OCDN: Oblivious Content Distribution Networks

Abstract: As publishers increasingly use Content Distribution Networks (CDNs) to distribute content across geographically diverse networks, CDNs themselves are becoming unwitting targets of requests for both access to user data and content takedown. From copyright infringement to moderation of online speech, CDNs have found themselves at the forefront of many recent legal quandaries. At the heart of the tension, however, is the fact that CDNs have rich information both about the content they are serving and the users who are requesting that content. This paper offers a technical contribution that is relevant to this ongoing tension with the design of an Oblivious CDN (OCDN); the system is both compatible with the existing Web ecosystem of publishers and clients and hides from the CDN both the content it is serving and the users who are requesting that content. OCDN is compatible with the way that publishers currently host content on CDNs. Using OCDN, publishers can use multiple CDNs to publish content; clients retrieve content through a peer-to-peer anonymizing network of proxies. Our prototype implementation and evaluation of OCDN show that the system can obfuscate both content and clients from the CDN operator while still delivering content with good performance.

Keywords: privacy, content distribution networks

DOI Editor to enter DOI
Received ..; revised ..; accepted ...

1 Introduction

As Content Distribution Networks (CDNs) host an increasing amount of content, they are fast becoming targets of requests for information about their content and who is requesting it, as well as requests for takedown of material ranging from alleged copyright violations to offensive content. The shifting legal and political landscape suggests that CDNs may soon face liability for the content that they host. For example, the European Union has been considering laws that would remove safe harbor protection on copyright infringement for online service providers if they do not deploy tools that can automatically inspect and remove infringing content [17]. In the United States, various laws under consideration threaten aspects of Section 230 of the Communications Decency Act [2], which protects CDNs from federal criminal liability for the content that they host. Tussles surrounding speech, from copyright violations to hate speech, are currently being addressed in the courts, yet the legal outcomes remain ambiguous and uncertain, sometimes with courts issuing opposing rulings in different cases. Regardless of these outcomes, however, CDNs are increasingly in need of *technical* protections against the liability they might face as a result of content that they (perhaps unwittingly) serve.

Towards this end, we design and implement a system that allows clients to retrieve web objects from one or more CDNs, while preventing the CDNs from learning either (1) the content that is stored on the cache nodes; or (2) the content that clients request, and the clients that request it. We call this system an *oblivious CDN* (OCDN), because the CDN is oblivious to both the content it is storing and the content that clients request.

OCDN allows clients to request objects with identifiers that are encrypted with a key that is shared by an open proxy and the origin server that publishes content to cache nodes, but is not known to any of the CDN cache nodes. To do so, the origin server publishes encrypted content, which the CDN cache nodes subsequently share with one or more proxies that are responsible for routing requests for objects corresponding to that URL. A client forwards a request for content through other OCDN clients in a way that prevents both other clients and the CDN from learning the client identity or requested content. Using OCDN requires only minimal modification to existing clients; clients can also configure aspects of the system to trade off performance for privacy.

Ensuring that the CDN operator never learns information about either (1) what content is being stored on its cache nodes or (2) which objects individual clients are requesting is challenging, due to the many possible inference attacks that a CDN might be able to mount.

CC BY-NC-ND

For example, previous work has shown that even when web content is encrypted, the retrieval of a collection of objects of various sizes can yield information about the web page that was being retrieved [7, 37]. Similarly, URLs can often be inferred from relative popularity in a distribution of web requests, even when the requests themselves are encrypted. Additionally, the OCDN design assumes a strong attack model (Section 3), whereby an adversary can request logs from the CDN, interact with OCDN as a client, a proxy, or a publisher, and mount coordinated attacks that depend on multiple such capabilities. Our threat model does not include active attempts to disrupt the system (*e.g.*, blocking access to parts of the system, mounting denial of service attacks), but it includes any type of attack that involves observing traffic and even directly interacting with the system as a client, proxy, or publisher.

The rest of this paper is structured as follows. We provide a brief background of CDNs, privacy implications, and legal questions in Section 2. We discuss the threat model in Section 3. In Section 4, we detail the design of OCDN. We describe our process for obfuscating requests and content in Section 5. Our prototype implementation is described in Section 6. Section 7 analyzes how OCDN defends against threats from our adversary. Section 8 studies the performance implications of the tradeoffs between performance and privacy, as well as how OCDN performs relative to a conventional CDN. Section 9 describes various limitations and possible avenues for future work, Section 10 discusses related work, and Section 11 concludes.

2 Background

We now outline how a CDN typically operates, including what information it has access to. We also discuss some of the ongoing legal questions that CDNs currently face.

2.1 Content Distribution Networks

CDNs provide content caching as a service to content publishers. A content publisher may wish to use a CDN provider for several reasons:

- CDNs cache content in geographically distributed locations, which allows for localized data centers, faster download speeds, and reduces the load on the content publisher's server.

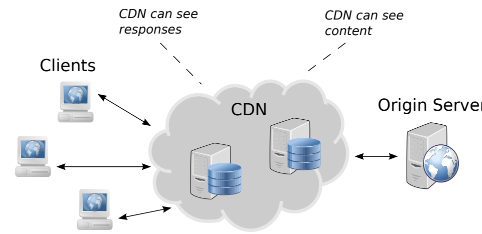

Fig. 1. The relationships between clients, the CDN, and content publishers in CDNs today.

- CDNs typically provide usage analytics, which can help a content publisher get a better understanding of usage as compared to the publisher's understanding without a CDN.
- CDNs provide a high capacity infrastructure, and therefore provide higher availability, lower network latency, and lower packet loss.
- CDNs' data centers have high bandwidth, which allows them to handle and mitigate DDoS attacks better than the content publisher's server.

CDN providers usually have a large number of edge servers on which content is cached; for example, Akamai has more than 216,000 servers in over 120 countries around the world [4]. Having more edge servers in more locations increases the probability that a cache is geographically close to a client, and could reduce the end-to-end latency, as well as the likelihood of some kinds of attacks, such as BGP (Border Gateway Protocol) hijacking. This is evident when a client requests a web page; the closest edge server to the client that contains the content is identified and the content is served from that edge server. Most often, this edge server is geographically closer to the client than the content publisher's server, thus increasing the speed in which the client receives the content. If the requested page's content is not in one of the CDN's caches, then the request is forwarded to the content publisher's server; the CDN caches the response and returns the content to the client.

2.2 What CDNs Can See

Because the CDN interacts with both content publishers and clients, as shown in Figure 1, it is in a unique position to learn an enormous amount of information. CDN providers know information about all clients who access data stored at the CDN, information about all content publishers that cache content at CDN edge servers, and information about the content itself.

Content. CDNs typically have access to all content that they distribute, as well as the corresponding URL. First, the CDN must use the URL, which is not encrypted or hidden, to locate and serve the content. Therefore, the CDN already knows what content is stored in its caches. Because CDNs provide analytics to content publishers, they keep track of cache hit rates, and how often content is accessed. The CDN not only knows about the content identifier; it also also has access to the plaintext content in order to perform optimizations on the content to increase performance (*e.g.*, minimizing CSS, HTML, and JavaScript files, which reduces file sizes by about 20%). They can also inspect content to upgrade a connection to HTTPS; we discuss how OCDN handles these types of optimizations in Section 9. In addition, requesting content via HTTPS does not hide any information from the CDN as the CDN terminates TLS connections on behalf of the content publisher. This means that not only does the CDN know the content, the content identifier, but also it knows public and private keys, as well as certificates associated with the content it caches.

Client information. Clients retrieve content directly from the CDN's edge servers. These requests reveal information about the client's location and what the client is accessing. CDNs can also see each client's cross-site browsing patterns: CDNs host content for many different publishers, which allows them to see content requests for content published by different publishers. Akamai caches enough content around the world to see up to 30% of global Internet traffic [3]. The implications of a CDN having access to this much information was evident when Cloudflare went public with the National Security Letters they had received [13]; these National Security Letters demanded information collected by the CDN and also included a gag order, which prohibits the CDN from publicly announcing the information request.

