Scalable and private media consumption with Popcorn

Trinabh Gupta*† Natacha Crooks*§ Srinath Setty‡ Lorenzo Alvisi* Michael Walfish†

*UT Austin §MPI-SWS ‡Microsoft Research †NYU

Abstract

This paper describes the design, implementation, and experimental evaluation of *Popcorn*, a media content delivery system that comprehensively hides (even from the content distributor) *what* is consumed but not necessarily *who* is doing the consumption. The motivation for Popcorn is both principled and pragmatic: we want to provide provable privacy while still respecting the current commercial context. To instantiate Popcorn, we turn to a powerful primitive from cryptography: private information retrieval (PIR). However, the cost and structure of PIR, as it appears in the literature, present major obstacles to using PIR as the foundation for an Internet-scale service. Nevertheless, with careful system design, and by composing a series of novel refinements and optimizations that leverage the properties of PIR protocols as well as the properties of media streaming, we have produced a system that cheaply hides media consumption, scales to the size of Netflix's library (8,000 movies) and respects current controls on media dissemination. The per-request cost in Popcorn is less than three times the per-request cost in a baseline system that does not provide privacy.

1 Introduction and motivation

This paper describes a Netflix-like media delivery system, Popcorn, that provably hides *what* is consumed by its users but not necessarily *who* is consuming. Popcorn is motivated by a fundamental tension in the ecosystem of online media consumption.

In one camp, there are people who are deeply uncomfortable about exposing their media diet to anyone or anything; a particular worry is a centralized media server (e.g., Netflix), as it is a target for capture, whether by hacking or subpoena. The discomfort is partly philosophical: privacy advocates observe that freedom requires the ability to consume privately [67], a right that public libraries, among others, have long fought to uphold [55]. But the discomfort is also practical: an entity with access[1] to a person's consumption profile can infer the person's sexual orientation, political leanings, private cultural affiliations, etc. [58, 59, 69]. And although many people do not share this discomfort—on the contrary, they *want* to expose their consumption, to gain recommendations—there may nonetheless be particular objects that they want to consume without others knowing.

Another camp observes that media often exists within a commercial framework. There are people who create it and services that distribute it, and these entities need to be compensated in order to sustain the ecosystem.

Our work in this paper responds to this tension by advancing a new design point in the realm of media consumption. Specifically, this paper asks the question, *Is it possible to build a system that hides content consumption while respecting current commercial arrangements, and if so, what would that system cost?*

A general solution is unlikely to be applicable to all media delivery systems, as they differ widely in their use case and requirements. YouTube's library, for instance, is large, frequently updated because of the high volume of user-generated content, and freely distributable. Netflix's library, by contrast, is comparatively small, updated infrequently [3], and subject to strict licensing and content protection requirements. This paper explicitly targets Netflix-like systems, while aiming to satisfy the following three requirements: *(1) provably and comprehensively hide content consumption; (2) scale; and (3) disseminate content in a way that mimics the status quo*. More specifically, we want to:

1. *Hide requests, in a way that is comprehensive and provable.* Content consumption must be hidden not only from a network eavesdropper [4, 29] but also from the content distributor. Our system must be wary of heuristic solutions as they can be exploited [12].
2. *Make it affordable, even at scale.* Our system should dispense privacy at an attractive price point. In practice, if the resource costs of guaranteeing private access were to be translated into currency and borne only by customers, they should result in no more than a small multiple of what customers pay to access content today.
3. *Respect current controls on content dissemination.* Given our pragmatic motivation, we are not trying to fundamentally reorient digital rights. Thus, our solution must be compatible with the existing commercial, legal, and policy regime (copyright, controls on content dissemination, etc.).

At first blush, systems that provide anonymity (defined informally as concealing the *who* of content creation or consumption, such as Tor [27]), satisfy the aforementioned requirements. However, these solutions conflict with the requirements of media delivery systems, which demand high bandwidth, low latency, and reliable streaming. Indeed, under a system that used Tor, a content provider would have to relinquish its control of resources to Tor nodes. Not only is

[1] To be clear, we are not challenging the trustworthiness of commercial media services. The issue is exposure to *other* parties, either accidentally, or through unlawful means.

the aggregate bandwidth available on a Tor network unlikely to be sufficient to accommodate every Netflix user, but using Tor nodes would force Netflix to rely on their altruism for reliable performance.

In light of this mismatch, Popcorn instead turns to a large body of cryptographic protocols known as *Private Information Retrieval*, or *PIR* (§2.2). These protocols [20, 31, 45, 62, 79] allow clients (content consumers) to request content from one or more servers (content distributors) without them inferring which items the clients requested.

These protocols are powerful, but applying them requires overcoming several challenges. First, to respond to a query, the server must compute over its entire library; otherwise the server would know what the client was *not* interested in, which would partially unmask the request. Another issue surrounds the choice of protocol. For instance, one type of PIR, called ITPIR [20], involves lightweight operations but demands multiple non-colluding servers and hence separate administrative domains; the plaintext content thus disseminates beyond its original distribution channel, conflicting with our requirements. Meanwhile, a different type of PIR, called CPIR [45], needs only one server, but the overhead is too high for our purposes. As we discuss in Section 7, there is a vast body of work that attempts to address some or both of these issues [7, 10, 16, 21, 23–25, 36, 37, 39–42, 50, 53, 56, 64, 71, 75, 77, 78], but to the best of our knowledge, no prior implementation is directly applicable to media delivery systems at the scale that we target.

Popcorn fills this void. It provably and cheaply hides media consumption, scales to the size of Netflix, and respects current content controls on media dissemination. To do so, Popcorn cherry-picks techniques from the PIR literature, tailors them to the specific domain of media consumption, and composes them with several novel optimizations.

