Qiuyu Xiao*, Michael K. Reiter, and Yinqian Zhang

Personalized Pseudonyms for Servers in the Cloud

Abstract: A considerable and growing fraction of servers, especially of web servers, is hosted in compute clouds. In this paper we opportunistically leverage this trend to improve privacy of clients from network attackers residing between the clients and the cloud: We design a system that can be deployed by the cloud operator to prevent a network adversary from determining which of the cloud's tenant servers a client is accessing. The core innovation in our design is a PoPSiCl (pronounced "popsicle"), a persistent pseudonym for a tenant server that can be used by a single client to access the server, whose real identity is protected by the cloud from both passive and active network attackers. When instantiated for TLS-based access to web servers, our design works with all major browsers and requires no additional client-side software and minimal changes to the client user experience. Moreover, changes to tenant servers can be hidden in supporting software (operating systems and web-programming frameworks) without imposing on web-content development. Perhaps most notably, our system boosts privacy with minimal impact to web-browsing performance, after some initial setup during a user's first access to each web server.

Keywords: server anonymity, pseudonyms, cloud computing

DOI 10.1515/popets-2017-0034
Received 2017-02-28; revised 2017-06-01; accepted 2017-06-02.

1 Introduction

Monitoring of online activities is a fact of life for many users, be it by, e.g., an employer to detect activity that is inconsistent with corporate policy or a government to monitor sites accessed by its citizens. While encryption is a first line of defense against monitoring, additional steps must be taken to hide the servers accessed by users. Numerous techniques have thus been developed to support *server-anonymous* access, i.e., access to a server in a way that hides the server identity from a monitor. The socalled "hidden service", supported by Tor [11], is an example of a technology that enables server-anonymous access.

The growth of large hosting infrastructures such as compute clouds and content distribution networks (CDNs) has opened new opportunities for deploying server-anonymous systems (see Section 2 for a discussion). Because these hosting platforms serve content from many tenant servers, intermingling controversial server content or anonymizing proxies among them (i.e., as other tenants) can make it more difficult or expensive for a monitor to disambiguate which tenant server a client is accessing. The vast resources and connectivity available via these infrastructures can also expand the capacity of server-anonymous systems. However, prior attempts (of which we are aware) to leverage these infrastructures to support server anonymity have been designed for an oblivious cloud operator that (at best) provides no specific support for server anonymity (e.g., [4, 16]). Changes are thus typically hoisted onto the client users of these systems, who often lack the permissions, trust, or know-how to do so.

In this paper, we instead propose a design for server-anonymous communication to tenant servers of a cloud that leverages support and cooperation of the cloud provider and, in doing so, avoids requiring any changes to client software. Our central innovation to enable this capability is a PoPSiCl (pronounced "popsicle"), a Personalized Pseudonym for a Server in the Cloud. Specifically, a PoPSiCl is a domain name with the following properties: First, it is *personalized*: A PoPSiCl can be used to access the tenant server only by the client for which the cloud generated it. Second, it is a *pseudonym*: A PoPSiCl is a persistent identifier that the client to whom it is issued can use to access the server over time (e.g., by bookmarking it). Moreover, the cloud protects the identity of the tenant server accessed using this PoPSiCl from an attacker who can both observe the client's communication with the cloud and probe the cloud as another client or tenant server itself. Though the cloud is trusted in our design, note that today the

***Corresponding Author: Qiuyu Xiao:** University of North Carolina at Chapel Hill, E-mail: qiuyu@cs.unc.edu
Michael K. Reiter: University of North Carolina at Chapel Hill, E-mail: reiter@cs.unc.edu
Yinqian Zhang: The Ohio State University, E-mail: yinqian@cse.ohio-state.edu

cloud is already typically trusted with knowing which users frequent a tenant server. Even if the user connects to a tenant server using an anonymizing service such as Tor, the cloud can access any identifying information the user provides to the tenant server, either intentionally (e.g., an email address) or not (e.g., HTTP cookies or browser fingerprints [5, 28, 29][1]). In such cases, our trust in the cloud does not substantially increase the trusted computing base for user privacy.

A design goal for PoPSiCls is that they can be implemented by the cloud operator in a way that is unobtrusive to their tenants or their tenants' clients. Specifically, we demonstrate an implementation of PoPSiCls to support private TLS accesses to web servers in the cloud that has the following features:

- Our implementation requires no changes to client-side software and works with all major web browsers. This stands in contrast to most work on anonymous access to servers (see Section 2) that requires the installation of proxies on client computers or the installation of a custom browser (e.g., Tor). Requiring no changes to the client is important for users who lack either the permissions needed to modify their client platforms (as an employee using a company-owned computer might) or the willingness to do so (e.g., since even security software is often riddled with vulnerabilities [9, 35, 38]).
- The only changes in user experience for supporting use of PoPSiCls is the use of client-side TLS certificates to support TLS connections, and a visit to a cloud-operated *PoPSiCl store* to obtain a PoPSiCl and the client-side certificate for a tenant server prior to her first (server-anonymous) access to that server.
- A tenant server requires some changes to its OS and, if the server is a web server, minimal other changes that can be hidden within high-level web programming frameworks like Ruby on Rails. So, these changes can be packaged either in a platform-as-a-service (PaaS) cloud offering or a virtual machine (VM) image for deployment to an infrastructure-as-a-service (IaaS) cloud, without imposing on web-content developers.

Supporting PoPSiCls does impose more substantially on cloud infrastructure, notably through the establishment of the PoPSiCl store; in dynamic generation of switching rules to configure software-defined networking (SDN) switches in the cloud infrastructure; and, as mentioned above, in tenant server operating systems. We detail the changes needed to OpenStack and Linux to implement the needed functionality. Our implementation therefore most directly reflects how an IaaS cloud operator could deploy and support PoPSiCls, with OS modifications provided through PoPSiCl-enabled virtual-machine images. We envision that a cloud operator might be motivated to support PoPSiCls as one component of a larger "security as a service" offering, charging tenant servers for PoPSiCl use, perhaps per PoPSiCl or even per PoPSiCl-based connection.

We have used CloudLab (https://www.cloudlab.us/) to characterize the performance impact of PoPSiCl usage, versus regular (non-server-anonymous) web browsing over TLS. Our results show that our design introduces modest overhead to server access latency and throughput, and is capable of scaling to large numbers of users, should PoPSiCl use catch on. We also show that the access latency of our implementation is considerably better than proxy-based systems such as Tor, which also enhance privacy for server access, as discussed above. (We caution the reader, however, that the threat model and protections offered by PoPSiCls are different than those for which systems like Tor were designed, as we will discuss in Section 2 and especially Section 7.)

The rest of this paper is structured as follows. We provide background and related work in Section 2, and outline the principles behind our design in Section 3. Section 4 contains our high-level system design, and Section 5 describes our current implementation. We evaluate that implementation in Section 6 and detail the limitations of our design in Section 7. Finally, we conclude in Section 8. In Appendix A, we extend our design to address some forms of traffic analysis.

2 Background and Related Work

The goal of our design of PoPSiCls is to provide *server anonymity* (elsewhere called *recipient anonymity* [34] or *recipient untraceability* [6]) against network attackers. That is, a network attacker can observe that a client is initiating communication with a server in the cloud, but the attacker is unable to determine the specific server with which the client is communicating. In this context, a PoPSiCl is an *implicit address* [34] for a tenant server; moreover, it is *visible* in that its reuse to reconnect to the server is evident—both to the cloud, which can use the PoPSiCl to route the client to the physical machine

1 The Tor Browser tries to mitigate browser fingerprinting by restricting browser features, but this requires reacting to new attacks as they are discovered [5, 29] and has an inevitable impact on usability.

currently hosting the tenant server, and to the attacker. The servers that appear to the adversary to be the possible targets of the client (i.e., the server's *anonymity set* [6]) is the set of all tenant servers in the same cloud datacenter as the target.

The most widely used methods to achieve server anonymity today are based on proxying (e.g., Tor [11], Psiphon (https://psiphon.ca/), and early versions of the Anonymizer [3]) or VPNs (such as the current Anonymizer, https://www.anonymizer.com/). Unlike these systems, PoPSiCls are not supported in our design through proxying or VPN tunneling. In particular, communication to a PoPSiCl is encrypted by the client and decrypted only by the tenant server in the cloud (vs. at a proxy or tunnel endpoint), leaving few opportunities for accidental leakage. Moreover, as we will show, proxies can become performance bottlenecks, and so our design scales better to heavy usage.

Variations on the goal of server anonymity have been studied in several forms, often under the rubric of *censorship resistance*. Like our design, several in this space leverage infrastructure providers explicitly (clouds, CDNs, or ISPs) to hide the server with which a client is trying to interact.

