OCDN: Oblivious Content Distribution Networks

***Abstract*—Recent episodes of governments' attempts to access user data have drawn increased concern to where data belonging to or about a user (including a user's access patterns) is located. The location of user data can affect which governing authorities can access it. Because Content Distribution Networks (CDNs) can cache content in many different countries, often without regard to the owner of the data, a user's data may reside in a country that differs from their own. Many battles over stored data privacy have played out in the courts, with legal opinions rendering essentially ambiguous outcomes thus far. This paper offers a technical contribution to this tussle. We design and implement OCDN, a system that provides *oblivious content distribution*, a property that allows CDNs to provide the performance benefits of content distribution and caching, while preventing the CDN from knowing what content is stored on the CDN or who is accessing it. These properties of OCDN provide protections for both the client, the content owner, and the CDN operator. We design a protocol for publishing and retrieving content using OCDN and, through a prototype implementation and evaluation, show that the performance overhead is negligible.**

I. Introduction

Content distribution networks (CDNs) makes the threat to citizens' privacy more widespread, because data may be replicated in geographically diverse regions, and users may not have purview over where their data is stored, or where their traffic goes. Another recent development is that the CDNs themselves may become liable for the content that they are hosting. 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 [**?**]. Both of these tussles are currently being addressed in the courts, yet the legal outcomes remain ambiguous and uncertain, sometimes with courts issuing opposing rulings in different cases. In this paper, we explore alternate designs that could help resolve this tussle.

In this paper, we design and implement a content distribution network that allows clients to retrieve web objects, while preventing the CDN cache nodes from learning either the content that is stored on the cache nodes or the content that clients are requesting. 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 are requesting. OCDN allows clients to request individual objects with identifiers that are encrypted with a key that is shared by a proxy and the origin server that is pushing content to cache nodes, but is not known to any of the CDN cache nodes. To do so, the origin server publishes content obfuscated with a shared key, which is subsequently shared with a corresponding proxy. To retrieve content, a client forwards her request through a set of peers (other clients) in a way that prevents other clients and the CDN from learning her identity or what content she requested. After being routed through other clients' proxies, the request is handled by an exit proxy; the exit proxy transforms the URL that it receives from a client to an obfuscated identifier using the key shared with the origin server. The CDN cache node then returns the object corresponding to the object identifier; that object is also encrypted with a key that is shared between the origin and the proxy. This approach allows a user to retrieve content from a CDN without the cache node ever seeing the URL or the corresponding content. Users can use OCDN with minimal modification to their existing configuration and specify one of two modes that OCDN operates in, one in which performance is a higher priority and another in which security is a higher priority.

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 run. 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 fetched [**?**], [**?**]. Similarly, URLs can often be inferred from relative popularity in a distribution of web requests, even when the requests themselves are encrypted. OCDN addresses these challenges using different techniques, most of which are tunable to different performance and privacy requirements. Our evaluation explores the implications of the tradeoffs between performance and privacy, as well as how OCDN performs relative to a conventional CDN.

We make the following contributions. First, we explore the problems that arise when data is stored across jurisdictional boundaries and highlight the need for a CDN that is oblivious to the content it is hosting and serving. Second, we design and implement OCDN, a CDN that can be oblivious to the content that it is hosting, as well as the content that clients are retrieving. Finally, we evaluate the performance of OCDN for individual object retrieval and cache hit rates, showing that OCDN incurs a negligible performance overhead relative to conventional CDNs.

The rest of the paper is organized as follows. Section **??** describes the typical operation of a CDN, the types of information that CDN operators typically know today, and the types of information we aim to obfuscate. Section **??** describes the threat model and security objectives for OCDN; based on these threats, Section **??** outlines various design decisions. Section **??** describes the protocol for both publishing and retrieving content and we describe our prototype implementation in Section **??**. Section **??** evaluates the performance overhead of the system and Section **??** analyzes the security of

OCDN. Section **??** describes various limitations and possible avenues for future work, Section **??** discusses related work, and Section **??** concludes.

II. Background

Before describing how OCDN works, we outline how a CDN typically operates, what information it naturally has access to, and some of the ongoing legal battles surrounding CDNs.

A. Content Distribution Networks

CDNs provide content caching as a service to content publishers. A content publisher may wish to use a CDN provider for a number of 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.
- 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 [**?**]. 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.

B. Information Visible to CDNs

Because the CDN interacts with both content publishers and clients, as shown in Figure **??**, 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.