Three techniques are central to Popcorn's design. First, Popcorn balances the trade-off between content protection and overhead by combining both types of PIR. Media objects, encrypted for content protection, are stored at multiple servers from distinct administrative domains and retrieved using the lighter-weight ITPIR. The much smaller cryptographic keys needed to decrypt those objects are retrieved using the heavier-weight CPIR. Second, Popcorn leverages the large numbers of concurrent users streaming content at any given time to batch requests and amortize the costs of PIR. Third, Popcorn mitigates the delay introduced by batching by exploiting the structure of the underlying PIR protocol (specifically, much of the query response work can be moved offline) as well as the nature of media streaming (specifically, progressive download).

Our experimental evaluation shows that, for a Netflix-sized movie library of 8,000 movies [3], Popcorn incurs modest resource overheads: the per-request cost to operate Popcorn in a popular cloud computing environment is, in terms of dollars, within a factor of three of a baseline system that does not provide privacy.

Though promising, Popcorn has several limitations. First, its overheads grow with the library size; this precludes scaling to media libraries that have more than a few hundred thousand media files (YouTube, for example, has at least a few million media files [19]). Second, it requires non-colluding servers for ITPIR, but we believe that this is not a severe limitation as content distributors already serve data from multiple CDNs [6]. Third, the current prototype lacks several features that are present in today's commercial streaming services: updates to the library, random seeks in a video, adaptive streaming based on available network bandwidth, etc. Section 8 describes how Popcorn can be extended to support some of these additional features.

Nonetheless, Popcorn is, we believe, the first system to demonstrate that users' media consumption can provably be hidden, even when scaled to the load of a commercial streaming service.

2 Setting and background

To provably hide users' media consumption, Popcorn relies on a family of cryptographic protocols known as PIR. In this section, we describe the setting in which Popcorn is intended to operate and provide the necessary background on PIR.

2.1 Scenario and threat model

The media delivery ecosystem has three principals: a *content creator*, a *content distributor*, and a *content consumer*. The creator (e.g., a movie company), delegates to the distributor (e.g., an online streaming service like Netflix) the tasks of disseminating content and charging consumers for it.

We model the content kept by the distributor as a collection L of n objects, each of ℓ bits; we call L the *library*. Since media objects are typically large (at least a few MBs [5, 35]), we assume that $\ell \gg n$. Associated with the library L is a one-to-one mapping between the integers $1 \dots n$ and the names of the objects in L. We assume that this mapping is known to both the distributor and the consumers; a consumer can therefore select to view a specific object by simply providing the distributor with the corresponding integer.

Threat model. We assume that both the content distributor and the network eavesdroppers are trying to identify the object that a consumer is retrieving. We assume that the distributor has full access to the content of the requests, and that the network eavesdroppers observe all communication between the distributor and the consumers. We do not consider side-channel attacks, such as using knowledge of where a consumer pauses playback, or of the content of the consumer's concurrent Web activity. Finally, we treat content integrity as an orthogonal problem; the content distributor may return incorrect content or no content at all. Such behavior would undermine correctness (defined in Section 2.2) but not privacy. The literature offers standard solutions to guarantee content integrity [30, 52, 68].

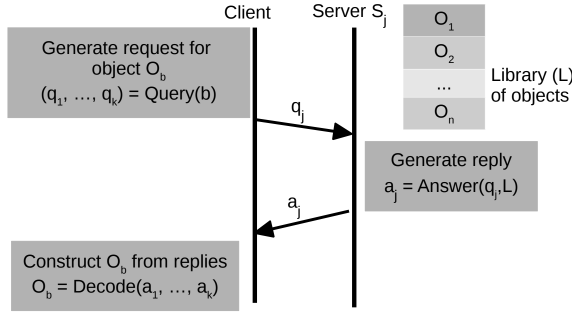

Figure 1—The structure of a PIR protocol.

2.2 Private Information Retrieval (PIR)

To achieve its goals, Popcorn leverages and refines the existing body of work on *Private Information Retrieval* (PIR), whose fundamentals we now quickly review. The high-level goal of PIR protocols is tightly aligned with that of Popcorn: they too allow a client to use an integer between 1 and n to retrieve any object from a library L of n ℓ-bit objects kept by a set of k servers ($k \geq 1$) without leaking to the servers any information about which object was retrieved.

A PIR protocol is structured around three procedures: Query, Answer, and Decode. To privately retrieve object $O_b = L[b]$ (see Figure 1), the client invokes $\mathsf{Query}(b)$ to output k query vectors $q_1, \cdots, q_k$, one for each server, and forwards q_j to server S_j ($1 \leq j \leq k$). Each S_j replies with $a_j = \mathsf{Answer}(q_j, L)$. Finally, the client computes $O_b = \mathsf{Decode}(a_1, \ldots, a_k)$ by applying the decode algorithm to the servers' responses.

Any PIR protocol must meet two properties [20]:

- ***Correctness.*** If a client requests the object in library L with index b, then the protocol indeed provides it with object $L[b]$.
- ***Privacy.*** After the server sees a query vector, its probability of guessing the client's requested index is no better than if the server had not seen the query in the first place. This property can be generalized to coalitions of $t < k$ servers, requiring that any t out of k servers jointly do not learn any information about the index of the requested object.

There is a trivial PIR protocol that meets these properties: a server transmits the entire library to the client. However, this protocol has high network costs. Thus, to the above two properties, a third is added.

- ***Communication efficiency.*** The size of a server's reply must not be much larger than ℓ and the size of a client's request must be far smaller than ℓ (though it is acceptable if there is some overhead above the minimum query size of $\log_2 n$ bits).