- In domain fronting [16], a client connects to a CDN edge server or reflector web application run in the cloud via a *front domain* other than the *hidden domain* of actual interest. The edge server or reflector then inspects the plaintext payload (e.g., the HTTP `Host` header) to discover the hidden domain and retrieves it for the client. A PoPSiCl can be viewed as a front domain, though the mapping to its hidden domain is maintained by the cloud operator and managed without inspecting the client's payload or, more to the point, without decrypting it, which is better for client/tenant security.
- CacheBrowser [20] enables a website's content to be retrieved from any CDN edge server without a DNS resolution, leveraging the assumption that it is untenable for censors to block IP addresses of CDN edge servers due to the collateral damage it would cause. However, this system still exposes the true server domain in the SNI field (see Section 3.1) and so is not truly server-anonymous. Recent improvements [41] rectify this concern, but do so in a way that is incompatible with some CDNs. In either case, these solutions work only for cacheable content.
- CloudTransport [4] repurposes cloud *storage* to implement interactive communication to a server in a way that will evade common censorship techniques.
- Telex [40] enables friendly on-path ISPs to recognize "tagged" traffic addressed to uncensored websites and divert it to the censored websites for which it is really intended. Tagging is implemented in the SSL handshake protocols, by embedding a tag into the random value field in the `ClientHello` message.
- LAP [21], Dovetail [37], HORNET [7], and PHI [8] are network-layer protocols that aim to provide low-latency and high-throughput anonymous communication. In these protocols, the source and destination addresses are encrypted so that the intermediate routing node only knows its adjacent nodes in the path (similar to Tor).
- There have been several proposals to host Tor relays in clouds [22, 27]. Moreover, systems like Tor have tended to be vulnerable to censors because users are connected to a small set of entry points that can be blocked. So, prior works have proposed to reduce the disclosure of IP addresses of Tor entry points through Tor bridges (a variation of keyspace hopping [15]; see https://www.torproject.org/docs/bridges) and, through the deployment of the Tor Cloud project (https://cloud.torproject.org/), to run Tor bridges inside clouds.

As they relate to our work, all of the above approaches require modifying client-side software. In contrast, our design requires no client-side software changes at all (albeit while requiring changes to infrastructure, as many of the above designs also require). That said, we stress that in contrast to some of the works above, our goal here is not censorship resistance, per se, but rather server anonymity, as the assumption of a trustworthy and cooperative cloud is somewhat at odds with the former. We discuss this issue further in Section 7.

3 Design Principles

In this section we detail the security (Section 3.1) and usability (Section 3.2) goals of our system.

3.1 Security

Threat model. We begin our discussion of the security principles of our design by recalling our threat model. A cloud hosts tenant servers, to which clients can connect (e.g., using TLS). As is the case today, the cloud operator is trusted by both tenant servers and their clients. We are concerned with enabling a client

to connect to a tenant server without divulging to an attacker the tenant server to which it is connecting. The client machine is trusted, as is the tenant server to which it connects. Other clients and other tenant servers are not trusted in the context of this client-server interaction; i.e., the identity of the server to which the client connects should remain hidden despite the efforts of other clients and other tenant servers. We also allow the attacker to capture and manipulate all traffic outside the cloud premises, including traffic to or from the client, but traffic within the cloud is invisible to the attacker (except if the attacker controls the source or destination of the traffic).

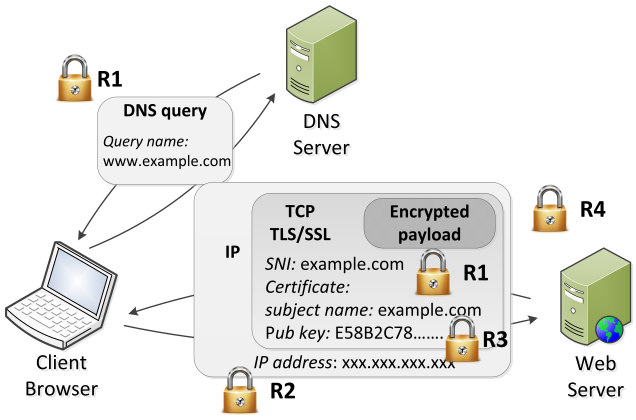


Fig. 1. In our threat model, the server identity can leak via the client's DNS query, the SNI field of the client-to-server TLS connection, the server IP address, or the server public key

Our threat model gives the attacker many opportunities to observe the identity of the tenant server to which a client connects in today's clouds (see Fig. 1). First, the DNS resolution of the server domain name can reveal that domain name to the attacker. Second, the IP address to which the client connects will be visible to the attacker; the attacker can then connect to this IP address itself to see what the server provides, or simply use this IP address to determine the server's identity from a preassembled database (like a reverse DNS lookup). The connection process itself can offer additional opportunities for the attacker to identify the server; in particular, a TLS connection exposes the server domain name in the Server Name Identification (SNI) field that the client sends, and in the certificate that the server provides to the client. The certificate also exposes the server's public key, which can be matched against the public keys in certificates obtained by other clients.

We leave several types of attack outside our scope, relying on orthogonal defenses to address them. For example, we assume the security of TLS and that the attacker can impersonate neither any cloud-provided service or tenant server that it does not control (by virtue of not having the needed server private key), nor any client that it does not control (by virtue of not having the client private key).

Also outside our scope are connection-level features that can divulge indications of the server involved in the connection, such as have been used in TCP fingerprinting (e.g., [17]), TLS fingerprinting (e.g., [26]), or website fingerprinting (e.g., [13, 31, 39]). Such features include the number of servers to which the client connects, the timing connections to relative to one another, connection volume patterns, etc. That said, we have made initial progress toward a framework for traffic-analysis defense, as we will discuss in Appendix A.

Security principles. To achieve server anonymity in the threat model described above, several steps are necessary. The first is to replace the server's domain name with a different domain name—the PoPSiCl—everywhere it is visible to the attacker. So, it will be necessary to cause the PoPSiCl to be used in the client's DNS lookup, the TLS SNI field, and the certificate that the tenant server sends to the client. In our system, the PoPSiCl takes the form *str*`.popsicls.com` where `popsicls.com` is the domain name of the cloud and *str* is a string that represents the PoPSiCl prefix. So, for example, `1f5qz7nfhj1uworr7laduh9fen.popsicls.com` might be a PoPSiCl. Of course, the PoPSiCl prefix *str* must be generated for this client in a way that prevents the attacker from correlating it with PoPSiCls generated for other clients to access the same tenant server.

R1: *The PoPSiCl for a client to access a tenant server is independent of the PoPSiCl generated for other clients, and the PoPSiCl is used in place of the tenant server's domain name everywhere that domain name appears in client communication.*

The PoPSiCl is intended to be a long-lived identifier that the client can use to access the server. To that end, the client uses the PoPSiCl just like any other domain name—by performing a DNS lookup on it to obtain an IP address to which to address network packets. To prevent the attacker from using this IP address to identify the server, however, the DNS resolution must produce a *pseudo-address*, which is a different IP address that the one the server actually uses. More specifically, a pseudo-address is a publicly routable IP address that is part of the IP address block allocated to the cloud,

so that a packet addressed to the pseudo-address will eventually reach a switch in the cloud datacenter. However, the pseudo-address should be otherwise unrelated to the actual IP address of the tenant server.

R2: *The pseudo-address to which a client addresses packets for the tenant server (and from which return packets arrive to this client) is independent from the actual network endpoint (i.e., IP address) of the tenant server in the cloud.*

A pseudo-address can be used in our system to establish a TLS connection to the tenant server associated with the PoPSiCl. TLS connection establishment introduces other potential identifiers that might be used to deanonymize the tenant server, particularly the public-key certificate for the server. As such, the tenant server should use a different public key per client.

R3: *In a TLS connection setup with a client that is accessing the server using a PoPSiCl, the tenant-server public key used was generated independently of the server public keys used in its TLS connections with other clients (regardless of whether those clients use PoPSiCls to access the server).*

There remains the risk that the attacker who observes the pseudo-address could simply connect to that pseudo-address itself and identify the server based on the content returned. To prevent this possibility, the client for which the PoPSiCl was created should be the only one that can complete a secure connection using it.

R4: *A tenant server completes a TLS connection setup with a client using a PoPSiCl only if that PoPSiCl was registered for use by that client, with this tenant server.*

3.2 Usability

Here, *usability* refers to the operational impact of PoPSiCls on all actors in the cloud ecosystem—the cloud operator, cloud tenants, and the tenant's clients. The approach we take in our design of a system to support PoPSiCls is to place a larger usability burden of deploying PoPSiCls on those groups of actors with greater technical capabilities. Major cloud operators arguably offer the highest concentration of technical capability, as their datacenter functioning is integral to all tenants' availability and security. As such, they will bear the greatest burden in supporting our design. Tenants who deploy servers to the cloud typically require at least a knowledge of how to populate a server with content, and so we will limit our design to small modifications to that process (at least in the case of web servers). Finally, we presume the tenants' clients might be driven by wholly nontechnical users (e.g., via web browsers) who might not have the permissions needed to install software on their computers (e.g., as a user of a corporate-controlled computer might not) or a willingness to do so (e.g., due to the vulnerabilities that such software can introduce [9, 35, 38]). So, we place a priority on minimizing client-side changes.

Treating these groups in reverse order, then, our first requirement is that changes to clients be very limited.

R5: *PoPSiCls are usable with no changes to client-side software, including no browser extensions or add-ons, in the case of web clients. While PoPSiCls are visible to clients and may require some adaptation of client user procedures (in the case of typical web browsing, for example), these adaptations are already supported by the dominant software clients in the market.*

The primary operational adaptations required by our design for a web user, for example, are the following. First, our design involves the use of client authentication via a client-side certificate in TLS. While not without its issues [32], support for client authentication is already in all major browsers and is in use by large communities (e.g., in Estonia, due to its national PKI initiative [32], and by MIT faculty, staff, and students to access some web services[2]). Second, a user must take an additional, online step to obtain ("register") a PoPSiCl for future accesses to a website. We will describe PoPSiCl registration in Section 4.1.