Content publisher information. A CDN must know information about their customers, the content publishers; the CDN keeps track of who the content publisher is and what the publisher's content is. The combination of the CDN seeing all content in plaintext and the content's linkability with the publisher, gives the CDN even more power. Additionally, as mentioned previously, the CDN often holds the publisher's keys (including the private key), and the publisher's certificates. This has led to doubts about the integrity of content because a CDN can impersonate the publisher from the client's point of view [30].

2.3 Open Legal Questions

Various parties are battling in the courts over cases that pertain to user data requests and intermediary liability. Intermediary liability would impose criminal liability on an Internet platform (or a CDN) for the content it provides on behalf of its customers or users. In this section, we highlight some of these cases, which point to a key problem that CDNs face: by knowing all the content that they distribute, CDNs may be burdened with the legal responsibility for the actions of their customers and clients.

User Data Requests. There are numerous open questions in the legal realm regarding which government can request data stored in different countries, which has led to much uncertainty. A series of recent events have illustrated this uncertainty. In the struggle over government access to user data, cases such as *Microsoft vs. United States* (often known as the "Microsoft Ireland Case") concerns whether the United States Government should have access to data about U.S. citizens stored abroad, given that Microsoft is a U.S. corporation.

Additionally, there have been user data requests asked of CDNs. The Cloudflare CDN has been required to share data with FBI [13]; similarly, leaked NSA documents showed that the government agency "collected information 'by exploiting inherent weaknesses in Facebook's security model' through its use of the popular Akamai content delivery network" [21].

Intermediary Liability. More recently, questions on intermediary liability have been in the spotlight. For example, many groups, including the Recording Industry Association of America (RIAA) and the Motion Picture Association of America (MPAA), have targeted CDNs with takedown notices for content that allegedly infringes on copyright, trademarks, and patent rights; CDNs are a more convenient target of these takedown notices than the content provider because oftentimes the content provider is either located in a jurisdiction where it is difficult to enforce the takedown, or it is difficult to determine the owner of the content [11, 12]. Although Section 230 of the Communications Decency Act protects intermediaries, such as CDNs, from being held liable for the content they distribute, there have been cases where CDNs are forced to remove content. This happened in 2015 involving the RIAA, Cloudflare, and Grooveshark [1]. In 2017, a district court ruled that

Cloudflare is not protected from anti-piracy injunctions by the Digital Millennium Copyright Act (DMCA) [16].

The role of a CDN as an intermediary has also come into question in new and currently pending legislation, including a 2017 German hate speech law [14] and a bill proposed by the U.S. Senate called Stop Enabling Sex Traffickers Act (SESTA) [5]. Both laws raise the spectre of potentially holding CDNs liable for the content that they distribute, despite the CDN not being a party in the content publishing.

3 Threat Model and Goals

In this section, we describe our threat model, outline the capabilities of the attacker, and introduce the design goals and protections that OCDN provides.

3.1 Threat Model

Our threat model is a powerful adversary who has a variety of capabilities, including both surveilling activities and joining the system in various capacities. We assume that an adversary can gain access to the CDNs logs, which typically contains client IP addresses and URLs for each request. Additionally, the adversary could join OCDN as either a client or any number of clients, or as an arbitrary number of proxies. The adversary could also act as an origin server (a content publisher). We also assume that the adversary can coordiante several of these actions to learn more information. For example, the adversary could join as a client and proxy, and request access to the CDN's logs to observe how its own requests are obfuscated. Additionally, the adversary can perform actions, such as generating requests as a client, or creating content as a content publisher. The goal of this type of adversary is to learn about the content being stored at the CDN and/or learn about which clients are accessing which content.

The strong adversary that we consider has seen some precedent in practice: for example, governments have demanded access to CDNs' data [13]. Although one possible adversary is a government requesting logs from the CDN, the government could also be colluding with a CDN; the CDN operator might even be an adversary.

Our design does not defend against an attacker who attempts to actively disrupt or block access to the system, such as by actively modifying content, disrupting communications (*e.g.*, through denial of service), or blocking access, content, or requests. Prior work on securing CDNs has introduced methods to handle an actively malicious adversary by preserving the integrity of content stored on CDN cache nodes [30]. We do not address an adversary that tampers, modifies, or deletes any data, content, or requests. In addition, it is out of the scope of this work to prevent an attacker that has global monitoring knowledge; for example, OCDN will not prevent an attacker who can see all incoming and outgoing messages from all clients and proxies.

3.2 Security and Privacy Goals

To defend against the adversary described in Section 3.1, we highlight the design goals for OCDN. Each stakeholder—in this case the content publisher, the CDN, and the client—has different risks, and therefore should have different protections. All three stakeholders can be protected by preventing CDNs from learning information, decoupling content distribution from trust, and maintaining the performance benefits of a CDN while reducing the probability of attacks. One strength of OCDN is that it protects the origin server, the CDN itself, and the client, whereas existing systems, such as Tor, only protect the client.

Prevent the CDN from knowing the content it is caching. First and foremost, the CDN should not have access to the information outlined in Section 2. By limiting the information that the CDN knows, OCDN limits the amount of information that an adversary can learn or request. OCDN should hide the content as well as the URL associated with the content. If the CDN does not know what content it is caching, then the CDN will not be able to supply an adversary with the requested data and it will have a strong argument as to why it cannot be held liable for its customers' content.

Prevent the CDN from knowing the identity of users accessing content. CDNs can currently see clients' browsing patterns. OCDN should provide privacy protections by hiding which client is accessing which content at the CDN. In addition, it should hide cross-site browsing patterns, which a CDN is unique in having access to. Some CDNs block legitimate Tor users because they are trying to protect cached content from attacks, such as comment spam, vulnerability scanning, ad click fraud, content scraping, and login scanning [10]; for example, Akamai blocks Tor users [26]. As a positive side effect, OCDN prevents privacy-conscious Tor users from being blocked by CDNs. Finally, some CDNs,

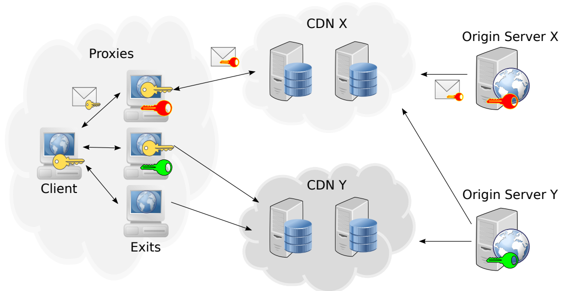

Fig. 2. The relationships between clients, exit proxies, CDNs, and origin servers in OCDN.

due to their ability to view cross site browsing patterns, could de-anonymize Tor users [40], but OCDN would prevent a CDN from compromising the anonymity of clients.

3.3 Performance Considerations

As one of the primary functions of a CDN is to make accessing content faster and more reliable, OCDN should consider performance in design decisions. The performance of OCDN will be worse than that of traditional CDNs because it is performing more operations on content, but OCDN is offering confidentiality, whereas traditional CDNs are not. OCDN should scale linearly in terms of load and storage requirements on proxies; additionally, it should be able to scale with the number of clients using the system, as well as with the growing number of web pages on the internet.

4 OCDN Design

OCDN that provides oblivious content distribution and comprises the following components: clients, exit proxies, CDNs, and origin servers. *Clients* are the Internet users who use the system to access content stored on CDN cache nodes; *exit proxies* are proxies that obfuscate the requests and responses retrieved from the CDNs; and the *origin servers* are the content publishers who are customers of the CDNs. Figure 2 shows how these components interact. This section discusses the design decisions of OCDN, and what functionality each decision provides. We separate design decisions into two parts: 1) hiding content and 2) hiding clients. We also highlight some additional options that the design of OCDN allows.

4.1 Hiding Content

We start by discussing how the system components communicate and authenticate one another. We introduce shared keys between origin servers and exit proxies, how these keys are stored, how the exit proxies authenticate themselves to origin servers, and how these keys are distributed.

