F_G}$ should not remove future events from its input queue, $\mathsf{Q_G}$. Formally, $\mathsf{Q_G}$ and $\mathsf{Q_G}'$ should agree on events that have timestamps greater than Tg.

C.1.3 HyPace acts

In its turn to act, HyPace's work corresponds to its (batched) actions in the epoch $(\mathsf{Tg}, \mathsf{Tg} + \delta_e]$, where δ_e is the epoch length. HyPace does two things.

Assumptions

(6) $\mathsf{F}^u_{\mathsf{H}}(\Phi, \mathsf{Eu}, \mathsf{Te}) = \Phi'$ and $\mathsf{Te}' < \mathsf{Te}$
$\Rightarrow [\Phi]_{\mathsf{Te}'} = [\Phi']_{\mathsf{Te}'}$

(7) $s_1 = s_2$ and
$\mathsf{F}^o_{\mathsf{H}}(\mathsf{Q}^p_{\mathsf{H}1}, s_1) = (\mathsf{Q}^{p\prime}_{\mathsf{H}1}, \mathsf{Q}_{\mathsf{E}1}{}'')$ and
$\mathsf{F}^o_{\mathsf{H}}(\mathsf{Q}^p_{\mathsf{H}2}, s_2) = (\mathsf{Q}^{p\prime}_{\mathsf{H}2}, \mathsf{Q}_{\mathsf{E}2}{}'')$
$\Rightarrow$
$\mathsf{Q}_{\mathsf{E}1}{}'' \sim \mathsf{Q}_{\mathsf{E}2}{}''$

Transition

$$\frac{\begin{array}{c}\sigma_{\mathsf{H}} = \{\mathtt{flow}_i \mapsto \Phi_i\}_{i=1}^N \qquad \mathsf{Q}^u_{\mathsf{H}} = \{\mathtt{flow}_i \mapsto \mathsf{U}_i\}_{i=1}^N \\ (\Phi_i', \mathsf{U}_i') = \mathsf{update\_prof}(\Phi_i, \mathsf{U}_i, \mathsf{Tg}) \\ \sigma_{\mathsf{H}}{}' \leftarrow \{\mathtt{flow}_i \mapsto \Phi_i'\}_{i=1}^N \\ [\sigma_{\mathsf{H}}{}']_{\mathsf{Tg}} \triangleq \{\mathtt{flow}_i \mapsto [\Phi_i']_{\mathsf{Tg}}\}_{i=1}^N \\ \mathsf{F}^o_{\mathsf{H}}(\mathsf{Q}^p_{\mathsf{H}}, [\sigma_{\mathsf{H}}{}']_{\mathsf{Tg}}) = (\mathsf{Q}^{p\prime}_{\mathsf{H}}, \mathsf{Q}_{\mathsf{E}}{}'') \\ \mathsf{Q}^u_{\mathsf{H}}{}' \leftarrow \{\mathtt{flow}_i \mapsto \mathsf{U}_i'\}_{i=1}^N\end{array}}{\begin{array}{c}(\sigma_{\mathsf{E}}, \sigma_{\mathsf{G}}, \sigma_{\mathsf{H}}, \mathsf{Q}_{\mathsf{G}}, \mathsf{Q}_{\mathsf{H}}, \mathsf{Q}_{\mathsf{E}}, \mathsf{Tg}) \\ \rightsquigarrow_{\mathsf{H}} (\sigma_{\mathsf{E}}, \sigma_{\mathsf{G}}, \sigma_{\mathsf{H}}{}', \mathsf{Q}_{\mathsf{G}}, \mathsf{Q}_{\mathsf{H}}{}', \mathsf{Q}_{\mathsf{E}} \cup \mathsf{Q}_{\mathsf{E}}{}'', \mathsf{Tg} + \delta_e)\end{array}}\ \text{hypace}$$

Figure 16: Assumptions and transition of HyPace

Invariants (unary)

$(I1)$ $I_{\mathsf{delay}}(\mathsf{Q}^u_{\mathsf{H}})$
$(I2)$ $I_{\mathsf{emax}}(\mathsf{Q}^u_{\mathsf{H}}, \mathsf{Te}_{\max})$

where I_{delay} and I_{emax} are defined in Figure 14.