We discuss below the two types of PIR protocols that meet these properties.

Query (index b):
  for $i = 1$ to n **do**
    $f \leftarrow (i == b)\ ?\ 1 : 0$
    $c_i \leftarrow \mathsf{Enc}(pk, f)$
  return $q = (pk, c_1, \ldots, c_n)$

Answer (query vector q, library L):
  // Represent L as a matrix of d-bit integers:
  // $L \in (\{0,1\}^d)^{n \times (\ell/d)}$
  for $j = 1$ to ℓ/d **do**
    $r_j \leftarrow \prod_{i=1}^{n} c_i^{[L]_{i,j}}$
  return $a = (r_1, \ldots, r_{\ell/d})$

Decode (answer a):
  return $\mathsf{Dec}(sk, r_1), \ldots, \mathsf{Dec}(sk, r_{\ell/d})$

Figure 2—Query, Answer, Decode algorithms for a computational PIR (CPIR) protocol based on an additively homomorphic cryptosystem (Gen, Enc, Dec). (pk, sk) is a public/private key pair generated using Gen. n is the number of objects in the library L, and ℓ is the length of each object.

2.3 Computational PIR (CPIR) protocols

CPIR protocols require only a single, computationally bound server ($k = 1$). They are commonly constructed using additively homomorphic cryptosystems (they do not require fully homomorphic encryption [32]). Readers already familiar with these cryptosystems can skip the next paragraph, which provides a brief review.

Consider a public key cryptosystem defined by three algorithms—Gen for key generation, Enc for encryption, and Dec for decryption. Gen is a randomized algorithm that, given a desired key length, produces a corresponding public key pk and private key sk. Enc is also a randomized algorithm. Given a plaintext message m from a set G and a public key pk, Enc uses pk to encode m into a ciphertext message m_{pk} in a set G'. For concreteness, G and G' can be thought of as large subsets of the integers. Dec is the inverse of Enc; it takes a secret key sk and a ciphertext message m_{pk} and outputs the corresponding plaintext message m. Generally, such a cryptosystem is considered secure if it is infeasible, without knowledge of the secret key, to extract any information about the plaintext message encoded in a ciphertext, even with oracle access to Enc. Such a cryptosystem is *additively homomorphic* if $\mathsf{Dec}(sk, \mathsf{Enc}(pk, m_1) \cdot \mathsf{Enc}(pk, m_2)) = m_1 + m_2$, where m_1, m_2 are plaintext messages, $+$ is the binary operation representing addition of two plaintext messages, $\cdot$ is a binary operation (for example, addition, multiplication etc.) on the ciphertexts, and pk is a public key. An example of an additively homomorphic cryptosystem is the Paillier cryptosystem [63].

The CPIR protocol in Figure 2 uses an additively homomorphic cryptosystem to meet the three properties of PIR (§2.2) [62].

- ***Correctness.*** $\mathsf{Dec}(sk, r_j) = \mathsf{Dec}(sk, \prod_{i=1}^{n} c_i^{[L]_{i,j}})$, which equals $\sum_{i=1}^{n} \mathsf{Dec}(sk, c_i) \cdot [L]_{i,j}$ after the application of the

Query (index b):
// Generate the first $k-1$ query vectors randomly
for $j = 1$ to $k-1$ **do**
select $q_j \in_R \{0,1\}^n$
$e_b \leftarrow$ an n-bit string with all zeros except at b-th position
$q_k \leftarrow e_b \oplus q_1 \oplus \cdots \oplus q_{k-1}$ // $\oplus$ is bit-wise XOR
return $q_1, \ldots, q_k$

Answer (query vector q, Library L):
// q is one of the outputs of Query
// L has n objects; each is ℓ bits
// Represent q as a row vector and L as a logical matrix: $L \in \{0,1\}^{n \cdot \ell}$
return $q \cdot L$ // product over the two-element Galois field $GF(2)$

Decode (answers $a_1, \ldots, a_k$):
// a_j is the output of Answer
return $a_1 \oplus \cdots \oplus a_k$

Figure 3—Query, Answer, Decode algorithms for the ITPIR protocol of [20]. n is the number of objects in library L, and ℓ is the length of each object. k is the total number of servers.

additively homomorphic property. But $\forall i \in \{1, \ldots, n\} \setminus b$, $\mathsf{Dec}(sk, c_i) = 0$, by construction of c_i. Similarly, $\mathsf{Dec}(sk, c_b) = 1$. Therefore, $\mathsf{Dec}(sk, r_j) = \mathsf{Dec}(sk, c_b) \cdot [L]_{b,j} = [L]_{b,j}$

- ***Privacy.*** The guarantee that server S does not learn b hinges on S being computationally bounded. All server S sees is $q = (pk, c_1, \ldots, c_n)$. If, from it, S could systematically guess b (that is, guess which ciphertext is c_b), then S could likewise systematically guess which entry is the encryption of 1 (versus 0)—which would contradict the security properties of the underlying encryption scheme.
- ***Communication efficiency.*** The length of the server's reply is $(\ell/d) \cdot |c|$ bits, where ℓ/d is the number of ciphertexts in the reply and $|c|$ is the size of a ciphertext. $(\ell/d) \cdot |c|$ is comparable to ℓ, the size of object O_b, if the message expansion ratio, $|c|/d$, of the underlying additively homomorphic cryptosystem is small.[2] The client's request contains n ciphertexts and is thus of size $|c| \cdot n$. When $\ell \gg n$ and $|c|$ is a small constant (e.g., 2048 bits in most Paillier cryptosystem implementations), $|c| \cdot n$ is much smaller than ℓ.