For tenant servers, who might range from large, well-staffed organizations to small online vendors, we allow changes to the software they use but require that those changes can be made largely "invisible" to them, if they so choose.

R6: *Changes to tenant servers can be hidden so that they do not impose on server content creation. For example, changes involving the tenant-server operating system (OS) or content-programming frameworks can be packaged within a virtual-machine image that respects existing application programmer interfaces (APIs). As such, a tenant-server creator should be able to "port" his content to this VM image with minimal effort.*

2 http://ist.mit.edu/certificates/guide

Our design intrudes on tenant servers primarily by requiring specific OS-level changes, discussed in Section 4.2. These can be packaged within a VM for deployment to Infrastructure-as-a-Service (IaaS) clouds. Alternatively, in a Platform-as-a-Service (PaaS) cloud, the OS is managed by the cloud operator, and so these changes would be invisible to the tenant server. In addition, some defenses specific to HTTP servers described in Section 4.3 and an optional extension described in Appendix A induce very minor additional changes to modern web programming frameworks (such as Ruby on Rails).

We allow for our design to impact cloud operators more directly. Again, though, cloud operators are the most technically savvy and so presumably the most capable of accommodating such changes.

4 Design

In this section we describe the design of a system to enable a cloud operator to implement PoPSiCls for its tenants and their clients. In order to use a PoPSiCl to access a tenant server, a client must first *register* the PoPSiCl, a process described in Section 4.1. The mechanisms supporting the use of a PoPSiCl to connect to a tenant server are described in Section 4.2. Adaptations specific to supporting HTTP clients using PoPSiCls are described in Section 4.3. Finally, how our design achieves the requirements laid out in Section 3 is the topic of Section 4.4.

4.1 Registering a PoPSiCl

The registration of a PoPSiCl is a user-initiated process, involving connecting to a particular cloud-operated service, the PoPSiCl store, using a web browser. The connection should employ TLS, though need not require a password login or any other form of client authentication. Rather, TLS is employed here simply to protect the privacy of the user. Upon accessing the PoPSiCl store, the user is presented with a web form to indicate the domain name, say `tenantA.com`, for which she wishes to register a PoPSiCl. Since we are trusting the cloud operator, we assume that if `tenantA.com` is not, in fact, hosted by the cloud, then it will decline the registration.

If `tenantA.com` is one of its tenants, then the PoPSiCl store takes the following actions (see Fig. 2a).

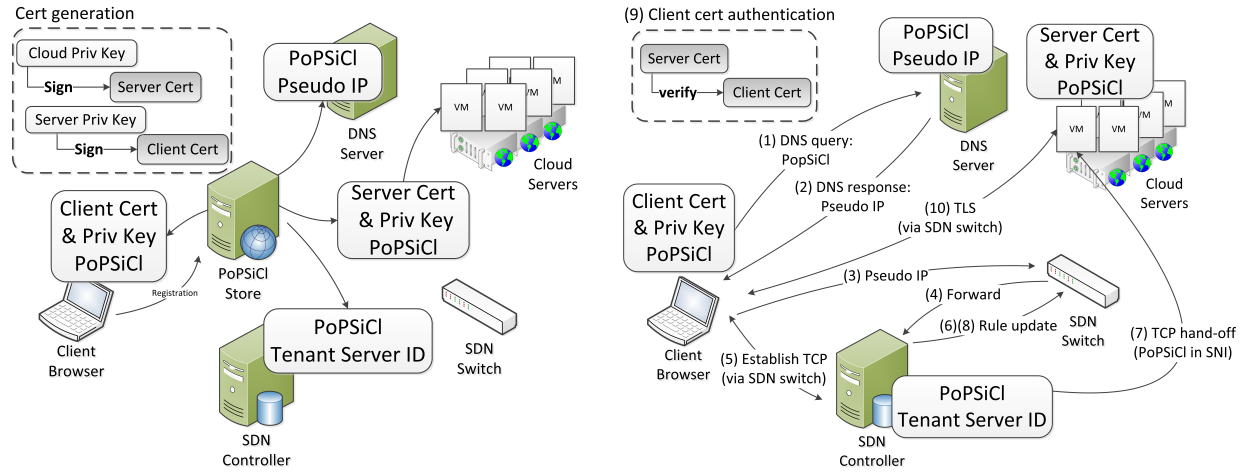
(i) The PoPSiCl store first creates a new PoPSiCl for `tenantA.com`, of the form *str*`.popsicls.com`, where *str* denotes a string of characters allowed in domain names. It then creates and exports a DNS record for *str*`.popsicls.com` that maps this PoPSiCl to one or more publicly routable IP addresses in the address ranges allocated to `popsicls.com`, to which we refer as pseudo-addresses. As we will see in Section 4.2, these pseudo-addresses are addresses of SDN controllers in the same datacenter (region) as `tenantA.com`. The PoPSiCl store also informs these SDN controllers that this PoPSiCl corresponds to `tenantA.com`.

(ii) The PoPSiCl store creates a new public/private keypair for use by `tenantA.com` and binds the public key to *str*`.popsicls.com` in a TLS server certificate signed by the PoPSiCl store. The PoPSiCl store delivers this private key and server certificate to the tenant server. It also delivers to the tenant server a root certificate for authenticating client certificates in TLS connection attempts to *str*`.popsicls.com`. For our purposes, it suffices for the newly generated server certificate to be used as this root certificate, as well. (Alternatively, a different root certificate generated for this specific purpose could be used.) This step assumes a tenant server capable of receiving this information. As described in Section 5, the Nginx web server already supports this capability, for example.

(iii) The PoPSiCl store generates a public/private keypair for the client to use to authenticate itself when connecting to *str*`.popsicls.com`; creates a certificate for this public key that can be verified using the root certificate of Step (ii); and returns the private key, public-key certificate, and PoPSiCl to the user. The user then saves this information and takes whatever steps are necessary to permit its TLS client to make use of this key when connecting to *str*`.popsicls.com`. For example, if the client is a web browser, then the client might bookmark the PoPSiCl and import this key pair and certificate into the browser. This step is already supported by major browsers.

Prior to connecting to *str*`.popsicls.com`, the client must also be configured with the PoPSiCl store as a certificate authority (CA) for TLS server certificates. In this way, the client will accept the tenant server's certificate (see Step (ii)) during TLS connection setup.

A user's first (or any) connection to the PoPSiCl store could be used to obtain a PoPSiCl for the PoPSiCl



(a) Registration, described in Section 4.1 **(b) Access, described in Section 4.2**

Fig. 2. Steps for registering a PoPSiCl (Fig. 2a) and then using it (Fig. 2b)

store, so that subsequent accesses to the PoPSiCl store can be hidden from an attacker.

4.2 Connection establishment

Via the steps described in Section 4.1, the client is in possession of a PoPSiCl for the server to which it wants to connect. To connect to this server, the client performs a DNS lookup on the PoPSiCl, as it would any other server domain name (steps 1–2 in Fig. 2b). Because the PoPSiCl is of the form *str*`.popsicls.com` where `popsicls.com` is the cloud domain name, the cloud ultimately provides the IP address returned to this DNS query. The IP address provided by the cloud is not the actual IP address of the machine hosting the tenant server (this would violate **R2**), but rather must be independent of it. One option would be to return the IP address of a (reverse) proxy in the cloud that relays client requests to the tenant server associated with the PoPSiCl (and responses back to the client); this design has similar security properties to ours, but as we will see in Section 6.2, this proxy will become a bottleneck. Rather, in our design, the DNS query is resolved to a publicly routable IP address of an SDN controller in the same datacenter (region) as the tenant server corresponding to *str*`.popsicls.com`; we refer to this IP address as a pseudo-address for the tenant server.

Let the pseudo-address be denoted by *pseudo-IP*, and let *client-IP* and *client-port* denote the source IP address and source port of the first packet that the client sends to *pseudo-IP* (a TCP SYN), when it arrives at the cloud switch (step 3 in Fig. 2b). *client-IP* and *client-port* need not be the actual IP address and port of the client; rather, these could instead be the address and port of a network-address translator (NAT) or proxy between the client and the cloud. Unless it has a higher-priority rule (see below) that matches specifically this source IP address (*client-IP*), source port (*client-port*), destination address (*pseudo-IP*), and destination port (denoted *server-port*), the switch forwards the packet using the default routing rule for this destination (*pseudo-IP*). Since in our design, the *pseudo-IP* is set to an IP address of the SDN switch controller in the cloud datacenter, this TCP SYN packet is forwarded to the controller (step 4 in Fig. 2b). In this case, the controller responds with a TCP SYN-ACK and completes the TCP connection with the client (step 5 in Fig. 2b).

After completing the TCP connection, the client then launches the TLS handshake. At this point, the controller learns *str*`.popsicls.com` from the `ClientHello` SNI field, with which it can look up the tenant server to which this PoPSiCl corresponds, say with IP address *tenant-IP*. The controller then takes the following steps (without responding with a `ServerHello` message), in order:

(i) The controller installs two new rules in the switch (step 6 in Fig. 2b). One matches packets with source address *client-IP*, source port *client-port*, destination address *pseudo-IP*, and destination port *server-port*; this rule simply drops any such packet silently. The second matches packets with source address *tenant-IP*, source port *server-port*,

destination address *client-IP*, and destination port *client-port*; changes the source address to *pseudo-IP*; and forwards the packet toward *client-IP*.[3] These rules have higher priority than any other rules that apply to the same packets.