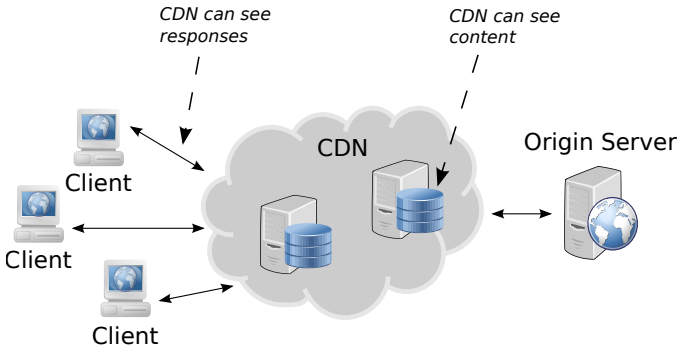

Fig. 1: The relationships between clients, the CDN, and content publishers in CDNs today.

a) Knowing the content: CDNs, by nature, have access to all content that they distribute, as well as the URL. First, the CDN must use the URL, which is not encrypted or hidden, to locate the content. Therefore, it is evident that the CDN already knows what content is stored in its caches. And because CDNs provide analytics to content publishers, they keep track of cache hit rates, and how often content is accessed. But the CDN does not just know about the content identifier, it also has access to the plaintext content. The CDN performs optimizations on the content to increase performance; for example, CDNs minimize CSS, HTML, and JavaScript files, which reduces file sizes by about 20%. They can also inspect content to conduct HTTPS re-writes; we discuss how OCDN handles these types of optimizations later in Section **??**. In addition, requesting content via HTTPS does not hide any information from the CDN; if a client requests a web page over HTTPS, the CDN terminates the TLS connection on behalf of the content publisher. This means that not only does the CDN know the content, the content identifier, but also knows public and private keys, as well as certificates associated with the content it caches.

b) Knowing client information: Clients fetch content directly from the CDN's edge servers, which reveals information about the client's location and what the client is accessing. Unique to CDNs is the fact that they can 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. This gives an enormous amount of knowledge to CDNs; for example, Akamai caches enough content around the world to see up to 30% of global Internet traffic [**?**]. And we have seen the implications of a CDN knowing this much information when Cloudflare went public with the years-long National Security Letters they had received [**?**]. These National Security Letters demand information collected by the CDN and also include a gag order, which prohibits the CDN from publicly announcing the information request.

c) Knowing 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 [**?**].

C. Open Legal Questions

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. In the copyright realm, the European Commision is considering legislation that would remove safe harbor protection against copyright law for online service providers if they host infringing content, regardless of the provenance of that content. The Cloudflare CDN has been required to share data with FBI [**?**]; 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" [**?**].

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 started targeting 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 [**?**], [**?**]. While U.S. law Section 230 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, as mentioned in Section **??**, which involved the RIAA, Cloudlfare, and Grooveshark [**?**]. And again in 2017, a district court ruled that Cloudflare is not protected from anti-piracy injuctions by the Digital Millennium Copyright Act (DMCA); the RIAA obtained a permanent injunction against a site known as MP3Skull, which contained pirated content, and was distributed by Cloudflare. The ruling did not specify that Cloudflare was enjoined with MP3Skull under the DMCA, but rather that Cloudflare was helping MP3Skull in evading the injunction (under Rule 65 of the Federal Rules of Civil Procedure) [**?**].

The role of a CDN as an intermediary has also come into question in the announcement of new legislation, including a new German hate speech law and a bill proposed by the U.S. Senate called Stop Enabling Sex Traffickers Act (SESTA). In October 2017, Germany passed a new law that imposes large fines, upwards of five million euros, on social media companies that do not take down illegal, racist, or slanderous comments and posts within 24 hours [**?**]. The law targets companies like Facebook, Google, and Twitter, but could also apply to smaller companies, which could be serviced by CDNs. In the latter case, it is an open question whether this new law also applies to CDNs. In the United States, a new bill, SESTA, would make Internet platforms liable for their user's illegal comments and posts [**?**]. SESTA would CDNs liable for the content that they distribute (despite the CDN not being a party in the content publishing); this could potentially lead to censoring content too much (an intermediary may be more willing to err on the side of caution–censorship). This type of law could set a dangerous precedent for the censoring other types of content that are unpopular, but still legal.

All these cases highlight a key problem that CDNs face by knowing all the content that they distribute: it may burden them with the legal responsibility for the actions of their customers and clients.

III. Threat Model and Security Goals

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

A. Threat Model

Our threat model is a passive, but powerful adversary. We are concerned with an adversary who has many different capabilities. A potential adversary could gain access to the CDNs logs, which can contain client IP addresses and URLs. Additionally, he could join OCDN as either a client or any number of clients; he is also capable of joining the system as an arbitrary number of exit proxies. The adversary could also act as an origin server (a content publisher). The potential attacker not only has the power to perform any of these actions, but can do these simultaneously. For example, the adversary could join as a client, an exit proxy, and request access to the CDN's logs. 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.

This adversary has many strong capabilities making him a powerful adversary. There has been precedent of this type of attacker; for example, governments have demanded access to CDNs' data [**?**]. A possible adversary is a government requesting logs from the CDN, but the government could also be colluding with a CDN. Additionally, the adversary could be a CDN operator, himself.