2.4 Information-theoretic PIR (ITPIR) protocols

ITPIR protocols require more than one server, i.e., $k > 1$, and are therefore also called *multi-server* PIR protocols. These protocols assume that servers do not collude and thus require them to belong to different administrative domains.

Figure 3 shows the Chor-Goldreich-Kushilevitz-Sudan [20] (CGKS) ITPIR protocol. It meets the three properties of PIR (§2.2) as follows.

- ***Correctness.*** The output of Decode is $\bigoplus_{j=1}^{k} a_j$, which equals $\bigoplus_{j=1}^{k} (q_j \cdot L)$. But $GF(2)$ is a field, so multiplication distributes over addition, and addition is XOR. Thus, $\bigoplus_{j=1}^{k} (q_j \cdot L) = (\bigoplus_{j=1}^{k} q_j) \cdot L = e_b \cdot L = L[b]$.
- ***Privacy.*** A server in $S_1, \cdots, S_{k-1}$ sees a randomly generated query vector, and therefore cannot learn any information about b (in fact, servers $S_1, \cdots, S_{k-1}$ combined cannot learn any information about b). Server S_k sees q_k, which is constructed by XORing unit vector e_b with the one-time pad $q_1 \oplus \cdots \oplus q_{k-1}$. Unless S_k learns the one-time pad (or equivalently colludes with all other servers), it cannot learn any information about e_b because of the perfect secrecy properties of one-time pads.
- ***Communication efficiency.*** The length of a server's reply is ℓ bits, which is the size of the objects in L. The size of a client's request, which consists of k n-bit-long query vectors, is much smaller than ℓ bits when the number of servers k is small.

3 Challenges of using PIR

Though the PIR protocols described in the previous section (§2.2) can provably hide the content of requests, they are unable to do so at scale, or while respecting current controls on media distribution, for two reasons:

- *Incompatible PIR usage models.* Both types of PIR protocols are problematic. ITPIR (§2.4) requires multiple non-colluding servers, and thus multiple administrative domains, which means library content would have to disseminate beyond its original distribution channel, in apparent conflict with the requirement of respecting existing controls on dissemination. CPIR protocols (§2.3), in contrast, need only one server, but require expensive cryptographic operations.[3] This conflicts with the requirement of building an affordable system.
- *Extensive server-side work.* In CPIR protocols, the server's work to serve an object is linear in the size of the library: the server must load and process all n objects in the library. Similarly, in ITPIR protocols, all servers combined must on average compute over n objects to serve one of them. That is, in either type of PIR, each query induces $O(n)$ more work in expectation than in a standard media delivery service.

4 Architecture and design

To address the above challenges, and to scale PIR to commercial media systems, Popcorn combines existing techniques from the PIR literature, and specializes them to media delivery. The rest of this section describes Popcorn's architecture and design.

In Popcorn, each media file is split into *segments*, and the library is partitioned into *columns*, as depicted in Figure 4. A *segment* is a variable-sized contiguous piece of a media

[2]The Paillier cryptosystem has a message expansion factor of 2.

[3]Olumofin et al. [61] found that the fastest known CPIR protocols [8] are approximately 10-100× slower than ITPIR protocols [20, 36].

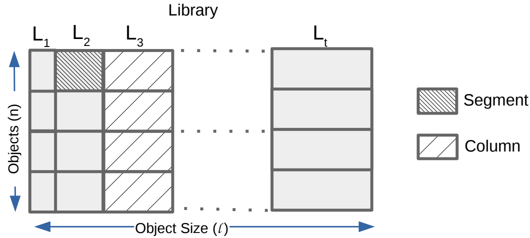

Figure 4—Popcorn Terminology

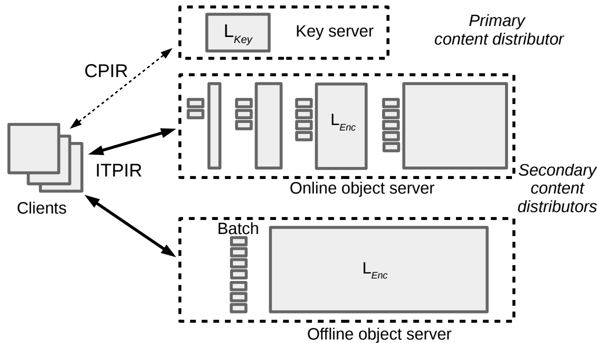

Figure 5—Architecture of Popcorn.

object containing, for example, a few seconds or minutes of a video. A *column* is the union of each of the corresponding segments for the n objects in the library (each is presumed to have the same decomposition into segments—which may require padding objects (§8)); therefore, a column's size is n times that of any given segment it contains.

Figure 5 depicts the architecture of Popcorn. A *primary content distributor* creates an encrypted version of the library, L_{Enc}, using *per-object keys*, and replicates L_{Enc} to *secondary content distributors*, each in separate administrative domains. The primary content distributor maintains a *key server*, and each secondary content distributor maintains an *object server* (Popcorn's current prototype uses two secondary content distributors).

The key server serves the per-object keys using CPIR, and the object servers deliver encrypted objects using ITPIR. The distinction between key and object servers maps to a similar distinction in DRM implementations used by media services today [1, 2, 49], where clients contact two separate servers: one serves encrypted video file segments, and the other serves the corresponding decryption keys. Netflix, for example, uses three different CDNs to serve files (Limelight, Akamai, Level3) but stores decryption keys on its own servers [6].