(ii) The controller then transfers the TCP connection state to the tenant server OS (step 7 in Fig. 2b), including the buffer containing the `ClientHello`, and so the tenant server picks up the TCP session where the controller left off (by responding with a `ServerHello`). Our implementation of TCP connection transfer uses the technique in the `tcpcp` tool [1].

(iii) The controller then replaces the drop rule installed in Step (i) above (i.e., matching packets with source address *client-IP*, source port *client-port*, destination address *pseudo-IP*, and destination port *server-port*) to instead change the destination address of any matching packet to *tenant-IP* and then to forward the packet toward *tenant-IP* (step 8 in Fig. 2b).

The TLS connection establishment that the tenant server continues with the client requires the client to present a client certificate that can be verified by the server certificate for this PoPSiCl (step 9 in Fig. 2b). The tenant server received the server certificate for this PoPSiCl, and the client received this matching client certificate, in the PoPSiCl registration process (Section 4.1). If this client certificate is not sent, then the TLS connection fails; otherwise, the TLS connection can be established and communication proceeds as normal (step 10 in Fig. 2b).

The above ordering of steps is chosen purposely. The drop rule is installed in Step (i) to drop any inbound messages from the client during the TCP state transfer, which could confuse the transfer process. The other rule in Step (i) is installed to ensure that the `ServerHello` and the following messages sent by the tenant server in Step (ii), when the TCP state transfer completes, are forwarded to the client with the *pseudo-IP* as their source addresses. The controller replaces the drop rule from Step (i) as described in Step (iii) to permit the connection between the client and tenant server to continue. Of course, it is possible that the controller does not finish Step (iii) before the client sends another message, in which case the drop rule from Step (i) will drop it. We leverage TCP's retransmission capabilities to overcome any such drops that occur.

A possible denial-of-service attack against our architecture is to overwhelm the SDN controller(s), which will handle all TCP connections established using PoPSiCls until they are handed off to their tenant servers. For this reason, the controllers must be defended using state-of-the-art denial-of-service defenses. That said, note that our design allows for multiple SDN controllers and switches, and load-balancing among them.

4.3 HTTP-specific mechanisms

PoPSiCls can be used to support any type of server accessed using TLS. Overwhelmingly, however, the most common example today is HTTP, and so in this section we address several issues specific to their use to support access to HTTP servers.

Same-origin policy and cookies. Our architecture for supporting PoPSiCls permits the browser to accurately track origins, i.e., to support its same origin policy [36], since the browser is provided a unique domain name (the PoPSiCl) per tenant domain. This same property also enables the browser to send cookies to (only) the right domains—avoiding a pitfall of some previous anonymous communication systems (e.g., early versions of the Anonymizer [3]). To prohibit tenants from setting cookies for the cloud domain (e.g., `popsicls.com`), the cloud operator should add `popsicls.com` to the public suffix list[4], just as is, e.g., `amazonaws.com` today.

Object hyperlinking. When a tenant web server accessed using a PoPSiCl serves hyperlinks to its own objects, their URLs should leverage the PoPSiCl as their domain name. Otherwise, the browser would retrieve these objects using a URL with a different domain name, causing them to be viewed by the browser as coming from a different origin. This could cause the web page to malfunction, or it could result in disclosure of the true domain name to an attacker. Fortunately, this is achieved easily by hyperlinking to relative URLs, or by authoring web content with the domain name inserted by a macro that is resolved to the PoPSiCl with which the current client is accessing the server.

3 The installation of this second rule assumes that return packets traverse this same switch. If they do not, then this second rule would need to be inserted into another switch that they will traverse.

4 https://publicsuffix.org/

Hyperlinking between servers requires additional attention, since one server should not learn the PoPSiCl that a client uses to access another server. First, to ensure that a client browser does not disclose the PoPSiCl that it uses for accessing a tenant server, say `tenantA.com`, to another server to which `tenantA.com` refers the client (i.e., in the HTTP `Referer` field), `tenantA.com` should set the referrer policy of its referring page to `no-referrer` or `same-origin`.[5] Second, to allow referrals to a tenant server, say `tenantB.com`, without disclosing the server's identity to our attacker, the cloud operator `popsicls.com` can support hyperlinking to it using a URL such as `https://linker.popsicls.com?tenantB.com/ ...`, where `linker.popsicls.com` is a cloud-operated server. Upon receiving the TLS connection from the client, `linker.popsicls.com` can look up the PoPSiCl that this client uses to access `tenantB.com` (authenticating the client using its client certificate) and then redirect the client browser to that PoPSiCl. (For reasons discussed below, however, `linker.popsicls.com` will have to apply additional policy before doing so.) If no such PoPSiCl exists, then `linker.popsicls.com` can simply redirect the client to `tenantB.com`. To support hyperlinking in this fashion, the PoPSiCl store should provide a client-side certificate and accompanying private key for the client to use to connect to `linker.popsicls.com`, during the client's first PoPSiCl registration (for a web server) at the PoPSiCl store.

Cross-origin attacks in browsers. A tenant server accessed using a PoPSiCl does not complete a TLS connection setup with a client other than the one that registered that PoPSiCl. An attacker who obtains a PoPSiCl in use by a client, say `1f5...fen.popsicls.com`, is thus unable to connect to it directly in an effort to retrieve content from it (and thereby deanonymize it). Through cross-origin side channels, however, the attacker could potentially infer the true server identity behind a PoPSiCl. For example, the attacker could set up a web server (not necessarily in the cloud) and, if it could convince the client browser to visit its server, could serve back to the client a hyperlink, say `https://1f5...fen.popsicls.com/`*path*, that uses the PoPSiCl as its domain name. An attacker script could then test if the browser successfully retrieved the object at this URL,[6] thereby inferring whether *path* is a valid path at the tenant server. Since this might be a distinctive pathname, the attacker could deanonymize the server this way.

To prevent such cross-origin attacks, the tenant server refuses requests for URLs containing the PoPSiCl `1f5...fen.popsicls.com` except from its own pages or via redirections from `linker.popsicls.com`. The tenant server enforces this property by requiring any URL utilizing `1f5...fen.popsicls.com` to be appended with a *capability*, specific to this PoPSiCl, that only itself and `linker.popsicls.com` can obtain. This capability is implemented as a random, unguessable string that must be encoded as a query string in any URL using `1f5...fen.popsicls.com`, so that it is always transmitted under TLS protection. It is created by the PoPSiCl store when `1f5...fen.popsicls.com` is first registered and is returned in the URL that the user is invited to bookmark. Both the tenant server and `linker.popsicls.com` are then permitted to retrieve the capability (from a cloud-operated database) when needed, for the purposes of producing URLs containing that PoPSiCl.

There remains a cross-origin attack that a web server with which the client is interacting can mount, to infer the PoPSiCl the client uses to contact a tenant server, say `tenantA.com`, provided that the adversary controlling the web server can simultaneously monitor the traffic from the client to the cloud. If the web server directs the client to retrieve `https://linker.popsicls.com?tenantA.com/...` and then monitors traffic for an interaction with `linker.popsicls.com` and then a connection using a PoPSiCl in close temporal proximity, then the attacker can infer that the client uses this PoPSiCl for `tenantA.com`. Fortunately, the intervening interaction with the trusted `linker.popsicls.com` provides an opportunity to mitigate this attack. For example, a reasonable policy might be for `linker.popsicls.com` to redirect the request to `https://linker.popsicls.com?tenantA.com/...` to one that uses the client's PoPSiCl for `tenantA.com` only if the referrer site is trusted by `tenantA.com` and the client is also accessing the referrer site using a PoPSiCl. (Otherwise, `linker.popsicls.com` redirects the client to `tenantA.com`, sans PoPSiCl.) The referrer site can indicate compliance with this last condition to `linker.popsicls.com`

5 See https://www.w3.org/TR/referrer-policy/. As of this writing, the latest versions of Edge and Safari support an older draft of the referrer-policy specification, for which the referring policy should be set to `never`.

6 There are many ways to perform this test, e.g., by hyperlinking to an image and then testing the height of the image, which would typically differ depending on whether the image retrieval succeeded or failed.

by, say, appending the client's capability for the referring site to the referral `https://linker.popsicls.com?tenantA.com/...`, which `linker.popsicls.com` can check by looking up the capability and corresponding referrer in a database.

4.4 Design principles, revisited

In this section we revisit the principles outlined in Section 3 to describe how our design achieves them.

Security. Requirement **R1** is met by having the PoPSiCl store generate each PoPSiCl (specifically, the *str* part of *str*`.popsicls.com`) pseudorandomly. The tenant server to which this PoPSiCl corresponds is protected during the registration process by TLS (Section 4.1) and thereafter is accessible only to the cloud's SDN controller(s) (Section 4.2) in order to route connection traffic appropriately (and the tenant server itself, of course). The PoPSiCl is used by the client as any other domain name would be, and so it appears everywhere that the domain name would in the normal course of client communication—notably in DNS queries, the TLS SNI field, and the server TLS certificate.

Requirement **R2** is met by having the cloud's DNS server resolve the PoPSiCl to an IP address of an SDN controller in the datacenter hosting the corresponding tenant server. This reveals the datacenter (region) in which the tenant resides, but nothing else.

Requirement **R3** is met in a manner similar to that for **R1**, i.e., by the PoPSiCl store generating a new public key (and public-key certificate) for the tenant server per client who registers a PoPSiCl for that server. This certificate is then provided to the tenant server for use in TLS connection setups when accessed using the corresponding PoPSiCl.