Invariants (relational)

$(I3)$ $\sigma_{\mathsf{E}1} \sim \sigma_{\mathsf{E}2}$ and
$\sigma_{\mathsf{G}1} \sim \sigma_{\mathsf{G}2}$ and
$\mathsf{Q}_{\mathsf{G}1} \sim \mathsf{Q}_{\mathsf{G}2}$ and
$I_{\mathsf{upd}}(\mathsf{Q}^u_{\mathsf{H}1}, \sigma_{\mathsf{H}1}, \mathsf{Q}^u_{\mathsf{H}2}, \sigma_{\mathsf{H}2})$ and
$\mathsf{Q}_{\mathsf{E}1} \sim \mathsf{Q}_{\mathsf{E}2}$ and
$\mathsf{Tg}_1 = \mathsf{Tg}_2$

where

$I_{\mathsf{upd}}(\mathsf{Q}^u_{\mathsf{H}1}, \sigma_{\mathsf{H}1}, \mathsf{Q}^u_{\mathsf{H}2}, \sigma_{\mathsf{H}2}) \triangleq$
$\forall i.\ (\mathtt{flow}_i \mapsto \mathsf{U}_1) \in \mathsf{Q}^u_{\mathsf{H}1}$ and
$(\mathtt{flow}_i \mapsto \mathsf{U}_2) \in \mathsf{Q}^u_{\mathsf{H}2}$ and
$(\mathtt{flow}_i \mapsto \Phi_1) \in \sigma_{\mathsf{H}1}$ and
$(\mathtt{flow}_i \mapsto \Phi_2) \in \sigma_{\mathsf{H}2}$
$\Rightarrow$
$\exists \mathsf{V}.\ (|\mathsf{U}_1| = \mathsf{V} +\!\!+ |\mathsf{U}_2|$ and $\Phi_2 = \mathsf{update}(\Phi_1, \mathsf{V}))$ or
$(|\mathsf{U}_2| = \mathsf{V} +\!\!+ |\mathsf{U}_1|$ and $\Phi_1 = \mathsf{update}(\Phi_2, \mathsf{V}))$

Figure 17: Invariants of the transition system

First, HyPace applies the longest prefix of profile updates from $\mathsf{Q}^u_{\mathsf{H}}$ whose availability timestamps Tu_i (*not* effective timestamps Te_i) are less than δ_e. This models applying pending updates that became available in the earlier epochs. We assume an abstract profile update function $\mathsf{F}^u_{\mathsf{H}}(\Phi, \mathsf{Eu}, \mathsf{Te}) = \Phi'$ that updates a current profile Φ to a new profile Φ' by applying the update Eu effectively from Te. Importantly, we assume that Φ and Φ' agree in the packet timing they provide up to time Te. In other words, the update becomes effective only at time Te. The updated profile Φ' is stored back in HyPace's internal state σ_{H}.

In detail, for each flow $\mathtt{flow}_i$, we iterate on the updates in $\mathsf{Q}^u_{\mathsf{H}}[\mathtt{flow}_i]$ from the left, till the first update event whose availability timestamp Tu_j is larger than Tg. All updates to the left of this event came to HyPace before the end of the previous epoch and are applied immediately to Φ_i using the function $\mathsf{F}^u_{\mathsf{H}}$ and stored back in σ_{H}. The remaining updates stay pending in $\mathsf{Q}^u_{\mathsf{H}}[\mathtt{flow}_i]$. This iteration is formalized by the defined function $\mathsf{update\_prof}$ shown in Figure 15. The function update.