Shared Keys. To prevent an adversary from learning information about content and clients, the CDN must not know anything about the content that it is caching. Therefore, the content *and* the associated URL must be obfuscated before the CDN sees them. The content can be obfuscated by encrypting it with a key that is not known to the CDN. Because this must be done prior to any caching, the content publisher must generate a shared key k to encrypt the content with. Encrypting the content alone does not hide all information about content from the CDN; the content identifier, or URL, must also be obfuscated, otherwise the CDN can still reveal information about which clients accessed which URLs (which is indicative of the content). The obfuscated URL should be fixed and relatively small; these requirements reduce storage requirements and prevent the adversary from guessing the URL based on the length of the obfuscated URL. Unfortunately, using a simple hash allows an attacker to guess the content identifier by hashing guesses and comparing with the hashes stored in the CDNs caches. To prevent this attack, the content publisher incorporates the use of the shared key k into the hash of the URL by using a hash-based message authentication code (HMAC). Additionally, if the domain supports HTTPS requests, then the content publisher must also encrypt the associated certificate with the same key k.

The encrypted content and corresponding HMAC are sent to the CDN[1] and stored in its caches. The content publisher then shares the key k with an exit proxy. This key allows the exit proxy to request encrypted content on behalf of clients by computing the HMAC on the URL.

Consistent Hashing. Each exit proxy stores a mapping of URLs to their associated shared key k; for example, if an origin server has shared key k and publishes a web page `www.foo.com`, then an exit proxy will store the mapping of `www.foo.com` to k. The set of exit proxies jointly compute a distributed hash table where

1 Most CDNs allow the publisher to decide on a push or pull model, but OCDN is compatible with either approach.

the key is the URL (`www.foo.com`) and the value is the shared key (k). To assign (key,value) pairs to exit proxies, OCDN uses consistent hashing [24, 31]. Consistent hashing uses a hash function $H(.)$ to generate identifiers for both exit proxies and for URLs; the identifiers are $H(exit\_ID)$ and $H(URL)$. We discuss what $exit\_ID$ is in the next section on Self-Certifying Identifiers. After the hashes are computed, then they are mapped to a point on an identifier circle (modulo 2^m, where m is the length of identifier in bits); each URL ($H(URL)$) on the circle is assigned to the first exit proxy ($H(exit\_ID)$) that is equal to or follows $H(URL)$ on the circle. This hashing method is used in OCDN because it provides: 1) an evenly distributed mapping of URLs to shared keys among the exit proxies, 2) a way to prevent an exit proxy from choosing which URL it wishes to be responsible for, and 3) a relatively small amount of (key,values) to be moved when a new exit proxy is established (or removed).

Self-Certifying Identifiers. Consistent hashing uses identifiers for both the URLs and the exit proxies. While the identifiers for URLs are straightforward ($H(URL)$), the identifiers for exit proxies must provide more information; an exit proxy identifier must be able to prove to an origin server that it is the exit proxy that is responsible for the associated URL. If this validation was not part of OCDN, then any (potentially malicious) exit proxy could request the shared key k from any or all origin servers. To prevent a malicious exit proxy from learning any shared key k, the proxy must be identified by a self-certifying identifer. This technique was first introduced in a self-certifying file system [33]; it allows for other entities (such as origin servers) to certify the exit proxy solely based on its identifier. The format of this identifier ($exit\_ID$) is `IP:hostID`, where `IP` is the exit proxy's IP address and `hostID` is a hash of the exit proxy's public key. A malicious exit proxy cannot *choose* where on the consistent hashing ring it sits because it cannot frequently change and re-hash its own IP address (whereas it could re-generate a new public key).[2] When an exit proxy is requesting the shared key k from an origin server, it sends its identifier and its public key to the origin server. The origin server can then hash the exit proxy's public key and verify it against the `hostID`; this action serves as a proof of the exit proxy's position in the consistent hashing circle, and thus prevents a proxy from lying about where it lies on the ring (and subsequently lying about which URL's shared key it is responsible for). Note that this $exit\_ID$ is used on the consistent hashing circle as $H(IP) : H(hostID)$; the $exit\_ID$ must be the same length as $H(URL)$, so the $exit\_ID$ consists of the first half of the bits of $H(IP)$ concatenated with the first half of the bits of $H(hostID)$.

DNS for Key Distribution. We have discussed how shared keys are generated, used, and stored, and here we describe how they are shared. As previously stated, the origin servers generate shared keys and must share them with the (correct) exit proxies. OCDN uses DNS to do so. To retrieve a shared key k, an exit proxy sends a DNS query to the origin server's authoritative DNS, and it includes its identifier, $exit\_ID$, and its public key in the `Additional Info` section of the query. The authoritative DNS for the origin server validates the exit proxy by hashing the public key and comparing it to the second part of $exit\_ID$, and verifying that the exit proxy is responsible for its URL based on the consistent hashing circle. If the verification is successful, then the authoritative DNS sends the shared key k encrypted under the exit proxy's public key, $\{k\}_{PK_{exit}}$ in the SRV record of the DNS response. The exit proxy extracts k by decrypting with its private key, and stores it in its hash table.

4.2 Hiding Clients

We make additional design choices that concern the requests that clients initiate and the responses they receive. We introduce session keys, how requests are routed from clients to exit proxies, and how responses are routed from exit proxies back to the original client.

Request Routing with Potentially Spoofed Source Routes. As previously described, exit proxies query the CDN on behalf of clients, but the exit proxy should not be able to learn which client sent which request. This obfuscation is accomplished by routing requests through a series of other clients. Each client runs a proxy and is also a peer in this system; this peer-

2 While a malicious exit proxy cannot specifically choose its location on the hashing ring, it could recompute a public key until it finds a certain location on the hashing ring. This is limited by the fact that the exit proxy's IP address is part of its identifier, and we assume that the adversary running the exit proxy cannot change IP addresses to a value of his choice or in a frequent manner. A potential attack that an adversary can execute on a DHT using a consistent hashing scheme is a Sybil attack, where the adversary runs *many* exit proxies to hopefully place himself in his desired location on the hashing ring. We describe countermeasures to a Sybil attack in Section 7.

to-peer system of clients borrows the protocols used for clients joining, leaving, and learning about other clients from the vast literature on peer-to-peer systems [20, 27, 28, 36, 39, 44]. A client routes a request through her peers by using source routing; when the client generates a request, it also generates a source route, which includes the addresses of a set of her peers. The last hop in the source route is the exit proxy that is responsible for the shared key k associated with the URL in her request. The client determines the correct exit proxy by looking up a locally-held URL-proxy map (which is retrieved from a central system that keeps the mapping of URLs to exit proxies). It appends this source route to its request and forwards it to the next peer in the route. When a peer receives a request, she simply forwards it on to the next peer; this continues until the last hop in the source route, which is an exit proxy.

To obscure the client that initiated the request, OCDN allows each client to spoof source routes; specifically, a client can prepend other peers in the route before it initiates a request. For example, a client with identity C could generate a route to exit proxy E that looks like $C \rightarrow G \rightarrow F \rightarrow E$ and can further obfuscate the source of the route by prepending additional clients to the beginning of the route as follows: $D \rightarrow A \rightarrow C \rightarrow G \rightarrow F \rightarrow E$. Neither G, F, nor E know who the original requestor was; from E's point of view, the original requestor could have been D, A, C, G, or F. Recall from Section 3.1 that our threat model does not include a global passive adversary, and therefore do not prevent an adversary with those capabilities from learning which client issued the request; OCDN does prevent an adversary that is running a client and/or exit proxy from identifying which client issued the request. Using a sequence of peers, or even just knowing that a client *can* use a series of peers, hides the identity of the client from other clients, exit proxies, and the CDN.

Spoofed source routes offer a tradeoff between privacy and performance. Clients interested in prioritizing performance can choose to send requests directly to the exit proxy. In such a case, the exit proxy will know the client's identity, but the CDN will not. For more privacy, the client can forward a request through a set of clients between it and the exit proxy. Lastly, the client could *only* prepend other clients' identifiers but simply forward the request *directly* to the exit proxy; this action provides the same performance benefit as the first mode, but still offers some additional privacy benefits. Although the last option would appear to strike the optimal balance between privacy and performance, it cannot be the only option because the exit proxy would always know that the true client is the previous hop in the source route.

Session Keys for Request and Response Confidentiality. In addition to shared keys between origin servers and exit proxies, OCDN uses session keys shared between clients and exit proxies. Session keys provide confidentiality of the requested URL and the response. When the client generates a request, it generates a session key *skey*. and encrypts the URL in her request with this key, which provides $\{\text{URL}\}_{skey}$. The client must also share this session key with the exit proxy, so that the exit proxy can learn the plaintext URL and subsequently compute the HMAC to query the CDN. The client encrypts the session key with the exit proxy's public key, result in $\{\text{skey}\}_{PK_{exit}}$, and appends this value as an additional header on the request. Because her request could be forwarded through a set of client peers, this hides the URL of the request from other clients.