As mentioned, OCDN addresses a *passive* adversary; active attackers are out of the scope of this work. 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 [**?**]. We do not address an adversary that tampers, modifies, or deletes any data, content, or requests.

B. Security and Privacy Goals for OCDN

To protect against the attackers described in Section **??**, we highlight the design goals for OCDN. Each stakeholder, in this case the content publisher, the CDN, and the client, each have 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. We further discuss our design decisions in Section **??**.

d) Prevent the CDN from knowing the content it is caching: First, and foremost, the CDN should not have access to all the information that was outlined in Section **??**. 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.

e) Prevent the CDN from knowing the identity of users accessing content: CDNs can currently see clients' browsing patterns and which web pages they are visiting. 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; for example, Akamai blocks Tor users [**?**]. OCDN would prevent privacy-concious Tor users from being blocked at CDNs. Lastly, some CDNs, due to their ability to view cross site browsing patterns, could de-anonymize Tor users [**?**], but OCDN would prevent a CDN from compromising the anonymity of clients.

A 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.

C. 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 with increasing file sizes; 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.

IV. Design

We describe the evolution of the design of OCDN starting with an initial strawman approach. We then alter the design to address each of the design goals discussed in the previous section, which brings us to a complete design.

A. Strawman: Hiding Information From CDN

To prevent an inside attacker or overreaching government from learning information, the CDN must not have the knowledge of what content it is caching. Therefore, the content *and* the associated URL must be obfuscated before they enter the CDN.

f) Encrypt Content: 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 has to generate some key k to encrypt the content with. Then this encrypted (and subsequently obfuscated) content is then sent to the CDN[1] and stored in its caches. Additionally, if the domain supports HTTPS requests, then the content publisher must also encrypt the associated certificate with the same key k. This content and certificate must be decrypted after it leaves the CDN; the client could decrypt the certificate and check its validity. If valid, then the client uses k to decrypt the content.

g) Obfuscate URL: Encrypting the content alone does not hide much 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). In obfuscating the URL, the result should be a fixed, and relatively small size; these requirements are to preserve storage space and to prevent the adversary from guessing the URL based on the length of the obfuscated URL. Hashing the URL provides these properties, and hides the actual URL from the CDN. With obfuscated content and URLs, the CDN does not know what content a given client is accessing.

h) Client Anonymization: The CDN may not know the content that a client is accessing, but it still knows information about the clients that are accessing any content at the CDN. It knows where the clients are located and how many times they are accessing any content. To address this, a client can simply use an anonymizing proxy or VPN when accessing content. This hides a certain amount of information about the client, including the client's location and a direct link of client to content request.

This strawman approach obfuscates content from the viewpoint of the CDN, which also allows the CDN to claim plausible deniability when served with a subpoena. Despite only caching encrypted and obfuscated content and identifiers, the CDN still has knowledge of which clients are requesting any content and how often they are requesting content. Unfortunately, this approach raises questions: are anonymizing proxies/VPNs trustworthy?, should clients be trusted to know k?

B. System of Proxies

While a VPN or anonymizing proxy may hide a client's identity from the CDN, it puts a large amount of trust in the VPN provider (or proxy), which has knowledge of each client's IP address, and what content each client is requesting.

[1]Most CDNs allow the publisher to decide on a push or pull model, but this makes no difference in our system design.

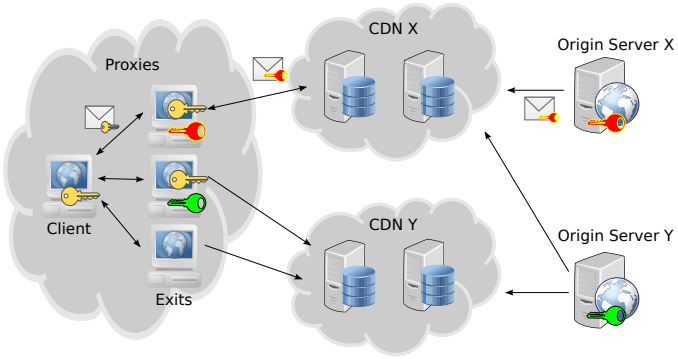

Fig. 2: The relationships between clients, the CDN, proxies, and content publishers in OCDN.