Note that the meta-variable V stands for updates *projected* to only the update event Eu and the effective time Te (i.e., removing Tu).

$$\mathsf{V} \quad ::= \quad (\mathsf{Eu}_1, \mathsf{Te}_1), \ldots, (\mathsf{Eu}_m, \mathsf{Te}_m)$$

Second, HyPace uses the updated profiles to generate output packets for the NIC. Only profile prefixes up to Tg are considered. This process is abstractly modeled by a function $\mathsf{F}^o_{\mathsf{H}}(\mathsf{Q}^p_{\mathsf{H}}, [\sigma_{\mathsf{H}}]_{\mathsf{Tg}}) = (\mathsf{Q}^{p\prime}_{\mathsf{H}}, \mathsf{Q}_{\mathsf{E}}{}'')$. This function takes as input the current packet subqueue between the guest and HyPace, and the current profiles on all flows, limited to the time interval $(0, \mathsf{Tg}]$ (denoted by $[\sigma_{\mathsf{H}}]_{\mathsf{Tg}}$). It returns an updated packet subqueue $\mathsf{Q}^{p\prime}_{\mathsf{H}}$ (as some packets have been consumed), and a set of events $\mathsf{Q}_{\mathsf{E}}{}''$ to output to the NIC.

The entire HyPace transition is formalized in the rule (hypace) in Figure 16. The index H in $\rightsquigarrow_{\mathsf{H}}$ indicates that this is HyPace's transition.

We make the following assumptions about the functions $\mathsf{F}^u_{\mathsf{H}}$ and $\mathsf{F}^o_{\mathsf{H}}$.

(Profile update respects effective time) $\mathsf{F}^u_{\mathsf{H}}$ should respect the effective time Te_i passed as its third argument. Formally, we let $[\Phi]_{\mathsf{T}}$ denote the restriction of profile Φ to the interval $(0, \mathsf{T}]$. Assumption (6) of Figure 16 represents this requirement. (Note that $[\Phi]_{\mathsf{T}}$ is abstract; we don't define it. However, it is also used in the rule (hypace) of Figure 16 to limit profiles before using them to generate events.)

(HyPace does not leak secrets explicitly) $\mathsf{F}^o_{\mathsf{H}}$ should not leak information from the packet subqueue $\mathsf{Q}^p_{\mathsf{H}}$, which may be secret-dependent, into the timing of outgoing packets. Formally, this is represented by assumption (7) of Figure 16.

C.2 Security theorem and proof

We formalize confidentiality of the guest's private state using the standard concept of *noninterference* [61]. Noninterference

is inherently a relational property, i.e., a property of two runs of the system. Noninterference is usually proved by establishing *invariants*. We state and prove the relevant invariants of our model before stating and proving security. Our model has two kinds of relevant invariants: unary and relational.

Unary invariants A unary invariant is a property of the configuration that is *preserved* by all transitions, i.e., if the property holds before the a transition, then it holds after the transition as well. There are two unary invariants of relevance to us. These are called $(I1)$ and $(I2)$, and shown in Figure 17.

Lemma 1. $(I1)$ and $(I2)$ are (unary) invariants.

Proof. $(I1)$: We need to prove that every transition preserves $(I1)$, i.e., $I_{\mathsf{delay}}(\mathsf{Q}^u_{\mathsf{H}})$. This property is defined pointwise on the individual elements of $\mathsf{Q}^u_{\mathsf{H}}$, so it can be violated only by a transition that *adds* to $\mathsf{Q}^u_{\mathsf{H}}$. The only such transition is $\rightsquigarrow_{\mathsf{G}}$ ($\rightsquigarrow_{\mathsf{E}}$ does not change $\mathsf{Q}^u_{\mathsf{H}}$ and $\rightsquigarrow_{\mathsf{H}}$ removes from $\mathsf{Q}^u_{\mathsf{H}}$). However, $\rightsquigarrow_{\mathsf{G}}$ trivially preserves the invariant due to assumption (3) of Figure 14.