When an exit proxy receives a request from a client, it first extracts the session key *skey* by decrypting it with his private key, and then he decrypts the URL with the session key. This operation yields the original plaintext URL. Using the shared key k from the origin server, it can then compute $\text{HMAC}_k(\text{URL})$ and forward the request to the CDN. Upon receiving a response from the CDN, the exit proxy then decrypts the content with the shared key k, and encrypts the content with the session key *skey* before sending it to the client. When it receives the encrypted response, the client can then decrypt it using *skey*.

Multicast Responses. Using session keys allows for a performance optimization in sending responses back to clients. Instead of sending the encrypted response from the exit proxy back to the client via the set of peers used in the source route, the exit proxy can send it in a multicast manner to all clients that were on the source route. The only client that knows *skey* is the true client that originated the request, therefore none of the other clients can interpret the response, and it reduces the latency for sending the response to the client.

4.3 Design Alternatives

While our explanation of the design of OCDN describes a series of proxies that are run by us, but are not trusted; here we describe alternative designs regarding the system of proxies. The system will work with both a closed, trusted system of proxies, as well as with an open (untrusted) system of proxies.

Closed System of Proxies. While the proxies in OCDN are not trusted, OCDN could use a system of closed and trusted proxies. There are potential groups or organizations that would support this type of system, and would be willing (and trusted) to run the exit proxies. If the exit proxies were trusted, then parts of the design of OCDN could be simplified; for example, if proxies were trusted, then OCDN would not need to hide the identity of clients from the exit proxies, and could remove spoofed source routes from the design. The primary drawback of this approach is finding an organization that everyone could trust to run the exit proxies.

Open System of Proxies. In OCDN, exit proxies are not trusted with client identities and information, which removes the need to find a universally trustworthy organization. In an alternative approach, OCDN could use a completely open system of proxies that are untrusted, which would allow anyone (clients, companies, etc.) to run an exit proxy. The addition of an exit proxy follows the protocol in consistent hashing for when a new node joins; some keys would be transferred to the new exit proxy, and clients' mapping of exit proxies will be updated. This allows for the load to be split among more proxies and increases the geographic diversity of the exit proxies.

4.4 Design Enhancements

We discuss possible enhancements to OCDN's basic design.

Encoding URLs. As described earlier, each URL is obfuscated by using a HMAC and then stored on the CDN. An adversary could potentially correlate a URL's popularity with its access patterns. To prevent this, OCDN allows origin servers to generate multiple different encodings of its URLs, such that $\text{HMAC}_k(\text{enc}_1(\text{URL})) \neq \text{HMAC}_k(\text{enc}_2(\text{URL}))$. Each origin server could produce n different encodings of popular URLs, such that the popularity distribution seen by an adversary is a uniform distribution of URL requests across all URLs.

DHT Replicas. Each exit proxy's hash table can be replicated by another (or many other) exit proxies, spreading the burden across multiple proxies. This behavior would provide less load per exit proxy, as well as redundancy in case of failures. Additionally, the CDN can cache the content associated with a given URL at more than one cache node; if only one exit proxy is responsible for a given URL's content, then it would likely

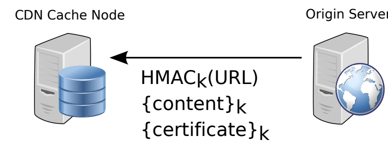

Fig. 3. Publishing content in OCDN. k is shared between the origin server and the corresponding exit proxy; the CDN has no knowledge of k.

only be cached at cache node closest to the exit proxy. Having multiple exit proxies responsible for a URL's content helps decrease the load on the proxies while maintaining some of the performance benefits of a CDN.

Hybrid OCDN/CDN Designs. Different origin servers have different needs, and each origin server might have different needs for different content. The design of OCDN allows origin servers to publish some of their content on OCDN and some on other CDNs. This is useful in a case where some content is more sensitive, while other content needs better performance.

Pre-Fetch DNS Responses. One way to increase the performance of OCDN is to pre-fetch DNS responses at the exit proxies. This would allow the exit proxy to serve each client request faster because it would not have to send as many DNS requests. Pre-fetching DNS responses would not take up a large amount of space, but it also would not be a complete set of all DNS responses. Additionally, if the CDN moves content between cache nodes, then DNS response must also change; therefore, the pre-fetched DNS responses should have a lifetime that is shorter than the lifetime of the content on a cache node.

5 OCDN Protocol

Based on the design decisions discussed in the previous section, we specify the steps taken to publish and retrieve content in OCDN. The OCDN protocol works on both plaintext and encrypted requests (HTTP and HTTPS requests).

5.1 Publishing Content

To publish content such that the CDN never sees it, the publisher must first obfuscate her content, as described in Section 4. Figure 3 shows the steps taken to publish content. The most important step in content publishing is obfuscating the data. We assume that the origin server already has a public and private key pair, as well as a certificate. To obfuscate the data the origin server will need to generate a shared key k.

Once the key is established, the origin server must first pad the content to the same size for some range of original content sizes (i.e., if content is between length x and y, then pad it to length z). The range of content sizes should be small, introducing negligible padding overhead, but reduces the probability of identifying the content based on the content length. This content padding is done to hide the original content's length, as it may be identifiable simply by its length. After content is padded, then the content is divided into fixed size blocks and padded to some standard length. Then each block is encrypted using the shared key k, resulting in a set of encrypted blocks. Because the CDN does not have access to the shared key, it cannot see what content it is caching.

Once the content is obfuscated, the origin server must also obfuscate the content's identifier. To do so, it computes the HMAC of the URL using the shared key k. Once the identifier and the content are obfuscated with k, they can be pushed to the CDN, or optionally to multiple CDNs. Recently, services have cropped up to allow and help facilitate the use of multiple CDNs for the same content; an origin server could use multiple CDNs' services. This mechanism could be used in OCDN to increase reliability, performance, and availability; an origin server can use a service, such as Cedexis [8], to load balance between CDNs. We discuss the use of multiple CDNs more in Section 5.4 on OCDN in partial deployment.

Because the exit proxies use consistent hashing to divide keys among proxies while balancing load, the origin server determines which exit proxy is correct (based on the consistent hashing ring). The origin server then encrypts the shared key k with the correct exit proxy's public key PK_{exit}. Figure 4 shows the steps for retrieveing a shared key. First, the exit proxy sends a DNS request to the origin server's authoritative DNS server, including its self-certifying identifier and its public key (these are both included in the `Additional Info` section of the DNS message). The origin server hashes the exit proxy's public key and verifies it against its self-certifying identifier; this acts as a proof of the exit proxy's position in the consistent hashing ring. If the origin is able to certify the exit proxy, then it will send the DNS response with $\{k\}_{PK_{proxy}}$ in the SRV record. The exit proxy will receive the encrypted shared key, which it can decrypt with it's private key.

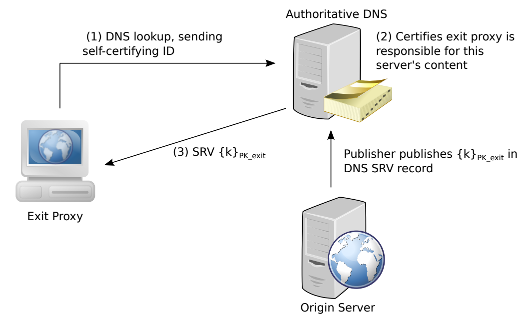

Fig. 4. How an origin server certifies an exit proxy and distributes its shared key to an exit proxy. In step (1), the exit proxy sends his self-certifying ID in the *Additional* section of the DNS message.

Updating Content. For an origin server to update content, it must follow similar steps as described in publishing content. Once it has updated the content on the origin server, it must obfuscate it using the same steps: 1) pad the original content length, 2) divide the content into fixed size blocks, and 3) encrypt the content blocks with the shared key k. Because it is updating the content (as opposed to creating new content), the obfuscated identifier will remain the same. The origin server signs the updated obfuscated content with its private key, such that the CDN can verify it was true origin server that sent the update.