This can be addressed by using a system of proxies instead of a VPN provider; by introducing a set of proxies that requests are forwarded through before reaching the CDN, clients no longer have to trust a single entity (VPN provider), and can achieve a level of anonymity.

i) Hiding Clients' Identities: By replacing the use of an anonymizing proxy or VPN with the use of a series of proxies—some of which are supplied by OCDN and some of which are simply other clients—, the system can provide unlinkability of client to her requests/responses. There are two distinct sets of proxies: 1) exit proxies and 2) client proxies. We discuss in Section **??** how these can be run by different entities. Exit proxies provide a way of requesting encrypted content from CDNs; these proxies store the keys generated by content publishers (as described in the strawman approach). Client proxies provide a way for any client to route her traffic to an exit proxy, while hiding her identity. When requesting content, she generates a route that can include a series of client proxies and must include an exit proxy, and each client proxy forwards the request based on this route. In order for other clients to be unsure of the original requestor client's identity, the original requestor client can lie about the route; she can lie by including other client proxies in the route before herself. For example, if she has identity *C*, she could generate a route to exit proxy *E* that looks like

$$C \rightarrow G \rightarrow F \rightarrow E$$

She can hide her identity from *G* and *F* by using the route

$$D \rightarrow A \rightarrow C \rightarrow G \rightarrow F \rightarrow E$$

Neither *G*, *F*, or *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*. By using a series of client proxies, or even just that a client *can* use a series of client proxies, the identity of the client is hidden from other clients, exit proxies, and the CDN. Using both types of proxies is essential for a couple of reasons: 1) just using client proxies would disallow any client behind a NAT, or any mobile client from joining the system and 2) just using exit proxies would put too much trust in the proxies, as they would know the identity of each client, and what content each client is requesting.

j) Decouple Content Distribution from Decision of Trust: Using proxies also separates the issues of trust and content distribution. A client no longer needs to trust the CDN, which all other clients would need to trust as well. Now the client can simply trust the set of proxies to perform the correct actions, but does not need to trust anyone to keep her information private. All proxies in the system are completely separate entities from the CDN (and sometimes even from each other proxy). Therefore this design allows a client to decouple these issues, but still access cached content at the CDN.

k) Usability: Instead of each client having knowledge of each k for each domain, each proxy could have knowledge of each k. This means that clients would not have to keep track of secret knowledge, and overall, the number of entities that know this secret key is much smaller (as the number of proxies is significantly smaller than the number of clients). Lastly, the proxy can perform attack detection and prevention before the attack even reaches the CDN; more optimizations made possible by the use of proxies are discussed in Section **??**.

C. Key Use and Management

The design has evolved into a system where: 1) the CDN does not know the content, the content identifier, or which clients are accessing any content, 2) other clients do not know which client requested which content, and 3) the exit proxies do not know which clients are accessing which content (although, they do know what content is being requested by OCDN users). There are still some remaining security vulnerabilities that must be addressed. First, simply hashing the URL allows an attacker who simply wishes to guess whether the obfuscated URL is a specified plaintext URL or not. Second, the shared key k for each content publisher has the potential to be compromised, and therefore, OCDN must have a way to create new keys and replace old keys with the new ones.

l) Adding Secrecy to URL Obfuscation: According to the strawman approach, the URL is obfuscated via hashing. Unfortunately, an attacker could guess what the content identifier is by hashing his guesses and comparing with the hashes stored in the CDNs caches. To mitigate this vulnerability, the content publisher should incorporate the use of the shared key k in the hash of the URL by using a hash-based message authentication code (HMAC). This provides the property of fixed length identifiers, which does not reveal information about the plaintext identifiers, while obfuscating and preventing against guessing attacks.

m) Minimizing Shared Key Holders: If a single shared key k associated with a single content publisher gets compromised, then all proxies have a compromised k for that content publisher. A naive approach to address this is for the content publisher to generate a different k for each proxy; unfortunately, this defeats the purpose of CDNs caching content. The approach OCDN takes is for each content publisher to generate

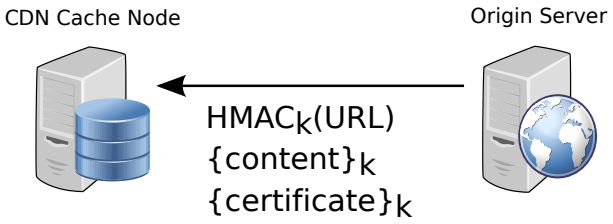

Fig. 3: How content is published in OCDN.

a single shared key k, and for a single proxy to hold that key; this maximizes the use of the CDNs caches. The shared key is assigned to an exit proxy based on consistent hashing [**?**], [**?**], which provides load balancing properties. *(Should we describe consistent hashing (at a high level)? Where is a good place for that?)* For more popular content, it may be useful to have more than one exit proxy serving that content; in this case, a few exit proxies can hold this key. We discuss this case in more detail in Section **??**.

n) Key Churn: To protect against attacks when a content publisher's shared key k is compromised, the publisher should generate a new key k and encrypt her content under the new key. Keys have an expiration timestamp that indicates when a key is no longer valid; the publisher should generate a new key when a current key has expired. Additionally, proxies should obtain the new key when a current key expires.

The resulting system is the basis of OCDN, and the high-level view of the system is shown in Figure **??**, which can be compared to what CDNs look like today (Figure **??**). The next section describes OCDN in more detail and outlines the specific steps for publishing and retrieving content.