Object retrieval protocol. To retrieve an object, a client first retrieves the object's decryption key from the key server and then retrieves the encrypted object from the object servers. The retrieval of the encrypted object proceeds in two concurrently running phases. In the first phase, the client indicates its desire to download the encrypted object by sending ITPIR query vectors; in the second phase, in line with *progressive download* [6], the client downloads on-demand (i.e., at the appropriate playback time) pieces of the encrypted object.

A client starts the first phase (and indicates its desire to download an encrypted object) by concurrently sending a query vector for every column at each of the object servers. An object server runs an independent *instance* of ITPIR for every column that it maintains; these instances do not share or contend for resources. On receiving a request, an instance adds the query vector to its request queue; it periodically services this queue by computing and buffering an ITPIR reply for every request.

The second phase starts after a client has sent the query vectors; it runs concurrently with the server-side computation. In this phase, the client reconstructs fixed-sized *chunks* of its desired encrypted object (for example, every megabyte in the object is a chunk). To do so, the client downloads from the object servers the corresponding ITPIR-encoded chunks (which are buffered at the server, as described above).

The primary challenge in designing the components of Popcorn lies in balancing the competing concerns of content protection, cost, and the real-time nature of media consumption. To this effect, Popcorn relies on three main techniques. First, as described earlier, Popcorn combines CPIR and ITPIR. CPIR is used at the key server to avoid distributing keys beyond the primary content distributor while ITPIR is used to serve the large encrypted objects. Second, Popcorn relies on batching at the object servers to amortize the I/O cost. By exploiting the sequential consumption pattern of media streaming, Popcorn mitigates the tension between wanting large batch sizes to amortize work, and having requests meet their deadlines. Third, Popcorn exploits the structure of the ITPIR protocol to push the work of one of the object servers offline, amortizing cost further.

The rest of this section describes how, taken together, these techniques allow Popcorn to scale to the demands of a commercial media service.

4.1 Composing ITPIR and CPIR

To provide content protection at low cost, Popcorn combines CPIR and ITPIR: the heavier-weight CPIR, which requires only one server, is used to serve per-object keys, while the lighter-weight ITPIR is used to serve the large encrypted objects. As a result, both keys and objects are served privately (because PIR is applied to them both), CPIR is not a performance bottleneck (because it is used only for small keys), and current controls on content protection are respected (because the plaintext content and keys are stored only at the primary content distributor). The main challenge that Popcorn still faces is addressing the overhead of ITPIR.

4.2 Batching

Popcorn uses the CGKS ITPIR scheme described in Section 2.4, as that protocol relies on extremely cheap operations (XORs) and as such, has low computational overhead. Yet, implemented naively, responding to a query requires reading from storage $n/2$ segments on average (corresponding to the bits set to 1 in the query vector) and then XORing them. This taxes I/O bandwidth, memory bandwidth, and CPU cycles.

To reduce costs, Popcorn thus chooses to batch queries. Two observations motivate batching. First, a query vector is dense: on average, half of its entries are set to 1 (§2.4, Figure 3); as a consequence, generating a reply to a query requires the server to read a significant number of segments. In fact, to exploit I/O bandwidth, a natural implementation of the server's query reply procedure would be to read the entire column (in large chunks[4])—even though this means loading into memory the segments whose corresponding entries in the query vector are 0. This brings us to the second observation: since the server is reading the entire column anyway, handling further queries generates no additional I/O work. Thus, the server can amortize the cost of reading a column over a batch of queries.

Batching not only amortizes I/O overhead but also (1) reduces computational overhead and (2) enables efficient parallelization of the computational task. Popcorn, like others [13, 50], makes the observation that the PIR computation required for a batch of requests can be expressed as matrix-multiplication ($q \cdot L$ in Figure 3 can be replaced by $Q \cdot L$, where Q is a matrix whose rows are query vectors). Therefore, once a chunk of a column has been read into memory, the required processing can be done via block matrix multiplication, which not only reduces work by leveraging better cache locality (one can view this as a form of batching at the CPU/memory interface), but also parallelizes it by making use of both multi-core processors and SIMD instructions.

It may then seem attractive to design Popcorn in a way that maximizes batch sizes, as huge batches can amortize I/O work dramatically. Unfortunately, the real-time nature of media delivery is at odds with aggressive batching, and places stringent constraints on when segments must be ready to be delivered to a client. Specifically, delay before starting playback should be small, and before a client finishes consuming a segment, the servers must be ready to start providing the next one (to avoid stutter). These constraints seemingly force small-size batches, which would negate almost all the benefits of batching.

4.3 Specializing batching for media delivery

To side-step these constraints, Popcorn looks to prior media-on-demand broadcast techniques, specifically *Pyramid Broadcasting* [76], and *specializes* PIR batching for media delivery. A client's initial delay (on top of the network delay) is given only by the time it has to wait before it can start downloading the first segment. This delay is small if the processing cycle (and consequently the batch size) for the first column is small. Other segments, on the other hand, are not needed until later, and hence the requests for them (which are sent alongside the request for the first segment) can afford higher processing times. In effect, as depicted in Figure 6, this means that the server can afford a longer processing cycle for a later column, allowing requests to accumulate and batch sizes to grow.

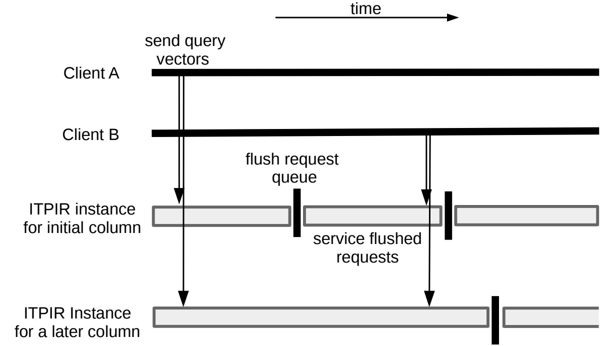

Figure 6—Batching at an object server in Popcorn. Requests for initial segments from two clients are serviced in separate batches as the processing cycle for the initial column is kept small to keep the initial delay low. Requests for a later segment from the same clients are batched together because the server can afford a longer processing cycle for the later column whose segments are not needed immediately.