$(I2)$: We need to prove that every transition preserves $(I2)$, i.e., $I_{\mathsf{emax}}(\mathsf{Q}^u_{\mathsf{H}}, \mathsf{Te}_{\mathsf{max}})$. Again, this property is defined pointwise on the individual elements of $\mathsf{Q}^u_{\mathsf{H}}$, so we only need to consider the transition $\rightsquigarrow_{\mathsf{G}}$. This transition trivial preserves the invariant due to assumption (4) of Figure 14. □

Relational invariants A relational property is a property of two configurations, conventionally denoted by the subscripts 1 and 2. A relational property is called a *relational invariant* if the following holds: Consider two configurations in the property. If both configurations step with the same kind of transition then the resulting configurations are also in the property.[9]

For our model, there is only one interesting relational invariant – $(I3)$ of Figure 17. This relation says two things: (1) The two configurations agree on the public (non-private) components of σ_{E}, σ_{G}, Q_{G}, and Q_{E} (the private components may arbitrarily differ) and (2) For every flow $\mathtt{flow}_i$, HyPace's internal state σ_{H} and the pending updates queue $\mathsf{Q}^u_{\mathsf{H}}$ differ across the two configurations only in that, in one of the two sides, fewer updates have been taken out of $\mathsf{Q}^u_{\mathsf{H}}$ and applied to the flow's profile. In other words, the *same* profile updates reach HyPace (and with the same effectiveness timestamps Te_i) in the two runs, but the two runs may differ in *when* they apply the updates. The latter difference arises because the time at which the updates reach HyPace (the timestamps Tu_is) may depend on guest secrets and may differ.

Lemma 2. $(I3)$ is a relational invariant.

[9] We do not need to consider different types of transitions on the two sides since we fix the order of the transitions. In other words, we assume a deterministic scheduler. This can be easily relaxed to any scheduler that only looks at the non-private components of the configurations.

Proof. We assume that the unary invariants $(I1)$ and $(I2)$ hold. We then assume that $(I3)$ holds *before* a step, and show that it holds *after* the step as well. To prove the latter, we show that all conjuncts of $(I3)$ hold. We use the quote symbol $'$ to denote elements after the transition.

$\underline{\sigma_{\mathsf{E}1} \sim \sigma_{\mathsf{E}2}}$: The only transition that modifies σ_{E} is $\rightsquigarrow_{\mathsf{E}}$. This transition trivially guarantees $\sigma_{\mathsf{E}1}' \sim \sigma_{\mathsf{E}2}'$ due to assumption (1) of Figure 13.

$\underline{\sigma_{\mathsf{G}1} \sim \sigma_{\mathsf{G}2}}$: The only transition that modifies σ_{G} is $\rightsquigarrow_{\mathsf{G}}$. This transition trivially guarantees $\sigma_{\mathsf{G}1}' \sim \sigma_{\mathsf{G}2}'$ due to assumption (2) of Figure 14.

$\underline{\mathsf{Q}_{\mathsf{G}1} \sim \mathsf{Q}_{\mathsf{G}2}}$: The two transitions that modify Q_{G} are $\rightsquigarrow_{\mathsf{E}}$ and $\rightsquigarrow_{\mathsf{G}}$. Both guarantee $\mathsf{Q}_{\mathsf{G}1}' \sim \mathsf{Q}_{\mathsf{G}2}'$ – the former due to assumption (1) of Figure 13, and the latter due to assumption (2) of Figure 14.

$\underline{I_{\mathsf{upd}}(\mathsf{Q}^u_{\mathsf{H}1}, \sigma_{\mathsf{H}1}, \mathsf{Q}^u_{\mathsf{H}2}, \sigma_{\mathsf{H}2})}$: This invariant is affected only by transitions that change $\mathsf{Q}^u_{\mathsf{H}}$ or σ_{H} or both. There are two such transitions: $\rightsquigarrow_{\mathsf{G}}$ and $\rightsquigarrow_{\mathsf{H}}$.