V. OCDN Protocol

Based on the design decisions discussed in the previous section, we specify the steps taken to publish and retrieve content in OCDN.

A. Publishing Content

In order to publish content such that the CDN never sees the content, the publisher must first obfuscate her content, as described in Section **??**. Figure **??** shows the steps taken when publishing 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 publisher 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, such that this causes negligible padding overhead, but reduces the linkability between exact content length and content identification. 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. As long as the CDN does not have access to the shared key, then the CDN cannot see what content it is caching.

Now that the content is obfuscated, the publisher must also obfuscate the content's identifier. To do so, she computes the HMAC of the URL using the shared key k.

Once the identifier and the content replicas are obfuscated with k, they can be pushed to the CDN. Recently, services have cropped up to allow and help facilitate the use of multiple CDNs for the same content; a content publisher could use multiple CDNs' services. This mechanism could be used in OCDN to increase reliability, performance, and availability; a publisher can use a service, such as Cedexis [**?**], to load balance between CDNs. We discuss the use of multiple CDNs more in Section **??** on OCDN in partial deployment.

As the exit proxies use consistent hashing to divide keys among proxies while balancing load, the content publisher determines which exit proxy is correct (based on consistent hashing). The content publisher then encrypts the shared key k with the correct exit proxy's public key PK_{proxy}. The exit proxies' identifiers in this consistent hashing scheme are self-certifying names—the format of the exit proxy's identifier is IP:hostID, where hostID is a hash of the exit proxy's public key and the IP is the exit proxy's IP address. This technique was introduced in a self-certifying file system [**?**], and it allows for certification of the exit proxy. To distribute this value, the publisher then publishes $\{k\}_{PK_{proxy}}$ in its DNS Service (SRV) record, which allows the exit proxy to retrieve it (this is discussed in more detail in Section **??**). When the exit proxy retrieves $\{k\}_{PK_{proxy}}$ it must prove to the publisher that it is responsible for that publisher's content. This protects against any (malicious) exit proxy from requesting the shared key from all publishers.

o) Updating Content: For a content publisher to update content, she must follow similar steps as described in publishing content. Once she has updated the content on her origin server, she must obfuscate it using the same steps: 1) padding the original content length, 2) divide the content into fixed size blocks, and 3) encrypt the content blocks with the shared key k. Because she is updating the content (as opposed to creating new content), the obfuscated identifier will remain the same. She must retain a copy of the obfuscated old content until after the new content has been updated on the CDN; this is to prove that the old content owner is the same as the new content owner. The CDN cannot simply authenticate the content publisher, as this typically requires some type of identification for the content publisher; the CDN does not need to know — and should not know — the identity of the publisher, just that the organization that originally published the content is the same as the one that is updating the content. Only the origin and the proxy, both of which are outside the CDN, know the old obfuscated content, so an attacker cannot update the content that belongs to a legitimate publisher. The publisher must present the old obfuscated content to the CDN in order to also push her new obfuscated content to the CDN.

p) Updating Keys: A content publisher must be able to update keys in case of compromise. To minimize the amount of time a key is compromised for, the content publisher specifies an expiration date and time for the key when it is originally generated. The content publisher periodically checks if the key is valid or not based on the expiration timestamp. If the key is still valid, the content publisher continues to use it. Otherwise, the content publisher generates a new key k_{new}, computes $\text{HMAC}_{k_{new}}$(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 it's 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. Then the exit proxy decrypts it to obtain k_{new}, updates its version of the key, and proceeds as normal.

B. Retrieving Content

The steps taken for an end-user to retrieve a web page that has been cached by OCDN are shown in Figure **??**. In this section, we assume the client has already joined the system, which is described in more detail in Section **??**; at this stage, the client has knowledge of a subset of it's 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 she generates a source route, which includes some of her peers, and could potentially include a false originator (as described in Section **??**). 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 $\{\text{k}_{session}\}_{PK_proxy}$ to the request and encyrpts 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 $\{\text{k}_{session}\}_{PK_proxy}$ with its private key and stores the source route locally; it then decrypts the URL with $\text{k}_{session}$. The exit proxy then resolves the domain using it's local resolver, which will redirect it to the CDN's DNS resolver.

In order for the proxy to generate the obfuscated identifier to query the edge server for the correct content, it must have the shared key k that the origin server generated and obfuscated the content and identifier with. These steps are shown in Figure **??**. The origin server publishes the shared key encrypted with the proxy's public key in the DNS SRV record. When the exit proxy sends a DNS request to the origin server's authoritative DNS server, it includes its self-certifying identifier (IP:hostID). The origin server can hash the exit proxy's public key and verify it against hostID; this acts as a proof of the exit proxy's position in the consistent hashing circle. Additionally, the origin server can use the IP part of the exit proxy identifier to verify that the exit proxy is not in the same network as the CDN. If the origin is able to verify both of these, then it will send the DNS response, and the exit proxy will receive the encrypted shared key, which it can decrypt with it's private key.