Finally, requirement **R4** is met because the tenant server will accept a TLS connection to a PoPSiCl only from the client that registered it. Additionally, in the case of HTTP traffic, cross-origin attacks to indirectly query a PoPSiCl are prevented through refusing requests for URLs containing the PoPSiCl unless those URLs are appended with the PoPSiCl-specific capability that only the tenant server or `linker.popsicls.com` can obtain (Section 4.3).

Usability. Registration to obtain a PoPSiCl for a server is the only per-server procedure that a user must perform. In this step, the user visits a cloud-run website via HTTPS and enters the web server domain of interest. In return, it receives a PoPSiCl and a file containing a client public-key certificate and corresponding private key for use in TLS connections using this PoPSiCl. How the user employs this data is client-specific, but for a modern web browser, it might involve creating a bookmark using the PoPSiCl and importing the client certificate and private key into the browser. As such, we believe that our design meets requirement **R5**. It should also be noted that to register a PoPSiCl for a server, a user needs to learn that the server is hosted in the cloud. This point is discussed further in Section 7.

Changes to tenant servers are as follows. A tenant server OS must be modified to support the receipt of TCP connection states from the SDN controllers in the cloud (Section 4.2), and a tenant server also needs to be modified to refuse any connection using a PoPSiCl except from the client who registered it. A tenant web server must also check any URL using a PoPSiCl for the corresponding capability, as discussed in Section 4.3, to prevent cross-origin request forgeries that might deanonymize the PoPSiCl. Finally, the hyperlinks in the server's content must be changed to use relative URLs (for content at the same site) or the cloud-operated linker service (for content at another site), and the tenant server must set its referrer policy appropriately (Section 4.3). As discussed below, these changes can be introduced within VMs (or by a PaaS cloud operator) in such a way that content programming APIs need not be altered. We thus argue that requirement **R6** is also met.

5 Implementation

We realized our design in an OpenStack[7]-based IaaS cloud on top of the CloudLab[8] testbed. Each cloud computing node supports one or more tenant virtual machines (VMs) using the KVM hypervisor[9]. All tenant VMs are connected to the cloud network via Open vSwitch [33]. Open vSwitch is a software switch that runs in each hypervisor and bridges the virtual network interfaces of VMs on multiple computing nodes to a single layer-two network. Besides normal layer-two switching, Open vSwitch can be integrated with SDN controllers to support dynamic rule deployment and packet rewriting. Although our system is built upon Open-

7 https://www.openstack.org/
8 https://www.cloudlab.us/
9 http://www.linux-kvm.org/

Stack, KVM and Open vSwitch, we believe that our design could also integrate easily with other cloud implementations, such as Amazon EC2.

Below we detail our implementation of the three major components in our design: the PoPSiCl store, the SDN controller, and tenant web servers. In addition, a video demonstration of the user experience for our prototype can be found at http://www.cs.unc.edu/~qiuyu/popsicl/.

5.1 PoPSiCl store

PoPSiCl store is implemented on one of the web servers that are controlled by the cloud operator. Its frontend is a regular HTTPS web server offering a web interface for browsers, which accepts registration requests from any client browser. The backend of PoPSiCl store is implemented as a native component (400 lines of C++ code) that interacts with the frontend using a FastCGI protocol. Upon receiving a registration request, the frontend passes the request to the backend to complete the registration process.

Generating PoPSiCl and pseudo-address. The PoPSiCl store backend generates a PoPSiCl for the client. The newly created PoPSiCl is prefixed by a pseudorandom string *str* that meets the domain-name format requirements [25], and so the PoPSiCl takes the form *str*`.popsicls.com` where `popsicls.com` is the domain name of the cloud. The PoPSiCl store also chooses a pseudo-address for the PoPSiCl uniformly at random from a block of addresses for cloud SDN controller(s). Uniqueness is not required for the pseudo-address; different PoPSiCls may be associated with the same pseudo-address.

Generating certificates. The PoPSiCl store generates an X.509 server certificate `SvrCert` for the tenant server for which the client is registering a PoPSiCl, and an X.509 client certificate `ClntCert` for the client. In `SvrCert`, the issuer is the cloud operator, `popsicls.com`; the subject name is the generated PoPSiCl; and the public key comes from a key pair (2048-bit RSA) that is newly generated by the PoPSiCl store using OpenSSL[10]. `SvrCert` is signed by the cloud operator's private key, and so a chain of trust can be established when the cloud's certificate is trusted. (To do so, the cloud operator must first obtain a CA certificate that authorizes its private key to sign new certificates.) The issuer of `ClntCert` is the PoPSiCl and the subject name is a unique string that is derived from the PoPSiCl. The PoPSiCl store generates another RSA key pair for the `ClntCert` and signs the `ClntCert` using the private key that was created for the tenant server, so that trust in the `SvrCert` can be extended to `ClntCert`. The PoPSiCl store bundles each certificate and its corresponding private key into a single PKCS#12 format file.

Distributing registration data. The PoPSiCl store distributes registration data to parties as follows: It sends the PoPSiCl and the pseudo-address to the cloud DNS server (which stores them as a DNS record); the PoPSiCl and the domain name of the tenant server to the SDN controller; `SvrCert` and the corresponding private key to the tenant server; and the PoPSiCl (or to support HTTP access, a URL containing the PoPSiCl and a capability for it, embedded as a query string; see Section 4.3), `ClntCert`, and its corresponding private key to the client through the frontend interface.

5.2 Cloud SDN controller

In our implementation, all Open vSwitch instances are managed by the same SDN controller, a vSwitch controller we implemented in about 600 lines of C code. The SDN controller uses ovs-ofctl[11], a Linux command-line tool, to install and remove rules in each Open vSwitch.

As discussed in Section 4.2, every new TCP connection request using a PoPSiCl will be directed to the SDN controller, which completes the TCP handshake and then, after receiving a `ClientHello` message, hands off the TCP connection to the tenant server. To seamlessly transfer the TCP state from the SDN controller to the tenant server, our system uses a custom kernel extension (i.e., a kernel driver) to the Linux kernel (v4.2.0) to create a new user-kernel interface on both the SDN controller and each tenant VM. Userspace programs can exploit this interface (through `ioctl` system calls) to query or make changes to the internal TCP states. To facilitate TCP state migration, we also developed a userspace library that enables the SDN controller to obtain a copy of the TCP state information (sequence number, acknowledgment number, etc.) for a specific Linux socket descriptor. The state information is then sent through a long-lived TCP connection to the tenant server, which uses our helper library to create a

10 https://www.openssl.org/

11 http://openvswitch.org/support/dist-docs/ovs-ofctl.8.txt

new TCP socket with the specified TCP state and then resume the TCP session.

5.3 Tenant HTTP server

Key to our tenant web-server implementation is the support of virtual hosts—or "server blocks" in Nginx, on which we base our implementation. A virtual host is a website implemented by a single web server; importantly, one web server can implement multiple virtual hosts. In our implementation, each virtual host corresponds to a PoPSiCl and thus a client of the website.

Upon PoPSiCl registration, the tenant web server receives the registration data for the client (i.e., the PoPSiCl, the server certificate, and the corresponding private key) from the PoPSiCl store. The configuration file of the Nginx web server is updated automatically to reflect these registration data: a server block is added to the configuration file, with its `server_name` directive set to be the PoPSiCl and the `server_certificate` and `server_certificate_key` directives set as the file-system paths of the server certificate and private key, respectively. The `server_client_certificate` is set to be the path of the server certificate, as well, so that the virtual server to be created accepts only connections from clients who possess a certificate signed by the server's private key (see Section 5.1). Nginx supports server reconfiguration on-the-fly, and so a new virtual server for the PoPSiCl is created with the updated configuration file without any server down time.

We also adapted the Ruby on Rails web-content development framework (v2.2.2) to defend the cross-origin attack (see Section 4.3). Several macros, including `stylesheet_link_tag`, `javascript_include_tag`, and `link_to`, were modified to append the capability query string to same-origin URLs constructed via these macros. For each incoming HTTP request, the Ruby on Rails framework first checks for the capability query string. If the validation fails, a 404 error is returned.

6 Evaluation

In this section, we evaluate the impact of our design on performance of server interactions. More specifically, the goals of our evaluation are to demonstrate the impact of PoPSiCls on server-access latencies and throughputs, as well as the scalability of our design.

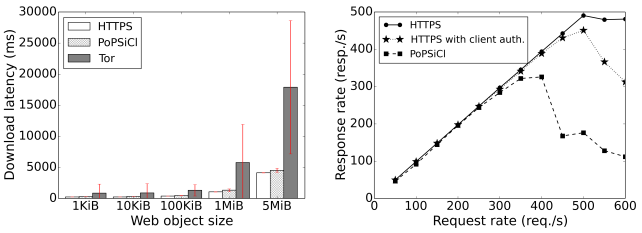
Our PoPSiCl-enabled OpenStack cloud was deployed in the CloudLab Wisconsin data center. For most experiments, we configured our cloud with three physical nodes: one for running OpenStack services (including DNS); one for running the PoPSiCl store and SDN controller; and another for running a tenant web server in a virtual machine. All nodes ran an Open vSwitch, though all rule installations described in Section 4.2 occurred on the controller machine. Each physical node was equipped with two Intel E5-2630 v3 8-core 2.40GHz CPUs, 128GiB of memory, and a Dual-port Intel X520-DA2 10Gb NIC. The web client was running on a desktop located in the UNC-Chapel Hill network. One experiment that compares our design with a proxy-based design has different settings, as will be discussed later.