Updating Keys. An origin server must be able to update keys in case of compromise. To minimize the amount of time a key is compromised for, the origin server specifies an expiration date and time for the key when it is originally generated. The origin server periodically checks if the key is valid or not based on the expiration timestamp. If the key is still valid, the origin server continues to use it. Otherwise, the origin server generates a new key k_{new}, computes $\text{HMAC}_{k_{new}}(\text{URL})$, and encrypts the content (and possibly certificate) with k_{new}. The content publisher then follows the same steps as in Updating Content to push the content to the CDN, and it publishes k_{new} encrypted with the exit proxy's public key in its DNS SRV record.

The corresponding exit proxy must also be able to fetch this new key k_{new} and replace the expired key with it. When the exit proxy sees an incoming request for a URL that uses key k, it first checks k's timestamp. If valid, then it continues as normal. Otherwise, it sends a DNS request to the publisher's authoritative DNS, and extracts $\{k_{new}\}_{PK_{proxy}}$ from the DNS response. The exit proxy then decrypts it to obtain k_{new}, updates its version of the key, and proceeds as normal.

5.2 Retrieving Content

The steps for a client to retrieve a web page that has been cached by OCDN are shown in Figure 5, where the client can forward her request directly to an exit proxy or the client can forward her request through two peers. We assume the client has already joined the system, which is described in more detail in Section 5.3; at this stage, the client has knowledge of a subset of its peers (other OCDN clients) and a mapping of exit proxies and which URLs they hold keys for. The client generates a request for a specific URL, and looks up which exit proxy holds the key for that URL in its local mapping. Next, the client selects a source route; this source route allows the client to specify which mode of OCDN they would like to use: 1) no additional source route, which has better performance, or 2) a source route, which has better privacy. If the client decides to use the privacy-preserving mode, then it generates a source route, which includes some of its peers, and could potentially include a false originator (as described in Section 4). Before sending the request, the client generates a session key $k_{session}$ and encrypts it with the exit proxy's public key. The client appends both the source route and $\{k_{session}\}_{PK_{proxy}}$ to the request and encrypts the URL with $k_{session}$ such that no other clients on the path can learn what the requested URL is. The client then sends it onto the next proxy in the source route, which could be either another client proxy or the exit proxy. The request is forwarded through all subsequent hops in the source route until it reaches the exit proxy. The exit proxy decrypts $\{k_{session}\}_{PK_{proxy}}$ with its private key and stores the source route locally; it then decrypts the URL with $k_{session}$.

The exit proxy then resolves the domain using its local resolver, which will redirect it to the CDN's DNS resolver. In order for the exit proxy to generate the obfuscated identifier to query the cache node for the correct content, it must have the shared key k that the origin server generated and obfuscated the content and identifier with. The steps an exit proxy takes to retrieve the shared key were outlined in Section 5.1 and are shown in Figure 4.

Now that the proxy has obtained the shared key k from the origin server, it can generate the obfuscated content identifier based on the request the client sent. It computes the HMAC of the URL with the shared key. The proxy then sends the (obfuscated) request to the edge server, where the CDN locates the content associated with the identifier. The CDN returns the associated obfuscated content, which we recall is the fixed-

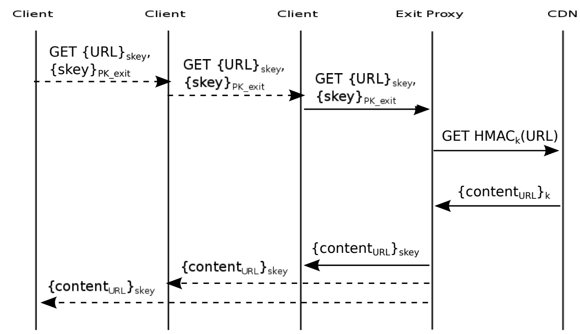

Fig. 5. Steps for retrieving content in OCDN. The dotted lines represent optional steps for when a client is prioritizing privacy and proxies a request through two other clients before reaching the exit proxy. In this case, the request is sent sequentially through peers, and the response is sent in a multicast manner back to the clients.

size blocks encrypted with the same shared key that the identifier was obfuscated with. The proxy can decrypt the content blocks with the shared key from the origin server, assemble the blocks, and strip any added padding, to reconstruct the original content.

Finally, the exit proxy must send the response back to the correct client without knowing who the client is. First, the exit proxy fetches the session key $k_{session}$ that it stored for the corresponding incoming request, and it uses this key to encrypt the response. Then, it looks up the source route it stored for the corresponding request and uses a multicast technique to send to the encrypted response to all clients on the source route. At this point, the exit proxy can delete the source route and session key entries for this request/response. Only the original (true) client has $k_{session}$, so only the original (true) client can decrypt the response. All other clients will discard the encrypted response because they cannot decrypt it.

5.3 Clients Joining and Leaving

When a client joins OCDN, it will download OCDN client software. This includes information about exit proxy mappings to URLs for which they hold a key, software for modifying requests with session keys and source routes, and software for running a proxy. Clients will learn about other clients in the system via a gossip protocol. We do not detail this as gossip protocols have been studied extensively in the past [18, 25, 41]. Similarly, when a client leaves the system, this information is propogated to its peers using a gossip protocol. OCDN scales with increasing numbers of clients and exit proxies, which is described in more detail in Section 8.2.

5.4 Partial Deployment

OCDN should be partially deployable, in the sense that if only some origin servers participate or only some CDNs participate, then the system should still offer some protections. We outline two different partial deployment possibilities below.

Deployment with Origin Servers' Full Participation. One option for deploying OCDN is to ensure there is some set S of origin servers that participate fully in the system. These publishers obfuscate their content, identifiers, and certificates, and most importantly, only have obfuscated data stored on the CDNs cache nodes. S must be greater than one, otherwise the CDN can infer that a client accessing this obfuscated content is actually accessing content that can be identified. This partial deployment plan protects the privacy of the clients accessing the content created by the set of origin servers S. It does not protect the clients' privacy as completely as full participation of all origin servers in OCDN because the CDN can still view cross site browsing patterns among the origin servers that are not participating. It is important to note though, that because the clients are behind proxies, the CDN cannot individually identify users. The CDN can attribute requests to exit proxies, but not to clients.

Deployment with Origin Servers' Partial Participation. Some origin servers may prioritize performance and availability. Therefore, they should have the option to gradually move towards full participation by pushing both encrypted and plaintext content to the CDN. In this partial deployment plan, we see some set of origin servers fully participating with only encrypted content, some other set of origin servers partially participating with both encrypted and plaintext content, and some last set of publishers that are not participating. Unfortunately, if a publisher has the same content that is both encrypted and plaintext content at a cache node, then an adversary can correlate the access patterns on encrypted and plaintext content for the origin server. In order to prevent this identification of the content, OCDN can use encoded URLs (described in Section 4), which obfuscates the access patterns for a given piece of content; this holds true if an origin server chooses to distribute its content in an encrypted manner using OCDN and in plaintext form on a different CDN. In this case, the origin server can still encode its URLs in multiple ways to prevent correlating access patterns between the encrypted and plaintext content. Therefore, this deployment option allows for differing levels of participation in the system, while still preserving the protections provided by OCDN.

6 Implementation

We have implemented a prototype of OCDN to demonstrate its feasibility and evaluate its performance. Our implementation allows a client to send a request for content through an exit proxy, which will fetch the corresponding encrypted content. Figure 6 shows our prototype; the solid line represents how OCDN communicates between the components, and the dotted line represents how a traditional CDN would communicate in our prototype. Here we will discuss each component—client proxy, exit proxy, and CDN—separately, and how they fit together.

CDN. As the design for OCDN requires encrypted content and identifiers to be stored in the CDN, we cannot request content from real-world CDNs without establishing a business relationship with an existing CDN ourselves. Additionally, we must evaluate the performance of OCDN in comparison to the same content, cache locations, etc., so we set up a data storage server. This server is run on a Virtual Private Server (VPS) located in Chicago, USA. To access content, we set up a web server on this VPS machine. To generate plaintext web content, we used Surge [6], which allows us to generate a set of files that are representative of real-world web server file distributions. In OCDN, the files are encrypted with a shared key k and the obfuscated file name is the HMAC_k(file name). We use AES with 256-bit keys for the shared key and SHA-256 for the hash function. Both the plaintext files and encrypted files are stored on this web server, and for the purposes of evaluating our prototype, act as a CDN in OCDN.