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 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.[2]

Lastly, 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 multicast routing 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.

C. Clients Joining & Leaving

When a client joins OCDN, they 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. Similarly, when a client leave the system, this information is propogated to its peers using a gossip protocol.

D. Partial Deployment

OCDN should be partially deployable in the sense that if only some of the content publishers participate or only some

[2]Proxies can cache content in times of a flash crowd to minimize correlation attacks when a provider has encrypted and unencrypted content on the same CDN. This raises billing issues because the CDN cant charge as much if edge servers dont see as many requests for the origin; fortunately, RFC 2227 describes a solution for this [**?**].

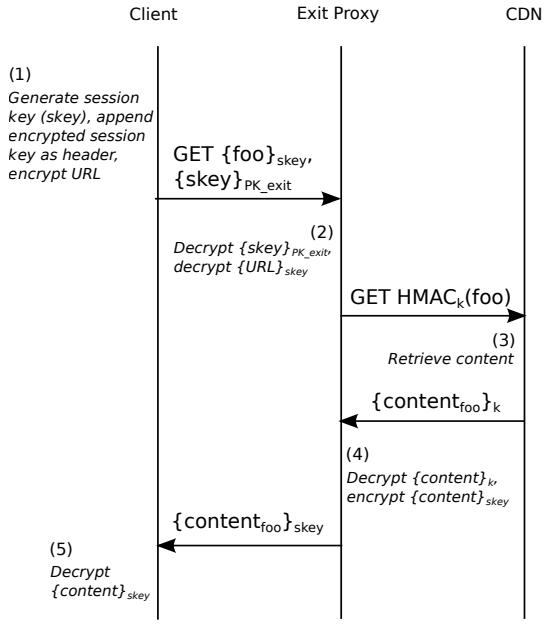

Fig. 4: Steps for retrieving content in OCDN when a client is prioritizing performance and goes directly to an exit proxy.

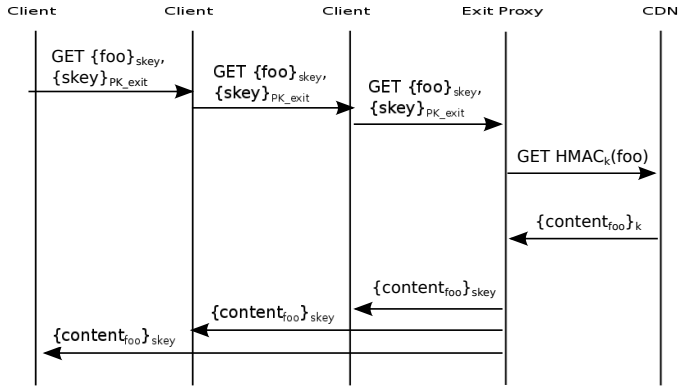

Fig. 5: Steps for retrieving content in OCDN when a client is prioritizing privacy and proxies her request through two other clients before reaching the exit proxy.

of the CDNs participate, then the system should still provide protections. We have two different partial deployment plans, and both provide protections for those publishers, CDNs, and clients that use OCDN.

q) Deployment Option 1: One option for deploying OCDN is to ensure there is some set S of content publishers the 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. Recall that there are n shared keys, resulting in n replicas of the content that *appear* to the CDN as different content (because each replica is encrypted with a different key). This allows the minimum set of publishers S to be relatively small; S must be greater than one, otherwise the CDN can infer that a

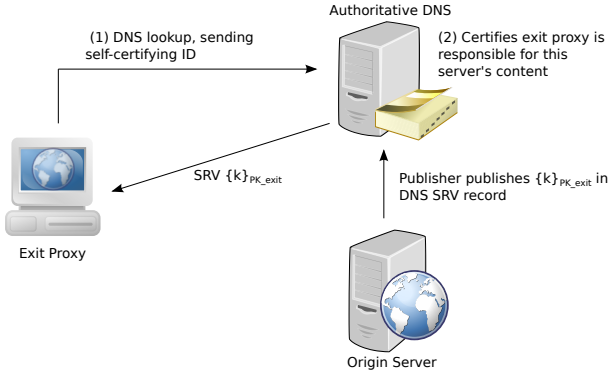

Fig. 6: Steps for certifying exit proxies and distributing shared keys to exit proxies.

client accessing this obfuscated content is actually accessing content that can be identified. This partial deployment plan partially protects the privacy of the clients accessing the content created by the set of publishers S. It does not protect the clients' privacy as completely as full participation of all publishers in OCDN because the CDN can still view cross site browsing patterns among the publishers 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 proxies, but not to clients.