6.1 Performance

In this section, we discuss the performance of our PoPSiCl implementation. We primarily compare to the performance of HTTPS alone (with no PoPSiCl and no client authentication) and, in one experiment, the performance of Tor. We caution the reader that our comparison with Tor is only somewhat fair: While Tor also provides server anonymity (a socalled "hidden service"), it does so against stronger adversaries than our design does; e.g., our design reveals the cloud datacenter within which a tenant server resides (Tor would hide this information) and additionally provides client-server unlinkability [34], to an extent. Still, we compare to Tor because it is the most widely used anonymous communication system today.

Web object download latency. In our first experiment, we measured the latency of downloading web objects. We chose Firefox as the web client for evaluating HTTPS and PoPSiCls, and the latest Tor browser (v5.5.5) for evaluating Tor. To fairly measure the extra overhead introduced by PoPSiCls, we restarted the web browser for each test, and so every web access was made through a new TCP and TLS connection, including the overheads of TLS client certificate authentication and TCP session hand-off from the SDN controller to the tenant server. We also restarted the browser between accesses when testing HTTPS. For the Tor browser, we measured the access latency after the Tor circuit had been built. We repeated the experiment 50 times per web-object size, which varied from 1KiB to 5MiB.

Average access latencies per object size are shown in Fig. 3a. As is clear from Fig. 3a, access using PoPSiCls is minimally more expensive than using HTTPS with



(a) Latency **(b)** Throughput

Fig. 3. Performance of PoPSiCl access

no client authentication, and is roughly 2.5–4× more efficient than accessing content using Tor. Moreover, the standard deviation of download latency for Tor is very considerable, indicating that for some Tor downloads, latencies were even worse than 4× more expensive.

The latency of web-object retrieval using PoPSiCls is robust as the request rate grows. For example, the average latency for retrieving a 10KiB object increases to 404ms when it is requested at a rate of 200 requests per second from distinct clients (not shown), an increase of less than 20% over the corresponding latency in Fig. 3a. Latency is also relatively unaffected by cross-site hyperlinking (see Section 4.3), especially for larger objects: accessing `linker.popsicls.com` involves another HTTPS connection but none of the mechanism in Section 4.2, and is unaffected by the retrieved objects' size. So, for example, the latency of retrieving a 1MiB object via `linker.popsicls.com` (not shown) is less than 11% larger than when retrieving it using a PoPSiCl directly.

Web access throughput. In this experiment, we measured the throughput of the web server. We used httperf[12], a popular web server benchmark tool, to measure the throughput. httperf can be used to dictate the rate of TCP requests and the number of HTTP requests per TCP connection, and it then will report the corresponding HTTP response rate. In our experiment, we measured and compared the server throughput when the server provides web access through HTTPS, HTTPS with client authentication, or a PoPSiCl. We scaled the request rate from 50 requests/s to 600 requests/s, with one HTTP request/response pair per TCP connection. The size of the requested web object was 1KiB. For each request rate and each condition, we took 10 samples of the response rate and calculated their mean.

As can be seen from Fig. 3b, before the web server reached its sustainable throughput, its response rate kept pace with the request rate. After the web server reached its limit, the response rate dropped as the request rate increased further. The maximum throughputs of HTTPS, HTTPS with client authentication, and PoPSiCl were 490.5, 450.9, and 325.8 responses/s, respectively. Compared with HTTPS, PoPSiCl induced a 33.5% throughput decrease. The throughput bottleneck in these tests was the switch rule installation procedure (see Section 4.2), which increasingly encountered failures when the request rate grew.

6.2 Scalability

Our design poses several potential scalability pitfalls. In this section we evaluate these elements of our design.

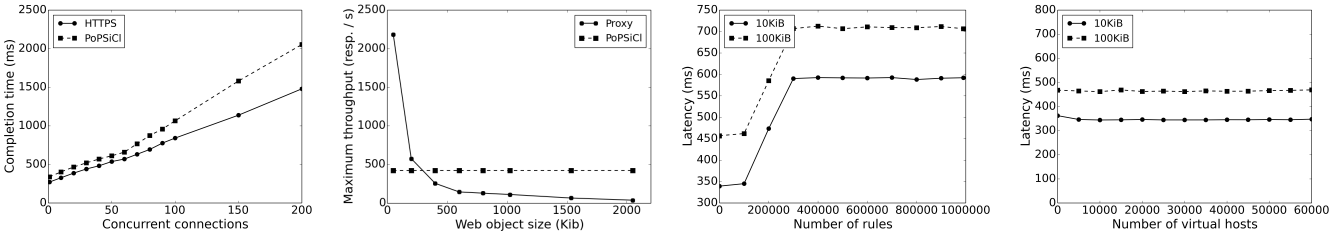
SDN rule installation and TCP handoff. As discussed in Section 4.2, our design results in the installation of two rules per PoPSiCl-based TCP connection through a switch (or one rule into each of two switches), followed by the transfer of TCP state from the SDN controller to the tenant server and then the adjustment of one of the rules previously installed for this connection. These steps slow the connection setup to a tenant server, and so in our first experiment we evaluated the impact of these overheads, as the number of concurrent connection setups grows.

We used cURL (https://curl.haxx.se/) as the web client for both HTTPS and PoPSiCl access. In each test, we launched concurrent cURL processes; each process opened a connection (HTTPS in one type of test, or using a PoPSiCl in the other type), retrieved a web object from the tenant server, and then terminated its connection. We measured the completion time of *all* connections, and plotted this completion time as a function of the number of processes (and connections) launched. The size of the web object in this experiment was 10KiB.

Fig. 4a shows the result of these experiments, where each point is the average of the results from ten runs. As can be seen there, the completion time for the PoPSiCl-based connections was at most 1.4× the completion time for the same number of concurrent HTTPS connections. Our SDN controller and tenant web server implementations already perform the steps for each connection concurrently, though otherwise the implementations are relatively unoptimized.

The need for TCP handoff. The motivation for SDN rule installation and TCP handoff steps detailed in Section 4.2 and evaluated above becomes evident when comparing our design to a proxy-based alternative. In

12 http://www.labs.hpe.com/research/linux/httperf/



(a) Concurrent connections **(b)** Per-connection volume **(c)** Open vSwitch rules **(d)** Virtual hosts

Fig. 4. Scalability of PoPSiCl access along several dimensions

this alternative, each PoPSiCl is resolved to a *pseudo-IP* that is the address of a proxy that completes the TCP connection with the client and then learns which tenant server the client wants to contact from the `ClientHello` SNI field (like our SDN controller does). The proxy then opens another TCP connection with the tenant server and relays traffic between the client and server, without handing off that connection to the server.

We built this proxy alternative and evaluated its throughput. To ensure that the clients and servers were not bottlenecks, we set up three httperf clients connecting to three tenant web servers, each on its own physical node. As can be seen in Fig. 4b, the proxy alternative outperformed ours when the retrieved web object was small. But as the web-object size increased, the maximum throughput of the proxy decayed dramatically, owing to the need for the proxy to relay each packet of the TCP connection. In contrast, our design hands off the TCP connection to the tenant server at the start of the connection. Aurelius et al. [2] found that 70% of Flash video flows from various services transferred at least 1MiB data. If such streaming services were deployed in the cloud, a proxy could be easily saturated.

SDN switch rules. By default, one million rules can be installed simultaneously in a vSwitch, which would thus accommodate up to a half million concurrent client TCP connections in support of PoPSiCl-based server accesses. This introduces two scalability concerns.

First, since some clients (notably browsers) tend to open multiple connections per server access, a vSwitch's default rule capacity might accommodate far fewer than a half million concurrent clients using PoPSiCls. However, this concern can be addressed by increasing the rule capacity of a vSwitch and also load-balancing rule installation across the potentially multiple vSwitches that a client's connections traverse. Indeed, additional vSwitches could be added in the cloud—even elastically—to boost rule capacity, if needed.

Second, deploying several hundred thousand rules to a vSwitch could slow the process by which the vSwitch matches incoming packets to rules, and so we conducted experiments to evaluate this performance degradation. Fig. 4c shows the average latency (over 200 trials) suffered by a client using a PoPSiCl to open a connection and retrieve an object of either 10KiB or 100KiB from a tenant web server, when the switch starts with the number of rules indicated on the horizontal axis. The performance clearly shows two "levels" of latency per object size, indicating internal switch data structures that permit searching few rules quickly and more rules somewhat more slowly (but very scalably). Still, the performance impact with nearly 1000000 rules is less than 2×. It is conceivable that engineering a switch specifically to accommodate the usage that our design imposes might further reduce this degradation.

Virtual hosts. As discussed in Section 5.3, our implementation deploys a virtual host to a tenant web server per client that registers a PoPSiCl for it. This virtual host is associated with the PoPSiCl and the server certificate that the web server should use in TLS connections using that PoPSiCl (and that doubles as the certificate for verifying the client certificate). Admittedly this is perhaps an abuse of the virtual-host mechanism, which presumably was not designed to accommodate a virtual host per web-server client—a popular web server could have millions of virtual hosts.