Exit Proxy. The exit proxy is the component that queries the CDN for encrypted content on behalf of a client. We have implemented a web proxy in Go; this proxy runs on a different VPS machine in Chicago, USA. In addition to proxying web requests, the exit proxy also provides cryptographic functionality. When receiving a request, it rewrites the URL in the request to be the HMAC_k(URL), and it parses the headers to retrieve a specific header, `X-OCDN`, which contains the client's session key encrypted under the exit proxy's public key. Our implementation uses 2048-bit RSA for asymmetric

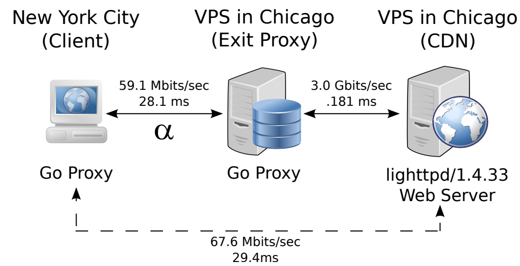

Fig. 6. The implementation of our OCDN prototype. The solid line represents how OCDN communicates between the components; the dotted line represents how a traditional CDN would communicate. α represents the latency between the client and the exit proxy; we simulate additional clients on this path by increasing α.

encryption. After decrypting the session key, it stores it in memory for use on the response. When it receives a response from the CDN, it decrypts the content with the shared key k, and subsequently encrypts it with the session key (both using AES 256-bit encryption). The exit then forwards the response onto the client proxy.

Client Proxy. The client proxy acts on behalf of the client that is requesting content. This proxy uses the same implementation as the exit proxy, but provides different cryptographic functions on the requests and responses. When a client makes a request, the client proxy generates a session key (AES 256-bit) and looks up the correct exit proxy's public key. The client proxy then adds a header to the request, where X-OCDN is the key, the encrypted session key is the value. The client then forwards this on to the exit proxy. When the client receives a response from the exit proxy, it must decrypt the content with the session key it originally generated.

7 Attacks Against OCDN

We analyze and discuss how OCDN defends against different attacks. Table 1 shows what security and privacy features OCDN provides in comparison to other related systems.

Popularity Attacks. An attacker that has requested or otherwise gained access to CDN cache logs can learn information about how often content was requested. Because not all content is requested uniformly, the attacker could potentially correlate the most commonly requested content with very popular webpages. While this does not allow the CDN to learn which clients are accessing the content, it can reveal information about what content is stored on the CDN cache

	Preserves Integrity at CDN	Preserves Confidentiality at CDN	Protects Client Identity
Stickler [30]	✓		
R & C [34]	✓		
Tor [15]			✓
OCDN		✓	✓

Table 1. The security and privacy features offered by related systems. To our knowledge, OCDN is the first to address confidentiality at the CDN.

nodes. OCDN handles this type of attacker by making the distribution of content requests appear uniform. The content publisher (of popular content) generates multiple encodings of their content and URLs, and encrypts each one with the shared key k, such that they have multiple, different-looking copies of their content. All of the content copies are pushed to the CDN and the key is shared with the exit proxy.[3] Now, the popular content does not appear as popular, and it makes it difficult for an attacker to infer the popularity of the content.

Chosen Plaintext Attacks. An attacker could attempt to determine whether a particular URL was being accessed by sending requests through specific OCDN proxies and requesting access to the CDN cache logs, which contain the corresponding obfuscated requests and responses. Blinding the clients' requests with a random nonce that is added by the proxy should prevent against this attack. We also believe that such an attack reflects a stronger attack: from a law enforcement perspective, receiving a subpoena for *existing* logs and data may present a lower legal barrier than compelling a CDN to attack a system.

Traffic Analysis Attacks. If a CDN itself is malicious and is attempting to learn information about the content and/or clients, the CDN may act as a client in the system. In this attack the (malicious) client sends a request for content and the CDN can correlate the request with a content access at the CDN because they have knowledge of both the CDN logs and the requests they are making as a client. OCDN defends against this by using the exit proxies as *natural* mixes; each exit proxy is receiving different requests from different clients and then forwarding the requests on to the CDN. These exit proxies naturally mix the requests enough

3 This also provides load balancing for exit proxies that hold the shared key k for the popular webpage because it distributes the load across multiple exit proxies (where each of these exit proxies are responsible for one of the encodings).

that the CDN cannot conduct traffic analysis and determine which request corresponds to which content on the CDN's cache nodes.[4] There has also been numerous studies that have proposed and evaluated defenses against traffic analysis attacks [38, 43]; OCDN could implement one of these solutions (such as slight timing delays) at the exit proxy.

Spoofed Content Updates. Because the CDN cache nodes do not know either the content that they are hosting or the URLs corresponding to the content, an attacker could masquerade as an origin server and could potentially push bogus content for a URL to a cache node. There are a number of defenses against this possible attack. This simplest solution is for CDN cache nodes to authenticate origin servers and only accept updates from trusted origins; this approach is plausible, since many origin servers already have a corresponding public key certificate through the web PKI hierarchy. An additional defense is to make it difficult for to discover which obfuscated URLs correspond to which content that an attacker wishes to spoof; this is achievable by design. A third defense would be to only accept updates for content from the same origin server that populated the cache with the original content.

Timing Attacks. An attacker who is passively observing traffic could potentially correlate requests and responses based on timing information. To address this type of attack, OCDN could employ techniques used in previous research, such as implementing timing delays, at different proxies (either client or exit proxies).

Sybil Attacks. An adversary who runs *many* exit proxies can learn information about many clients and many content requests as they are responsible for encrypting/decrypting *many* requests and responses. Previous work has analyzed the security of DHTs in this context [19, 42]. OCDN can employ a few different defenses to limit the probability and size of a Sybil attack. To limit the number of exit proxies running on a single machine, OCDN can limit the number of exit proxies with the same IP address (which is a part of the exit proxy's self-certifying identifier) to one; therefore, the attack becomes more expensive as the number of machines the adversary must control increases.[5] This defense can be expanded to entire network prefixes; for example, if a large (malicious) organization owns an entire prefix, they could launch a Sybil attack using various IP addresses within their network. OCDN could limit the number of IP addresses in a given prefix to either one (which may result in a smaller set of exit proxies) or some small number (in which case the size of the Sybil is extremely small and cannot achieve its goal of being in a certain location on the hashing ring).

Flashcrowds. A flashcrowd is a large spike in traffic to a specific web page. An attacker could see that some content on the CDN has just seen a surge in traffic and correlate that with other information (for example, major world events). This leaks information about what content the CDN is caching. Fortunately, the design of OCDN can defend against this type of inference attack. The exit proxy can cache content in the time of a flashcrowd, such that the CDN (and therefore the attacker) does not see the surge in traffic.[6]

8 Performance Evaluation

To evaluate how much overhead is caused by OCDN we measure the performance of OCDN. In addition to understanding the latency and overhead produced by the system, we also discuss the scalability of the design and show how OCDN scales well with an increasing number of clients.

8.1 Throughput and Latency Overhead

For measuring performance characteristics of OCDN, we use the implementation described in Section 6. Figure 6 shows how our measurements reflect OCDN (solid line) and a traditional CDN (dotted line).

Figure 7 shows the time to first byte (TTFB) and the time for both OCDN and without OCDN. We can see the the TTFB using OCDN increases linearly with

file size, whereas without OCDN TTFB remains fairly constant. Interestingly, we can see that there are some fixed-time operations that OCDN performs, which

is visible by looking at the smaller file sizes. Completion time increases linearly with file size, but for both OCDN and without OCDN; while both follow the same

4 For clarification, the exit proxies are not a type of mixnet [9], they are simply receiving requests and then forwarding them, and due to the variety of requests, they provide natural mixing characteristics.

5 Note also that this countermeasure prevents two or more exit proxies from being behind the same NAT.

6 This raises billing issues because the CDN can't charge as much if edge servers don't see as many requests for the origin; fortunately, RFC 2227 describes a solution for this [35].