r) Deployment Option 2: It is reasonable to believe that some content publishers are skeptical of OCDN and 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 publishers fully participating with only encrypted content, some other set of publishers partially participating with both encrypted and plaintext content, and some last set of publishers that are not participating. Unfortunately, if a publisher has both encrypted and plaintext content at a cache node, and some event causes a flashcrowd — the CDN sees a significantly larger spike in accesses to certain content — then the CDN can correlate the access spike on encrypted and plaintext content for the same publisher. In order to prevent this deanonymization of the content publisher, we can utilize multiple CDNs. The publisher can spread replicas over different CDNs such that the encrypted replicas are on one CDN and the plaintext replicas are on a different CDN. In this case the publisher is not susceptible to flashcrowds correlations and can still join the system.

E. Optimizations

While there are some optimizations that CDNs typically perform today that would not be possible with OCDN, the architecture of OCDN allows for new optimizations that are not possible in existing CDNs. Here we describe how OCDN limits current traditional CDN's optimizations, and then we

outline some ways in which OCDN can be optimized in terms of performance.

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 content that they cache, but this can only be done if the CDN has access to the decrypted content. Similarly, the CDN needs the decrypted content to perform minimizations on HTML, CSS, and Javascript files. While this likely 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.

s) Pre-Fetch DNS Responses: One way to increase the performance of OCDN is to pre-fetch DNS responses at the proxies. This would allow the 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 content is moved between cache nodes at the CDN, 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.

t) Load Balance Proxy Selection: As the proxy performs a number of operations on the client's behalf, it runs into the possibility of being overloaded. OCDN uses consistent hashing to generally address the load balancing problem, but an edge case that may still cause some exit proxies to be overloaded is extremely popular content. To mitigate this, multiple exit proxies can store the shared key k for the URL that is extremely popular, and therefore divide the load, while still taking advantage of the the CDNs caching benefits.

u) Use OCDN for Partial Content: Different content publishers have different needs, and each content publisher might have different needs for different content. The design of OCDN allows content publishers 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.

v) Different Modes of Operation: As briefly mentioned in Section **??**, there are two different modes of operation, where one provides better performance, and the other provides better privacy. In the first mode, the client can choose to send her request directly to the exit proxy, and not forward her request through any of her peers. While this allows the exit proxy to identify her, it does not allow the CDN to identify her or her request. In the second mode, the client forwards her request through a set of peers before it reaches the exit proxy. In this mode, the client can decide to prepend other clients' identifiers before her own, so it appears as if the request came from a different client. This provides unlinkability between the client and the request. This mode also provides the option of *only* prepending other clients' identifiers and then forwarding the request directly to the exit proxy; this provides the same performance benefit as the first mode, but also provides unlinkability. While this seems like the optimal solution, 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. These two modes of operation provide the client with different ways to use the system both based on their privacy preferences and based on the type of content they are requesting.

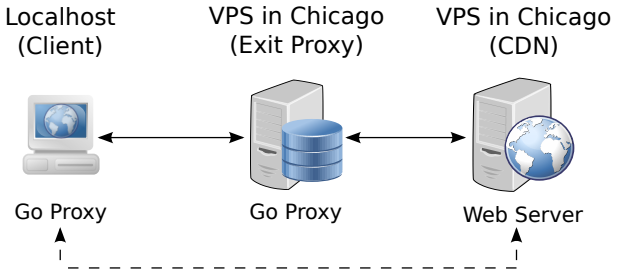

Fig. 7: The implementation of our OCDN prototype. The solid line represents how OCDN communicates between the components; the dotted line represents how a traditional CDN would communicate.

VI. 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 **??** 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. 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 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 [**?**], 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 and it 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

	Preserves Integrity at CDN	Preserves Confidentiality at CDN	Protects Client Identity
Stickler [**?**]	✓		
Repeat & Compare [**?**]	✓		
OCDN		✓	✓
Tor [**?**]			✓

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

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 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 who 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.

VII. Security Analysis

We analyze and discuss how OCDN addresses different attacks. Table **??** 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 nodes. OCDN handles this type of attacker by making the distribution of content requests appear uniform. The content publisher (of popular content) generates multiple shared keys and encrypts their content under each key, such that they have multiple, different-looking copies of their content. All of the content copies are pushed to the CDN and each key is shared with the exit proxies.[3] Now, the popular content does not appear as popular, and it makes difficult for an attacker to infer the popularity of the content.

[3]This also provides load balancing for exit proxies that hold originally held the sole key for the popular webpage, but this is now distributed across multiple exit proxies.

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.

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.

Flashcrowds. A flashcrowd is 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 the 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 time of a flashcrowd, such that the CDN (and therefore the attacker) does not see the surge in traffic.

VIII. Performance Analysis

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.

A. OCDN Overhead

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

Figure **??** shows the Time to First Byte (TTFB) for both OCDN and without OCDN. We can see the the TTFB using OCDN grows 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.