$\rightsquigarrow_{\mathsf{G}}$ adds to $\mathsf{Q}^u_{\mathsf{H}}$. First, note that due to the clause $\mathsf{Q}^{u\prime\prime}_{\mathsf{H}1} \sim \mathsf{Q}^{u\prime\prime}_{\mathsf{H}2}$ in assumption (2) of Figure 14, the updates $\mathsf{Q}^{u\prime\prime}_{\mathsf{H}1}$ and $\mathsf{Q}^{u\prime\prime}_{\mathsf{H}2}$ that are added to $\mathsf{Q}^u_{\mathsf{H}1}$ and $\mathsf{Q}^u_{\mathsf{H}2}$ agree in their content and have the same effective times (they may differ in when they reach HyPace, but this is irrelevant for $I_{\mathsf{upd}}(\mathsf{Q}^u_{\mathsf{H}1}, \sigma_{\mathsf{H}1}, \mathsf{Q}^u_{\mathsf{H}2}, \sigma_{\mathsf{H}2})$ as it projects each U to U|). Second, due to invariant $(I2)$, all events $\mathsf{Q}^u_{\mathsf{H}1}$ have effective times less than $\mathsf{Te}_{\mathsf{max}1}$, while the new events being added have timestamps greater than $\mathsf{Te}_{\mathsf{max}1}$ due to assumption (5) of Figure 14. It follows that $\mathsf{Q}^{u\prime\prime}_{\mathsf{H}1}$ is simply *appended* to the end of $\mathsf{Q}^u_{\mathsf{H}1}$ by the $\divideontimes$ operation in rule (guest) of Figure 14. A similar observation holds for $\mathsf{Q}^{u\prime\prime}_{\mathsf{H}2}$ and $\mathsf{Q}^u_{\mathsf{H}2}$. This immediately implies that $I_{\mathsf{upd}}(\mathsf{Q}^{u\prime}_{\mathsf{H}1}, \sigma_{\mathsf{H}1}', \mathsf{Q}^{u\prime}_{\mathsf{H}2}, \sigma_{\mathsf{H}2}')$ holds.

$\rightsquigarrow_{\mathsf{H}}$ modifies $\mathsf{Q}^u_{\mathsf{H}}$ by taking a prefix of it and applying it to profiles. Hence, it trivially yields $I_{\mathsf{upd}}(\mathsf{Q}^{u\prime}_{\mathsf{H}1}, \sigma_{\mathsf{H}1}', \mathsf{Q}^{u\prime}_{\mathsf{H}2}, \sigma_{\mathsf{H}2}')$.

$\underline{\mathsf{Q}_{\mathsf{E}1} \sim \mathsf{Q}_{\mathsf{E}2}}$: The two transitions that modify Q_{E} are $\rightsquigarrow_{\mathsf{E}}$ and $\rightsquigarrow_{\mathsf{H}}$. Of these, $\rightsquigarrow_{\mathsf{E}}$ guarantees $\mathsf{Q}_{\mathsf{E}1}' \sim \mathsf{Q}_{\mathsf{E}2}'$ due to assumption (1) of Figure 13.