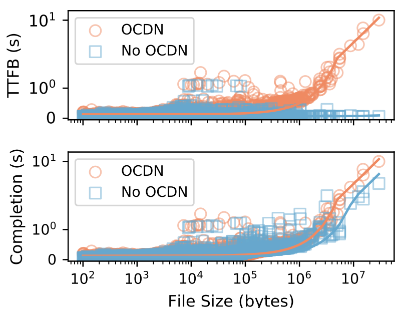

Fig. 7. Time to first byte and time to complete a request with and without OCDN.

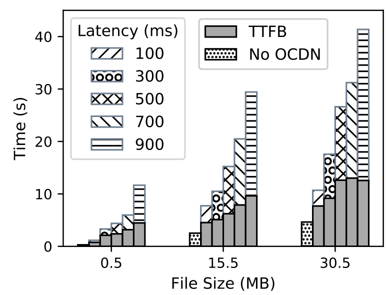

Fig. 8. Time to first byte and time to complete a request with varying the file size and latency.

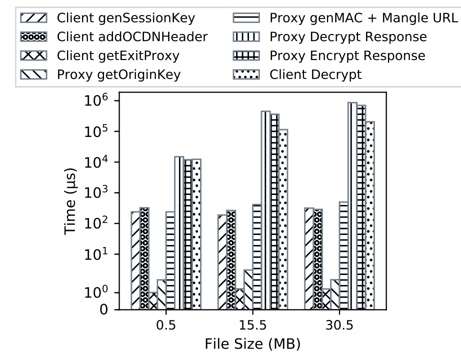

Fig. 9. Overhead of different operations performed by OCDN.

pattern, the time to complete requests is, as expected, longer using OCDN as it performs many cryptographic operations and proxies traffic between the client and the CDN.

As described in Section 6, our prototype included only a single client, but our design allows for a client to proxy her request through additional clients. To simulate this, we add latency between the client and the exit proxy, and measure both the TTFB and time to complete a request when there are different values of latency, which represent different numbers of clients on the path between the original client and the exit proxy. Figure 8 shows the results for three different file sizes. The bottom portion of each bar in the graph shows the TTFB, and the top portion shows the additional time needed to complete the request. As expected, the TTFB increases more slowly as file size and latency increase; completion time increases more quickly than TTFB as the file size and latency increase.

Finally, we measure the performance overhead of the individual operations used in OCDN; figure 9 shows the overhead of different components of the system for three different file sizes. We can see that some of the fixed cost/time operations include the client locally looking up the correct exit proxy to use for a given URL and the exit proxy generating the $\mathrm{HMAC}_k(\mathrm{URL})$. The operations that have the most overhead and continue to grow with the size of the file are the exit proxy decrypting the response with the shared key k, the exit proxy encrypting the response with the session key $k_{session}$, and the client decrypting the response with the session key $k_{session}$.

8.2 Scaling with Users and Content

For evaluating performance, we are also concerned with how well OCDN will scale with more users and more URLs. In particular, we need to reason about how much load is put on the exit proxies as the system increases; clients do not bare much load in the system as they simply proxy requests and the CDN is designed to handle high numbers of requests, therefore, we limit our scalability analysis to the exit proxies.

As previously mentioned, we balance load among the proxies by using consistent hashing to assign URLs to exit proxies. OCDN can additionally distribute load by replicating exit proxies, meaning that two exit proxies can have the same distributed hash table of shared keys. This way, both exit proxies can accept requests from clients for the URLs they are responsible for. Also worth noting is that the exit proxy is only recieving requests for the content corresponding to the shared keys it contains. Therefore, as the number of clients grow, the exit proxy is still only responsible for its set of shared keys and subsequent URLs. And as the number of URLs increase, the additional load per proxy is still low because of the load balancing properties of consistent hashing. We also discuss in Section 9 how clients can set up exit proxies; this will further decrease the load per exit proxy because each exit proxy will be responsible for a smaller number of shared keys/URLs.

9 Discussion

In this section, we discuss the various technical, political, and legal limitations of OCDN, as well as possible avenues for future work.

OCDN limitations. CDNs become slightly limited in terms of the possible performance optimizations when following OCDN's design. For example, many CDNs perform HTTPS re-writes on cached content, but this can only be done if the CDN has access to plaintext content. Similarly, the CDN needs the decrypted content to perform minimizations on HTML, CSS, and Javascript files. While this increases performance in traditional CDNs, it does not provide the greatest increase in performance; content caching around the world is the greatest benefit to performance, which OCDN preserves.

CDNs operated by content hosts. The design of OCDN assumes that the entities operating the proxies and delivering content are distinct from the original content provider. In many cases, however—particularly for large content providers such as Netflix, Facebook, and Google—the content provider operates their own CDN; in these cases, OCDN will not be able to obfuscate the content from the CDN operator, since the content host and the CDN are the same party. Similarly, because the CDN operator is the same entity as the original server, it also knows the keys that are shared with the clients. As a result, the CDN could also discover the keys and identify both the content, as well as which clients are requesting which content.

10 Related Work

To our knowledge, there has been no prior work on preventing surveillance at CDNs, but there has been relevant research on securing CDNs and finding security vulnerabilities in CDNs.

Securing CDNs. Most prior work on securing CDNs has focused on providing content integrity at the CDN as opposed to content confidentiality (and unlinkability). In 2005, Lesniewski-Laas and Kaashoek use SSL-splitting — a technique where the proxy simulates an SSL connection with the client by using authentication records from the server with data records from the cache (in the proxy) — to maintain the integrity of content being served by a proxy [29]. While this work does not explicitly apply SSL-splitting to CDNs, it is a technique that could be used for content distribution. Michalakis et al., present a system for ensuring content integrity for untrusted peer-to-peer CDNs [34]. This system, Repeat and Compare, uses attestation records and a number of peers act as verifiers. More recently, Levy et al., introduced Stickler, which is a system that allows publishers to guarantee the end-to-end authenticity of their content [30]. Stickler includes content publishers signing their content, and users verifying the signature without having to modify the browser. Unfortunately, systems like Stickler do not protect against an adversary that wishes to learn information about content, clients, or publishers; OCDN is complementary to Stickler.

Security Issues in CDNs. More prevalent in the literature than defense are attacks on CDNs. Liang et al., studied 20 CDN providers and found that there are many problems with HTTPS practice in CDNs [32]. Some of these problems include: invalid certificates, private key sharing, neglected revocation of stale certificates, and insecure back-end communications; the authors point out that some of these problems are fundamental issues due to the man-in-the-middle characteristic of CDNs. Similarly, Zolfaghari and Houmansadr found problems with HTTPS usage by CDNBrowsing, a system that relies on CDNs for censorship circumvention [45]. They found that HTTPS leaks the identity of the content being accessed, which defeats the purpose of a censorship circumvention tool. Jung et al., show that some CDNs might not actually provide much defense against flash events [23]. Recently, researchers have studied the privacy implications of peer-assisted CDNs; peer-assisted CDNs allow clients to cache and distribute content on behalf of a website. It is starting to be supported by CDNs, such as Akamai, but the design of the paradigm makes clients susceptible to privacy attacks; one client can infur the cross site browsing patterns of another client [22].

11 Conclusion

As more content is distributed via CDNs, CDNs are increasingly becoming the targets of data requests and liability cases. In response, we design OCDN, which provides oblivious content distribution. OCDN obfuscates data such that the CDN can operate without having knowledge of what content they are caching. This system not only provides protections to CDNs, but also preserves client privacy by ensuring that the CDN never learns the identity of clients that make requests for content. OCDN is robust against a strong adversary who has access to request logs at the CDN and can also join the system as a client, publisher, or CDN. Our evaluation shows that OCDN imposes some performance overhead, but that this overhead is acceptable, particularly for the sizes of files that make up common web workloads.

References

[1] Court Says RIAA Can Just Tell Cloudflare Any Site Is a Grooveshark Clone... And Cloudflare Has 48 Hours To Dump Them. https://www.techdirt.com/articles/20151214/17392833079/court-says-riaa-can-just-tell-cloudflare-any-site-is-grooveshark-clone-cloudflare-has-48-hours-to-dump-them.shtml.