In addition to measuring TTFB, we measured the time it took to complete a request (with and without OCDN); the

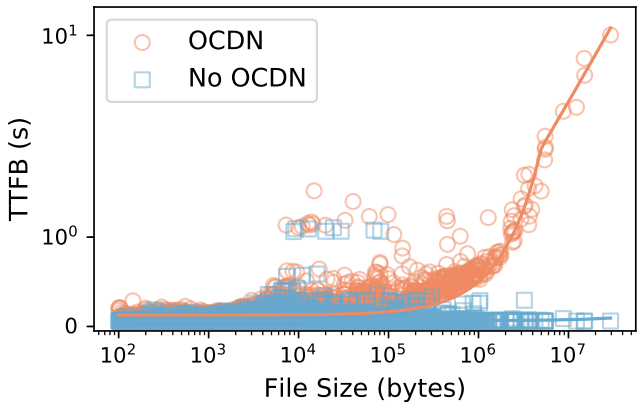

Fig. 8: Time to First Byte measurements with and without OCDN.

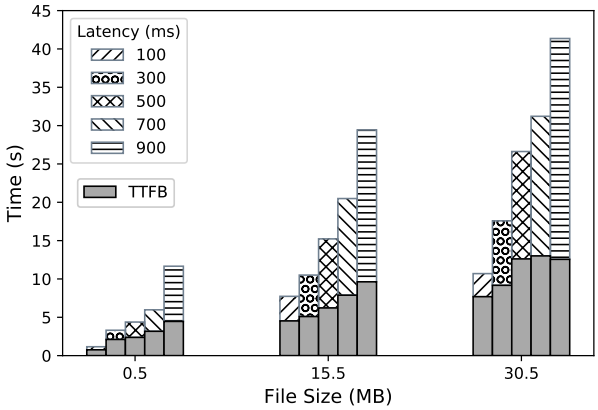

Fig. 10: Time to First Byte and time to complete a request with varying the file size and latency; increasing the latency is representative of increasing the number of client proxies on the path to the exit proxy.

results are shown in Figure **??**. Again, completion time grows linearly with file size, but for both OCDN and without OCDN; while both follow the same 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 **??**, 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 **??** 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 grows much slower as file size and latency increase; completion time grows more quickly than TTFB as the file size and latency increase.

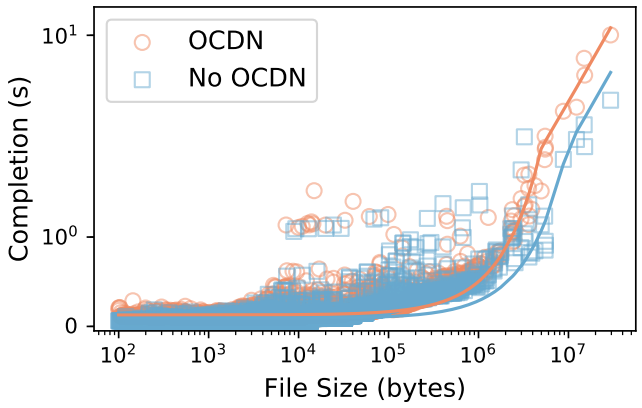

Fig. 9: Time to complete a request with and without OCDN.

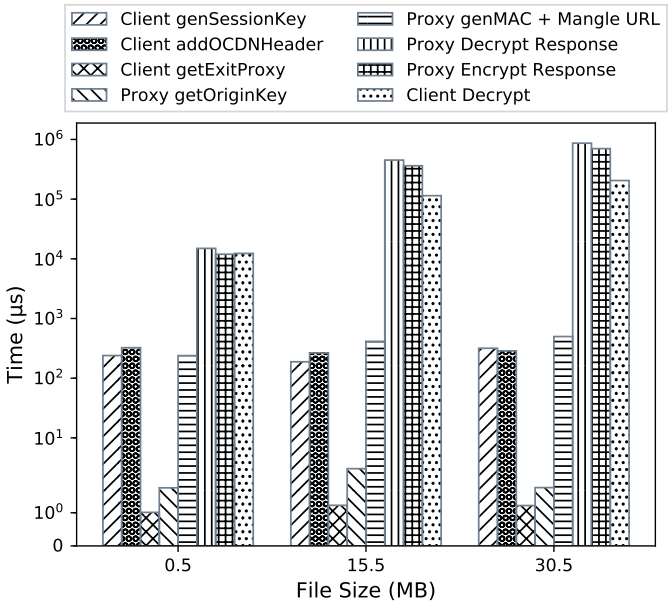

Fig. 11: Overhead of different operations performed by OCDN.

Lastly, we measure the performance overhead of the individual operations used in OCDN; figure **??** 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 $\text{HMAC}_k(\text{URL})$. 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 $k_{session}$.