Showing that $\rightsquigarrow_{\mathsf{H}}$ guarantees $\mathsf{Q}_{\mathsf{E}1}' \sim \mathsf{Q}_{\mathsf{E}2}'$ is harder. First, note that from the rule (hypace) of Figure 16, $\mathsf{Q}_{\mathsf{E}1}' = \mathsf{Q}_{\mathsf{E}1} \cup \mathsf{Q}_{\mathsf{E}1}''$ and, similarly, $\mathsf{Q}_{\mathsf{E}2}' = \mathsf{Q}_{\mathsf{E}2} \cup \mathsf{Q}_{\mathsf{E}2}''$. $\mathsf{Q}_{\mathsf{E}1} \sim \mathsf{Q}_{\mathsf{E}2}$ by assumption about the invariant holding before the transition, so we only need to prove that $\mathsf{Q}_{\mathsf{E}1}'' \sim \mathsf{Q}_{\mathsf{E}2}''$. Now, again following the rule, $\mathsf{Q}_{\mathsf{E}i}''$ (for $i = 1,2$) is obtained from the function $\mathsf{F}^o_{\mathsf{H}}$, so by assumption (7) of Figure 16, we only need to show that $[\sigma_{\mathsf{H}1}']_{\mathsf{Tg}_1} = [\sigma_{\mathsf{H}2}']_{\mathsf{Tg}_2}$. We already know from the invariant before the transition that $\mathsf{Tg}_1 = \mathsf{Tg}_2 = \mathsf{Tg}$ (say) and, following the definition of $[\sigma_{\mathsf{H}}']_{\mathsf{Tg}}$, we only need to show that for every $i \in \{1, \ldots, N\}$, $[\Phi_{i1}']_{\mathsf{Tg}} = [\Phi_{i2}']_{\mathsf{Tg}}$. However, from the previous clause ($I_{\mathsf{upd}}(\mathsf{Q}^{u\prime}_{\mathsf{H}1}, \sigma_{\mathsf{H}1}', \mathsf{Q}^{u\prime}_{\mathsf{H}2}, \sigma_{\mathsf{H}2}')$) we know that Φ_{i1}' and Φ_{i2}' only differ in *which* of two identical sets of updates have been applied. However, all updates with *effective* times less than Tg *must have been applied* to both. To see this, note that the time at which any such update comes to HyPace (the timestamp Tu_i) has to be lower than the effective time due to invariant $(I1)$ and, hence, lower than Tg. So, $\mathtt{update\_prof}(\Phi_i, \mathsf{U}_i, \mathsf{Tg})$

in the rule (hypace) will apply all such updates (in both runs). Φ'_{i1} and Φ'_{i2} can still differ in applied updates with effective timestamps *after* Tg. However, due to assumption (6) of Figure 16, such differences are irrelevant for the projections $[\Phi'_{i1}]_{\mathsf{Tg}}$ and $[\Phi'_{i2}]_{\mathsf{Tg}}$. Hence, $[\Phi'_{i1}]_{\mathsf{Tg}} = [\Phi'_{i2}]_{\mathsf{Tg}}$, as required.

$\underline{\mathsf{Tg}_1 = \mathsf{Tg}_2}$: The only transition that modifies Tg is $\rightsquigarrow_{\mathsf{H}}$, but this transition increases Tg by a fixed amount (δ_e), so it trivially preserves equality of Tg_1 and Tg_2. □

Security We formulate security as follows standard non-interference. Consider two runs that both start from empty queues, the same states for clients and the network, the same non-private state for the guest, but possibly *different private guest state*. Then, after n steps in each run, the non-private state of the environment is exactly the same. Let $\emptyset$ denote an empty queue.

Theorem 3 (Security)**.** Let (1) $\sigma_{\mathsf{G}1} \sim \sigma_{\mathsf{G}2}$, (2) $(\sigma_{\mathsf{E}}, \sigma_{\mathsf{G}1}, \sigma_{\mathsf{H}}, \emptyset, \emptyset, \emptyset, \mathsf{Tg}) \rightsquigarrow^n (\sigma'_{\mathsf{E}1}, \_, \_, \_, \_, \_, \_)$ and (3) $(\sigma_{\mathsf{E}}, \sigma_{\mathsf{G}2}, \sigma_{\mathsf{H}}, \emptyset, \emptyset, \emptyset, \mathsf{Tg}) \rightsquigarrow^n (\sigma'_{\mathsf{E}2}, \_, \_, \_, \_, \_, \_)$. Then $\sigma'_{\mathsf{E}1} \sim \sigma'_{\mathsf{E}2}$.

Proof. It is trivial to see that $(I1)$, $(I2)$ and $(I3)$ hold of the starting states. Since these properties are invariants, they hold of the final states. In particular, from $(I3)$ on the final state, we get that $\sigma'_{\mathsf{E}1} \sim \sigma'_{\mathsf{E}2}$. □