[2] 47 U.S. Code § 230 - Protection for private blocking and screening of offensive material. https://www.law.cornell.edu/uscode/text/47/230.

[3] Akamai Empowers Operators to Deploy Their Own Content Distribution Network. https://www.akamai.com/us/en/resources/content-distribution-network.jsp.

[4] Akamai: Facts & Figures. https://www.akamai.com/us/en/about/facts-figures.jsp.

[5] Balancing Child Protection and Digital Rights. https://medium.com/@jmalcolm/balancing-child-protection-and-digital-rights-1b9c4ab0b93f.

[6] Barford, P., and Crovella, M. Generating representative web workloads for network and server performance evaluation. In *ACM SIGMETRICS Performance Evaluation Review* (1998), vol. 26, ACM, pp. 151–160.

[7] Cai, X., Zhang, X. C., Joshi, B., and Johnson, R. Touching from a distance: Website fingerprinting attacks and defenses. In *ACM Conference on Computer and Communications Security* (2012), ACM, pp. 605–616.

[8] Cedexis. https://www.cedexis.com/.

[9] Chaum, D. L. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM 24*, 2 (1981), 84–90.

[10] Cloudflare: 94 Percent of the Tor Traffic We See is "Per Se Malicious". https://arstechnica.com/tech-policy/2016/03/new-data-suggests-94-percent-of-tor-traffic-is-malicious/.

[11] Content Delivery Networks Aren't Notorious Markets. https://medium.com/niskanen-center/content-delivery-networks-arent-notorious-markets-2a448a549cf7.

[12] Content Delivery Networks (CDNs). https://www.eff.org/free-speech-weak-link/cdn.

[13] CREDO and Cloudflare Argue Against National Security Letter Gag Orders. https://techcrunch.com/2017/03/23/credo-and-cloudflare-argue-against-national-security-letter-gag-orders/.

[14] Delete Hate Speech or Pay Up, Germany Tells Social Media Companies. https://www.nytimes.com/2017/06/30/business/germany-facebook-google-twitter.html.

[15] Dingledine, R., Mathewson, N., and Syverson, P. Tor: The second-generation onion router. Tech. rep., Naval Research Lab Washington DC, 2004.

[16] District court rules against Cloudflare in anti-piracy suit. https://thestack.com/security/2017/03/29/district-court-rules-against-cloudflare-in-anti-piracy-suit/.

[17] EU Proposal for a Directive of The European Parliament and of the Council on Copyright in the Digital Single Market. https://ec.europa.eu/transparency/regdoc/rep/1/2016/EN/1-2016-593-EN-F1-1.PDF.

[18] Eugster, P. T., Guerraoui, R., Kermarrec, A.-M., and Massoulié, L. Epidemic information dissemination in distributed systems. *Computer 37*, 5 (2004), 60–67.

[19] Geambasu, R., Falkner, J., Gardner, P., Kohno, T., Krishnamurthy, A., and Levy, H. M. Experiences building security applications on DHTs. *Technical Report, UW-CSE-09–09–01* (2009).

[20] Gupta, I., Birman, K., Linga, P., Demers, A., and Van Renesse, R. Kelips: Building an efficient and stable p2p dht through increased memory and background overhead. In *International Workshop on Peer-to-Peer Systems* (2003), Springer, pp. 160–169.

[21] How the NSA & FBI made Facebook the perfect mass surveillance tool. https://venturebeat.com/2014/05/15/how-the-nsa-fbi-made-facebook-the-perfect-mass-surveillance-tool/.

[22] Jia, Y., Bai, G., Saxena, P., and Liang, Z. Anonymity in peer-assisted CDNs: Inference attacks and mitigation. *Proceedings on Privacy Enhancing Technologies 2016*, 4 (2016), 294–314.

[23] Jung, J., Krishnamurthy, B., and Rabinovich, M. Flash crowds and denial of service attacks: Characterization and implications for CDNs and web sites. In *International Conference on World Wide Web* (2002), ACM, pp. 293–304.

[24] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *ACM Symposium on Theory of Computing* (1997), ACM, pp. 654–663.

[25] Kermarrec, A.-M., and Van Steen, M. Gossiping in distributed systems. *ACM SIGOPS Operating Systems Review 41*, 5 (2007), 2–7.

[26] Khattak, S., Fifield, D., Afroz, S., Javed, M., Sundaresan, S., Paxson, V., Murdoch, S. J., and McCoy, D. Do You See What I See? Differential Treatment of Anonymous Users. In *Network and Distributed System Security Symposium* (2016).

[27] Leong, B., and Li, J. Achieving one-hop DHT lookup and strong stabilization by passing tokens. In *IEEE International Conference on Networks* (2004), vol. 1, IEEE, pp. 344–350.

[28] Lesniewski-Laas, C. A Sybil-proof one-hop DHT. In *Workshop on Social Network Systems* (2008), ACM, pp. 19–24.

[29] Lesniewski-Laas, C., and Kaashoek, M. F. SSL splitting: Securely serving data from untrusted caches. *Computer Networks 48*, 5 (2005), 763–779.

[30] Levy, A., Corrigan-Gibbs, H., and Boneh, D. Stickler: Defending against malicious CDNs in an unmodified browser. *arXiv preprint arXiv:1506.04110* (2015).

[31] Lewin, D. M. *Consistent hashing and random trees: Algorithms for caching in distributed networks*. PhD thesis, Massachusetts Institute of Technology, 1998.

[32] Liang, J., Jiang, J., Duan, H., Li, K., Wan, T., and Wu, J. When HTTPS meets CDN: A case of authentication in delegated service. In *IEEE Symposium on Security and Privacy (S&P)* (2014), IEEE, pp. 67–82.

[33] Mazières, D. D. F. *Self-certifying file system*. PhD thesis, Massachusetts Institute of Technology, 2000.

[34] Michalakis, N., Soulé, R., and Grimm, R. Ensuring content integrity for untrusted peer-to-peer content distribution networks. In *USENIX Conference on Networked Systems Design & Implementation* (2007), USENIX Association, pp. 11–11.

[35] Mogul, J., and Leach, P. Simple Hit-Metering and Usage-Limiting for HTTP. https://www.ietf.org/rfc/rfc2227.txt, 1997.

[36] Monnerat, L. R., and Amorim, C. L. D1HT: a distributed one hop hash table. In *International Parallel and Distributed Processing Symposium* (2006), IEEE, pp. 10–pp.

[37] Panchenko, A., Lanze, F., Zinnen, A., Henze, M., Pennekamp, J., Wehrle, K., and Engel, T. Website fingerprinting at internet scale. In *Network & Distributed System Security Symposium (NDSS). IEEE Computer Society* (2016).

[38] Rackoff, C., and Simon, D. R. Cryptographic defense against traffic analysis. In *ACM symposium on Theory of Computing* (1993), ACM, pp. 672–681.

[39] Risson, J., Harwood, A., and Moors, T. Stable high-capacity one-hop distributed hash tables. In *IEEE Symposium on Computers and Communications* (2006), IEEE, pp. 687–694.

[40] The Trouble with Tor. https://blog.cloudflare.com/the-trouble-with-tor/.

[41] Vogels, W., Van Renesse, R., and Birman, K. The power of epidemics: robust communication for large-scale distributed systems. *ACM SIGCOMM Computer Communication Review 33*, 1 (2003), 131–135.

[42] Wolchok, S., Hofmann, O. S., Heninger, N., Felten, E. W., Halderman, J. A., Rossbach, C. J., Waters, B., and Witchel, E. Defeating Vanish with Low-Cost Sybil Attacks Against Large DHTs. In *NDSS* (2010).

[43] Wright, C. V., Coull, S. E., and Monrose, F. Traffic morphing: An efficient defense against statistical traffic analysis. In *NDSS* (2009), vol. 9.

[44] Zhu, Y., and Hu, Y. Efficient, proximity-aware load balancing for DHT-based P2P systems. *IEEE Transactions on Parallel and Distributed Systems 16*, 4 (2005), 349–361.

[45] Zolfaghari, H., and Houmansadr, A. Practical censorship evasion leveraging content delivery networks. In *ACM SIGSAC Conference on Computer and Communications Security* (2016), ACM, pp. 1715–1726.