To get a sense for the scalability limitations that the existing Nginx virtual-host design would impose, Fig. 4d shows the degradation in responsiveness of the tenant web server as a function of the number of virtual hosts installed. These tests were conducted by first creating the number of virtual hosts on the horizontal axis and then connecting to the server using a PoPSiCl that matches one of these virtual hosts, to retrieve an object of either 10KiB or 100KiB.

Fig. 4d shows the performance impact of the number of virtual hosts set up before the connection. Each point is an average over 200 trials. The number of virtual hosts had no impact on the response latency for the numbers we tested. However, the memory consumption of the server with 60000 virtual hosts approached 2GiB. As in the case of vSwitch rules above, we anticipate that this scalability limitation could be addressed with a virtual-host design that anticipates our proposed usage, e.g., relieving this memory pressure by writing the data for inactive virtual hosts to stable storage.

DNS entries. Each PoPSiCl registration results in the creation of a new DNS record for the PoPSiCl, mapping the PoPSiCl to the addresses of SDN controllers in the datacenter where the tenant resides. The number of DNS records could thus grow large, if the use of PoPSiCls became popular. We have not evaluated the potential performance impact of this growth on DNS resolutions, however, since backing the DNS server with a simple database for these records would support ample storage and fast access. Going further, the *str* portion of a PoPSiCl *str*`.popsicls.com` could be computed to be the encryption (using a chosen-ciphertext-secure scheme) of the IP address(es) to which it should be mapped, using a key that the DNS server holds. The DNS server would then not need to store the mapping, but upon receiving a request to resolve *str*`.popsicls.com` could instead decrypt *str* and return the result.

7 Limitations

Our current implementation of an architecture to support PoPSiCls is limited in at least the following senses.

Scope of defense. Our design provides PoPSiCls only for servers hosted in a cloud that supports their use. While major cloud operators host substantial numbers of web servers (e.g., [19]), for example, obviously numerous web servers do not fall into this category, as well. Related to this limitation is that a user must know or be directed to the cloud that hosts a server in order to register a PoPSiCl for it. This information could be disseminated by the cloud, provided that the cloud is trusted to not claim to host a web server that it does not (as a major cloud operator might be); by a link to the appropriate PoPSiCl store from the web server itself, so that a user could leverage one access to the server to be able to access it using a PoPSiCl subsequently; or by myriad other means (e.g., social media).

Censorship. We assume a trustworthy cloud operator that is motivated to help tenants protect the privacy of their customers from an attacker who might try to observe their customers connecting to them. This assumption is arguably stronger than most (though not all, c.f., [40]) threat models considered in works addressing censorship resistance. Indeed, in the context of censorship by governments, clouds are often *used* by activists to circumvent censors, but this is typically done without the cloud operator's consent [12]. Major clouds have admittedly shown little cooperation for resisting censorship by governments (e.g., [18]), preferring instead to accommodate censorship for business reasons. Moreover, the PoPSiCl store is vulnerable to being blocked. As such, our design seems unlikely to be deployed specifically to resist government censorship, but it still offers an opportunity for a cloud to actively contribute to privacy for customers of tenant servers to which censors permit access (even while disallowing access to servers that censors forbid).

Going further, it is conceivable that our design offers an attractive balance between social responsibility and client privacy by assuming a trusted cloud that retains the ability to censor servers. For example, evidence suggests that a majority of Tor "hidden services" are criminally oriented and the most frequently requested sites host child abuse imagery [30]. Such abusive sites could be shut down by the operator once it is informed of the abusive content.

Traffic analysis. As discussed previously, our basic design (Section 4) leaves traffic analysis to the tenant server to address, should it choose to. While we have made initial steps to support per-connection traffic-analysis defense (Appendix A), that defense does not immediately address traffic analysis based on aggregates of connections, e.g., the number of servers to which connections are made or the relative timings of these connections. Defending against this type of attack remains a very active area of research independent of our proposal (e.g., [13, 14, 31, 39]).

OS compatibility. Our current implementation of TCP hand-off (Section 5) requires that both the controller and the tenant server run on the same Linux OS kernel version. We also disabled the TCP timestamp and selective acknowledgment (SACK) options to facilitate TCP state migration. In future work, we aim to support TCP hand-off across different TCP stack implementations so that PoPSiCls will be suitable for more heterogeneous deployments.

8 Conclusions

In this paper we presented PoPSiCls, which are personalized pseudonyms for servers that a client can use like regular, long-lived server domain names to open TLS connections to those servers. We described a design and implementation for PoPSiCls that leverages trust in a cloud to implement PoPSiCls for tenant servers that it hosts. PoPSiCls have several desirable security and usability properties in our threat model. First, TLS connections established using a PoPSiCl exhibit identifiers (domain names, IP addresses, and server public keys) to the attacker that he cannot correlate against those exhibited in connections involving other clients or other tenant servers. Second, the burdens placed on various parties in our implementation of PoPSiCls correspond to their levels of technical capabilities: cloud operators bear the most (which, based on our experience, is still minor); changes to tenant web servers can be hidden from web-content developers by packaging these changes within VMs (in an IaaS scenario) or the cloud platform (in a PaaS scenario); and tenants' clients need only suffer minor changes to the user experience and no changes to client software (in the case of web browsers). The last is important since client users often lack the permissions or willingness to modify their platforms (e.g., due to the vulnerabilities that those modifications can introduce [9, 35, 38]). Our evaluations show that performance for PoPSiCl access to tenant servers is competitive with baseline HTTPS and scales well as PoPSiCl use grows. We thus believe that PoPSiCls provide a promising opportunity for cloud operators to improve privacy for its tenants' clients. We also illustrated extensions of our design to permit the implementation of defenses against traffic analysis with no client-side changes.

Acknowledgements

This work was supported in part by NSF grant 1330599.

References

[1] W. Almesberger. TCP connection passing. In *Linux Symposium*, volume 1, July 2004.

[2] A. Aurelius, C. Lagerstedt, and M. Kihl. Streaming media over the Internet: Flow based analysis in live access networks. In *Broadband Multimedia Systems and Broadcasting, 2011 IEEE International Symposium on*, 2011.

[3] J. Boyan. The Anonymizer: Protecting user privacy on the web. *Computer-Mediated Communication Magazine*, 4(9), Sept. 1997.

[4] C. Brubaker, A. Houmansadr, and V. Shmatikov. CloudTransport: Using cloud storage for censorship-resistant networking. In *Privacy Enhancing Technologies, 14th International Symposium*, volume 8555 of *Lecture Notes in Computer Science*. July 2014.

[5] Y. Cao, S. Li, and E. Williams. (cross-)browser fingerprinting via OS and hardware level features. In *ISOC Network and Distributed System Security Symposium*, Feb. 2017.

[6] D. Chaum. The dining cryptographers problem: Unconditional sender and recipient unlinkability. *Journal of Cryptology*, 1(1), 1988.

[7] C. Chen, D. E. Asoni, D. Barrera, G. Danezis, and A. Perrig. Hornet: High-speed onion routing at the network layer. In *22nd ACM Conference on Computer and Communications Security*, pages 1441–1454, 2015.

[8] C. Chen and A. Perrig. Phi: Path-hidden lightweight anonymity protocol at network layer. *Proceedings on Privacy Enhancing Technologies*, 2017(1):100–117, 2017.

[9] L. Constantin. Antivirus software could make your company more vulnerable. *PCWorld*, Jan. 2016. http://goo.gl/Amju2A.

[10] S. Coull, M. P. Collins, C. V. Wright, F. Monrose, and M. K. Reiter. On web browsing privacy in anonymized NetFlows. In *16th USENIX Security Symposium*, Aug. 2007.

[11] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation Onion Router. In *13th USENIX Security Symposium*, Aug. 2004.

[12] E. Dou and A. Barr. U.S. cloud providers face backlash from China's censors. *The Wall Street Journal*, 16 March 2015.

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

[14] K. P. Dyer, S. E. Coull, and T. Shrimpton. Marionette: A programmable network-traffic obfuscation system. In *24th USENIX Security Symposium*, 2015.

[15] N. Feamster, M. Balazinska, W. Wang, H. Balakrishnan, and D. Karger. Thwarting web censorship with untrusted messenger discovery. In *3rd International Workshop on Privacy Enhancing Technologies*, 2003.

[16] D. Fifield, C. Lan, R. Hynes, P. Wegmann, and V. Paxson. Blocking-resistant communication through domain fronting. *Proceedings on Privacy Enhancing Technologies*, 2, 2015.

[17] Fyodor. Remote OS detection via TCP/IP stack fingerprinting. https://nmap.org/nmap-fingerprinting-article.txt, Oct. 1998.

[18] D. Goldman. Google: The reluctant censor of the Internet. *CNN Money*, 4 January 2015.

[19] K. He, A. Fisher, L. Wang, A. Gember, A. Akella, and T. Ristenpart. Next stop, the cloud: Understanding modern web service deployment in EC2 and Azure. In *Internet Measurement Conference*, Oct. 2013.

[20] J. Holowczak and A. Houmansadr. CacheBrowser: Bypassing Chinese censorship without proxies using cached content. In *22nd ACM Conference on Computer and Communications Security*, Oct. 2015.

[21] H. C. Hsiao, T. H. J. Kim, A. Perrig, A. Yamada, S. C. Nelson, M. Gruteser, and W. Meng. Lap: Lightweight anonymity and privacy. In *2012 IEEE Symposium on Security and Privacy*, pages 506–520, 2012.

[22] N. Jones, M. Arye, J. Cesareo, and M. J. Freedman. Hiding amongst the clouds: A proposal for cloud-based Onion Routing. In *Free and Open Communications on the Internet*. USENIX, 2011.