Much of the work in Popcorn then goes to carefully choosing, for each column, the longest possible processing cycle (and consequently the largest possible batch size), such that users perceive a low initial delay and that the movie is played back smoothly (with no stutter). This requires guaranteeing two properties: (1) that the initial delay will always be smaller than the maximum tolerable initial delay (d), and (2) that the worst case delay before which segment i ($i > 1$) can be downloaded will always be smaller than the time to consume segments 1 to $i-1$ [76].

Each of these properties will hold if the ITPIR instance for the i-th column starts processing a query for segment i within a time budget of $T_i = d + \sum_{j=1}^{i-1} t_j$, where t_j is the playing time of segment j, or put differently, these properties hold if both the computation and the I/O parts of the processing logic for the i-th column finish in time T_i. This directly upper-bounds the segment size for the i-th column, which in turn (because T_{i+1} depends on t_i) bounds the length of the processing cycle (and the batch size) for the subsequent column. We now delve into what those bounds are.

Denote the target playback rate by μ, and the request rate by λ. Let p_i be the processing throughput (that is, the rate at which the CPU XORs data) available for column i, r be the

[4] In Popcorn, a column is stored as thin vertical slices that are successively read into memory and processed.

Server resource	Naive impl. of an ITPIR server (§2.2)	Online's i-th column (§4.3)	Offline server (§4.4)
Storage			
Bandwidth	$\lambda \cdot T \cdot n \cdot \mu$	$\alpha \cdot n \cdot \mu + 2 \cdot \lambda \cdot t_i \cdot \mu$	$n \cdot \mu + 2 \cdot \lambda \cdot T \cdot \mu$
Space	$n \cdot \ell$	$n \cdot \ell_i + \lambda \cdot t_i \cdot \ell_i + \lambda \cdot T_i \cdot \ell_i$	$n \cdot \ell + m \cdot R \cdot \ell$
CPU			
Processing throughput (p_i)	$\lambda \cdot T \cdot n \cdot \mu$	$\lambda \cdot t_i \cdot n \cdot \mu$	$\lambda \cdot T \cdot n \cdot \mu$
Network			
Bandwidth	$\lambda \cdot T \cdot \mu$	$\lambda \cdot t_i \cdot \mu$	$\lambda \cdot T \cdot \mu$

n = number of objects in the library (§2.1)
ℓ = size of an object (§2.1)
t_i = playing time of i-th segment of an object (equals ℓ_i/μ) (§4.3)
T_i = the playing time of the first $i-1$ segments of an object (i.e., $\sum_{j=1}^{i-1} t_j$) (§4.3)
λ = request rate (§4.3)
α = a parameter greater than 1 (§4.3)
μ = object playback rate
ℓ_i = size of i-th segment (defined in Figure 4) of an object (equals $t_i \cdot \mu$)
T = playing time of an object (equals ℓ/μ) (§4.3)
R = number of registered clients
m = number of precomputed replies (§4.4) buffered per client

Figure 7—Provisioning requirements of Popcorn.

bandwidth available to read the column (we assume that each column gets the same read bandwidth), and b_i be the batch size, given by $b_i = \lambda \cdot T_i$ (during a processing cycle that lasts for T_i, $\lambda \cdot T_i$ requests are accumulated). The bound on t_i is derived by relating load (how much work needs to be done by the deadline) to capacity (how much work can be done by the deadline). Specifically, the amount of data processed for a batch (batch size times the column size, or $b_i \cdot t_i \cdot \mu \cdot n$) is upper-bounded by the total amount of data that can be processed in the time budget T_i, and similarly, the amount of data read for a batch (column size, or $t_i \cdot \mu \cdot n$) is upper-bounded by the total amount of data that can be read in T_i. This results in the following relations:

$$(\lambda \cdot T_i) \cdot t_i \cdot \mu \cdot n \leq T_i \cdot p_i, \quad (1)$$

$$t_i \cdot \mu \cdot n \leq T_i \cdot r \quad (2)$$

From the second inequality, we can derive the bound for t_i as $T_i \cdot \frac{r}{\mu \cdot n}$ provided that there is sufficient processing throughput available (i.e., the first inequality holds). As the column number increases (i.e., for greater i), the segment sizes increase (because T_i is an additive sequence). For the same reason, batch sizes ($b_i = \lambda \cdot T_i$) also increase with i. The parameter $\frac{r}{\mu \cdot n}$, which we term α, captures the rate at which segment sizes grow. It must be at least one to ensure that the read bandwidth r is not less than $\mu \cdot n$, the minimum rate at which data must be read due to the n times overhead of PIR.