[23] M. Juarez, S. Afroz, G. Acar, C. Diaz, and R. Greenstadt. A critical evaluation of website fingerprinting attacks. In *ACM Conference on Computer and Communications Security*, 2014.

[24] M. Liberatore and B. N. Levine. Inferring the source of encrypted HTTP connections. In *13th ACM Conference on Computer and Communications Security*, Oct. 2006.

[25] P. Mockapetris. Domain names – implementation and specification. RFC 1035, RFC Editor, Nov. 1987. http://www.rfc-editor.org/rfc/rfc1035.txt.

[26] R. Moore. TLS Prober – an SSL/TLS server fingerprinting tool. https://github.com/WestpointLtd/tls_prober/blob/master/doc/tls_prober.md, Mar. 2015.

[27] R. Mortier, A. Madhavapeddy, T. Hong, D. Murray, and M. Schwarzkopf. Using dust clouds to enhance anonymous communication. In *18th International Workshop on Security Protocols*, 2014.

[28] N. Nikiforakis, A. Kapravelos, W. Joosen, C. Kruegel, F. Peissens, and G. Vigna. Cookieless monster: Exploring the ecosystem of web-based device fingerprinting. In *IEEE Symposium on Security and Privacy*, May 2013.

[29] J. C. Norte. Advanced Tor browser fingerprinting. http://jcarlosnorte.com/security/2016/03/06/advanced-tor-browser-fingerprinting.html, Mar. 2016.

[30] G. Owen and N. Savage. The Tor dark net. No. 20, Global Commission on Internet Governance Paper Series, Sept. 2015.

[31] A. Panchenko, F. Lanze, A. Zinnen, M. Henze, J. Pennekamp, K. Wehrle, and T. Engel. Website fingerprinting at Internet scale. In *ISOC Network and Distributed System Symposium*, Feb. 2016.

[32] A. Parsovs. Practical issues with TLS client certificate authentication. In *ISOC Network and Distributed System Security Symposium*, Feb. 2014.

[33] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado. The design and implementation of open vswitch. In *12th USENIX Symposium on Networked Systems Design and Implementation*, May 2015.

[34] A. Pfitzmann and M. Waidner. Networks without user observability. *Computers and Security*, 6(2), Apr. 1987.

[35] S. Ragan. Hola VPN client vulnerabilities put millions of users at risk. *CSO*, Mar. 2015. http://goo.gl/yZnkzF.

[36] J. Ruderman. Same-origin policy. https://developer.mozilla.org/en-US/docs/Web/Security/Same-origin_policy, Mar. 2016.

[37] J. Sankey and M. Wright. Dovetail: Stronger anonymity in next-generation internet routing. *Proceedings on Privacy Enhancing Technologies*, pages 283–303, 2014.

[38] L. Seltzer. Research shows antivirus products vulnerable to attack. *ZDNet*, Feb. 2016. http://goo.gl/9kbgqX.

[39] T. Wang, X. Cai, R. Johnson, and I. Goldberg. Effective attacks and provable defenses for website fingerprinting. In *23rd USENIX Security Symposium*, Aug. 2014.

[40] E. Wustrow, S. Wolchok, I. Goldberg, and J. A. Halderman. Telex: Anticensorship in the network infrastructure. In *20th USENIX Security Symposium*, Aug. 2011.

[41] H. Zolfaghari and A. Houmansadr. Practical censorship evasion leveraging content delivery networks. In *ACM Conference on Computer and Communications Security*, Oct. 2016.

A Traffic Analysis

As discussed in Section 3.1, PoPSiCls hide the identity of the servers contacted by clients from being directly disclosed to an attacker between the clients and the cloud. However, they do not hide connection characteristics from the attacker, such as the number or sizes of packets in each direction, connection duration, etc. What can be inferred from these features has long been studied and debated, particularly in the context of traffic directed through anonymizing proxies (e.g., [13, 16, 23, 24, 31, 39]) and, similarly, traffic logs in which payloads have been removed (e.g., [10]).

Our design so far has left this issue to tenant servers to address, should they choose to. However, in this section we summarize an approach that we have developed for HTTP servers that can be used to implement some proposed defenses. In keeping with **R6**, this defense can be deployed with very modest adaptations to web content. Also, the proposed approach relies on client-side Javascript and does not require the client user to install new software (in keeping with **R5**). We stress that our goal here is not to innovate in terms of new per-connection defenses, but instead to provide a framework in which such defenses can be implemented.

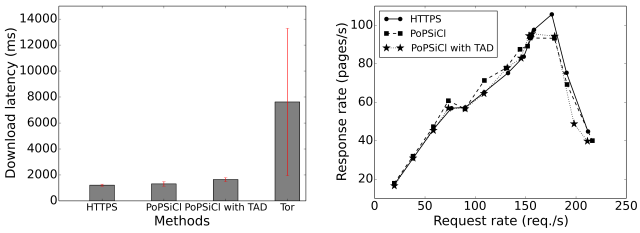
A.1 Design

The key enabler for these defenses without requiring modifications to the client platform is Javascript

blobs[13], which provide a way for client-side Javascript to construct file-like objects and pass them to APIs that expect URLs. This functionality permits a tenant server to serve Javascript to the client browser that customizes how objects are retrieved from the server, and then post-processes the retrieved contents and provides them to the browser (as blobs) for rendering. So, for example, the client-side Javascript could replace the retrieval of one web object with that of many smaller objects and then reassemble the original object before providing it to the browser.

We have prototyped this approach in a simple adaptation to the Ruby on Rails web-content development framework. This adaptation prepends a Javascript script to every HTML file; the script takes control of retrieving the objects that otherwise would have been hyperlinked directly in that page. This script patches the `XMLHTTPRequest` class to remove padding and reassemble the original object from smaller pieces of that object. When the `send` method of a `XMLHTTPRequest` instance is called, instead of sending a single HTTP request to the server, it sends several HTTP requests. The URL of each request contains the original URL and also an extra query string that informs the server of the number of pieces into which to split the object at that URL and the index of the piece to return in response. After receiving the first such request, the modified Ruby on Rails framework splits the web object into the requested number of pieces; the piece at the index indicated in each request is then returned as the request's response, after appending padding and prepending the length of that padding to the piece (to allow for padding removal). After the `XMLHTTPRequest` instance gets all the pieces of the original object, it reads a length field at the beginning of each piece, strips the piece of any content (i.e., padding) that extends past that length indicator, reassembles the original object, creates a blob containing the resulting object, and submits it to the browser for rendering. Rewriting the `XMLHTTPRequest` class ensures that even objects retrieved by other Javascript scripts in the page will be split and reassembled in this way.

While providing a foothold for addressing traffic analysis based on the features of individual object retrievals, this design does not interfere with hints that might be available to the attacker based on his viewing multiple connections in aggregate, such as the number of servers that are accessed simultaneously. Developing a similar foothold for obfuscating these features is a topic of ongoing work.



(a) Latency **(b)** Throughput

Fig. 5. Performance of PoPSiCl with traffic-analysis defense (TAD), for retrieving a web page hyperlinking to thirteen objects of total size 474.3KiB

A.2 Evaluation

To evaluate the performance impact of this form of traffic-analysis defense, we modified an open-source blog site[14], which is written in Ruby on Rails, to adopt it. In terms of modifications to the application-specific web content itself, we needed to modify only one line of embedded Ruby code in two templates; the rest of the implementation is embedded in the Ruby on Rails framework, hidden from the web-content developer. In our implementation, the Javascript code chooses uniformly from among retrieving a web object in one, two, or three pieces, and each piece is padded to make its size a multiple of 512 bytes. The root page of this website hyperlinks to thirteen web objects of total size 474.3KiB. We evaluated our traffic analysis defense approach, in terms of latency and throughput, by visiting this root page repeatedly, being sure to clear the browser cache between retrievals.

Latency. We chose the Firefox browser for evaluating the latency of root-page retrievals via HTTPS, PoPSiCls, and PoPSiCls with traffic-analysis defense (TAD) implemented as above, and the Tor browser (v5.5.5). The latency was measured as the time of downloading and rendering the whole web page. The experiment was repeated 20 times for each setting. As can be seen from Fig. 5a, the latency of accessing the page using PoPSiCls with traffic-analysis defense is only 1.35× that of HTTPS, while the latency of Tor is 6.3× that of HTTPS.

13 http://developer.mozilla.org/en-US/docs/Web/API/Blob

14 https://github.com/natew/obtvse

Throughput. To measure throughput of root-page retrievals, we chose a headless browser, PhantomJS[15], as the web client. Though lacking a graphical user interface, PhantomJS still parses HTML documents, runs Javascript code, and downloads hyperlinked web objects in a web page. (The httperf tool, used in the throughput experiments of Section 6, does not parse returned HTML.) We wrote a script to spawn PhantomJS processes in the background, and each PhantomJS instance was scripted to visit the root page once and then terminate. By adjusting the spawning rate, we adjusted the web-page request rate, and the response rate was measured as the number of root-page retrievals completed per second. To ensure that the client was not the bottleneck in these experiments, we ran the client in a physical node with 32 cores and 128GiB memory. As can be seen in Fig. 5b, the throughput of root-page retrievals using PoPSiCls was approximately the same with or without traffic-analysis defense, and only slightly lower than the throughput using HTTPS alone.

15 http://phantomjs.org