Provisioning requirements of Popcorn In practice, a system administrator for Popcorn would be given a library and a request rate, and would have to provision the ITPIR instances of the object servers in Popcorn. For each instance, its column size (t_i), time budget (T_i), and I/O bandwidth and processing throughput (r and p_i) can be derived by using the relations described above, once the values for d (maximum initial delay) and parameter α have been chosen. Figure 7 summarizes these provisioning requirements. To the I/O bandwidth r, we must add the bandwidth required to buffer and serve replies (i.e., support the second phase of the object retrieval protocol), which equals, for the i-th column, $2 \cdot \lambda \cdot t_i \cdot \mu$ (the sum of the rates at which reply data is generated and served). Buffering replies also requires additional storage capacity (on top of the column size), which equals, for the i-th column, $(\lambda \cdot t_i + \lambda \cdot T_i) \cdot (\mu \cdot t_i)$ (the number of replies that are being read or written times the size of a reply).[5]

4.4 Moving work offline

One of the main limitations in the aforementioned design is that, because the time budgets for the first few columns are small, the batch sizes for these columns are also small and the per-request I/O is high. To mitigate this effect, Popcorn observes that much of the work for one of the servers can be moved offline by exploiting the structure of the ITPIR protocol. As a result, for one of the two servers, there can be a single large column—of the size of the library—with a long processing cycle and a large batch size.

Recall that in the CGKS ITPIR scheme (§2.4), a client generates $k-1$ query vectors $q_1, \ldots, q_{k-1}$ uniformly at random, and then sets q_k so that $\bigoplus_{i=1}^{k} q_i = e_b$, where e_b represents the unit vector corresponding to the desired index. Only q_k depends on b; the first $k-1$ vectors are independent of the requested object. For our setting of $k = 2$, if we arrange for the client to direct the index-independent query vector to one of the two servers, then that server can compute the reply offline.

We hence modify the protocol outlined at the start of this section to distinguish between an *online* object server and an *offline* object server. When a client uses Popcorn for the first time, it generates a configurable number (m) of random query vectors which it sends to the offline server. This server precomputes (and stores locally) replies for these query vectors.

[5]In the current prototype of Popcorn, an ITPIR instance uses separate storage to store the column and the replies, to avoid competition between the two types of I/O.

The per-request dollar cost in Popcorn is less than three times the per-request dollar cost in a system without privacy.	§6.3
80% of the per-request cost in Popcorn is the cost of transferring data over the network.	§6.3
Popcorn requires large objects and many concurrent clients to effectively reduce costs.	§6.2

Figure 8—Summary of main evaluation results.

Later, when a client wants to download an object, it downloads (pieces of) a pre-computed reply from this server and piggybacks a new query vector to replenish the set of pre-computed replies.

The offline server is provisioned such that it has a single column with a segment size and a processing cycle (time budget) of T. This means that if λ is the request rate, then $\lambda \cdot T$ is the batch size. Figure 7 shows the resulting I/O bandwidth, storage space, and processing throughput requirements for the offline server's column.

5 Implementation

We have fully implemented Popcorn. The key server implementation uses the Trostle-Parrish CPIR protocol [75] and is borrowed from PIRMAP [53].[6] It is written in Java and is 1200 lines. For the object servers and the client-side code, we borrow the CGKS ITPIR implementation in Percy++ [37], modify it to support progressive download, and extend it with Popcorn's techniques described in Section 4. The resulting code base has a total of $8,000$ lines of C++, out of which 7,100 lines are for the object servers. The clients and the object servers communicate using Open Network Computing (ONC) RPC [74]. The library objects are encrypted using AES with 128-bit keys.

6 Evaluation

Our evaluation answers the following questions:

1. Can we make Popcorn affordable at scale, and if yes, for what configurations is it affordable?
2. What is the resulting price of privacy?

Figure 8 summarizes our evaluation results.

Setup. We compare Popcorn to NoPriv—a baseline media delivery system based on progressive download [6] that does not provide privacy. For completeness, we also compare Popcorn to BaselinePIR—a straightforward progressive download aware implementation of CGKS ITPIR (this is essentially Popcorn without the techniques described in Section 4). Figure 9 gives details of these baselines.

[6]Attacks have recently been discovered on this protocol [48]. We plan to port our implementation to the recently released XPIRe [7] CPIR library, which is based on lattice hardness assumptions. We expect the change to produce the same or better performance.

System	description
NoPriv	Apache webserver (version 2.4.10) that serves 1 MB *chunks* of library objects. This models an implementation of today's media delivery systems [6].
BaselinePIR	Two ITPIR servers, each of which splits up objects into 1 MB segments (and the library L into corresponding columns). Each server receives PIR queries directed to individual columns, and on receiving a query, spawns a new thread to do the CGKS ITPIR computation (Figure 3) for that column.
Popcorn	Section 4

Figure 9—Description of evaluated systems

	Type	vCPUs	RAM	SSDs	Network	Cost/hr
c3.4xl	1	16	30	2×160	1	$0.314
c3.8xl	2	32	60	2×320	10	$0.628
i2.8xl	3	32	244	8×800	10	$1.690

Figure 10—Hourly cost of reserved Amazon EC2 instances. Instances starting with "c" are compute-optimized, and the instances starting with "i" are I/O-optimized. RAM and SSD capacities are in GB and network bandwidth is in Gbps. Amazon charges an additional $0.05 per GB of data transfered to the Internet (assuming that the total volume is high).

For the three system variants, we measure resource usage in terms of CPU time, I/O transfers, amount of storage space used, and network transfers. We measure CPU time by instrumenting code with `clock()`, I/O transfers and amount of storage space using `iostat`, and network transfers using kernel network accounting (`/proc/net/dev`).

In each experiment, we run a system variant in one of the different configurations based on the following parameters: number of objects in the library (n), object playback time (T), and number of concurrent clients ($\lambda \cdot T$). We use the following values for the parameters: 2048, 8192 for n; 1 minute, 10 minutes, 60 minutes for T; and 1, 1000, 10,000 for the number of concurrent clients. These clients arrive according to a Poisson process ($\lambda \cdot T$ arrive in time T); on arrival a client issues a request for an object (which lasts for time T). The index of requested objects follows a Zipfian distribution ($\theta = 0.8$). The aforementioned workload models that of today's media delivery services [73]. We set the playback rate μ to 1 Mbps and the maximum tolerable initial delay to 30 seconds.

Our testbed is Amazon EC2. We choose to use compute and I/O-optimized instances of EC2 as the system variants we evaluate are either CPU bound or I/O bound or both. Figure 10 describes these instances; the next subsections provide more details on the number of instances used for different experiment configurations.

Operation	Data size (GB)	Time (s)	Throughput (Gbps)
Read	2	4.8	3.3
Read	8	18.5	3.5
MMul	2048	125.0	131.1
MMul	8192	444.6	147.4

Figure 11—Time taken by basic operations of reading data and computing matrix-matrix product as in ITPIR (§4.2). These microbenchmarks were conducted on an instance of type c3.8xl.

6.1 Using microbenchmarks to provision resources

Much of the work in Popcorn goes towards carefully provisioning the system to ensure smooth playback (§4.3, §4.4). In this subsection, we describe how one can go about identifying a satisfactory machine allocation for an experiment configuration.

Provisioning of Popcorn takes place in two steps: (1) benchmarking the basic operations in Popcorn, such as reading a column (r in §4.3) and processing it (p_i in §4.3), and (2) combining these results with the model presented in Figure 7. For example, consider provisioning the offline server of Popcorn (specifically, offline server minus the drives required to buffer and serve replies), for a library with $n = 8192$ objects, each $\ell = 450$MB in size (i.e., playback time $T = 60$ min at playback rate of $\mu = 1$Mbps) and $\lambda \cdot T = 10{,}000$ concurrent clients. According to Figure 7, this configuration requires a read bandwidth of $n \cdot \mu = 8$ Gbps to read the column, storage capacity of $n \cdot \ell = 3.5$ TB to store the column, and a processing throughput of $\lambda \cdot T \cdot n \cdot \mu = 80,000$ Gbps. Using the data from the microbenchmarks (Figure 11), this translates to using a minimum of three drives to satisfy the read bandwidth requirement, and using 543 CPUs to match the required processing throughput. Since the resource requirement is skewed towards CPU, we choose to instantiate this setup by allocating 18 type 2 compute-optimized instances (each instance contributes 31 CPUs to processing and reserves 1 CPU to I/O).

6.2 Overheads of Popcorn

In this subsection, we report the per-request overheads of Popcorn over the two baselines and describe how they change for different experimental configurations. This gives us an idea of configurations for which Popcorn's overheads may be affordable. We find that Popcorn is most effective when object sizes are large and there are many concurrent clients.

Overhead versus number of concurrent requests Figure 12 shows the per-request server-side CPU time and I/O transfers for the three system variants for a library with 8192 objects, each 450MB in size (each object has a playback time of 60 minutes at 1Mbps) as a function of the number of concurrent requests ($\lambda \cdot T$).

When there are no concurrent requests (i.e, a very small request rate), we find that Popcorn's overheads are the same as BaselinePIR as there is no opportunity to amortize costs by batching requests. Indeed, for each request, both systems read and process the entire library twice, once for each object server. As the request rate increases, Popcorn's overheads are amortized: I/O transfers are amortized because of batching (§4.3, §4.4) while CPU overhead decreases because of better cache locality of block matrix multiplication (§4.2). For instance, going from 1 to 10,000 requests, the per-request amortized I/O in Popcorn decreases from approximately 7 TB to 6 GB (1.5 GB of which is for buffering of replies), a reduction of 1200×. Note that this is less than the maximum possible reduction of 10,000× as batch sizes for the columns at the online server are smaller than 10,000 (the online server has five columns, among which the average batch size increases from 63 for the first to 2500 for the fifth (§4.3); the offline server, on the other hand, has a single column with a batch size of 10,000 (§4.4)); Likewise, going from 1 to 10,000 requests, the per-request CPU time decreases by a factor of approximately 6; the 365 seconds of processor time is in accordance with the performance of the matrix-multiplication microbenchmark in Figure 11. (7 TB of data processed in 365 seconds gives a throughput of 157 Gbps).

Popcorn's per-request I/O is higher than NoPriv as our workload in NoPriv always requests the same object (to give maximum caching benefit to the baseline), resulting in no I/O transfers. Popcorn's per-request CPU time is also much higher than NoPriv as the Apache webserver in NoPriv directly serves 1 MB pieces of an object. This requires almost no server-side processing (constant time lookup). In contrast, Popcorn must XOR n objects (on average for the two servers combined) to serve a single request.

Besides the I/O and CPU overhead, Popcorn incurs both network and storage space overhead. Like BaselinePIR, Popcorn incurs a two-fold network overhead over NoPriv: there are two servers in Popcorn and BaselinePIR from which a reply has to be downloaded (compared to one server in NoPriv). With respect to storage space overhead, Popcorn needs to buffer entire replies at each server (note that the extra I/O transfers are included in Figure 12).

Overhead versus number of objects Figure 13 shows the per-request server-side CPU time and I/O transfers for Popcorn for 2048 and 8192 objects while keeping the number of concurrent clients and the size of objects fixed to 10000 clients and 450 MB respectively. Note that the depicted I/O strictly refers to reading the library (in contrast to Figure 12 which includes the I/O transfers associated with buffering replies).

We expect Popcorn's overheads to scale linearly (because the server's computational and I/O work is proportional to n). The data that we have is consistent with this hypothesis; future work is to understand our prototype's scalability better, via experiments with object numbers between 2048 and 8192, and beyond 8192. Network transfers and server-side storage

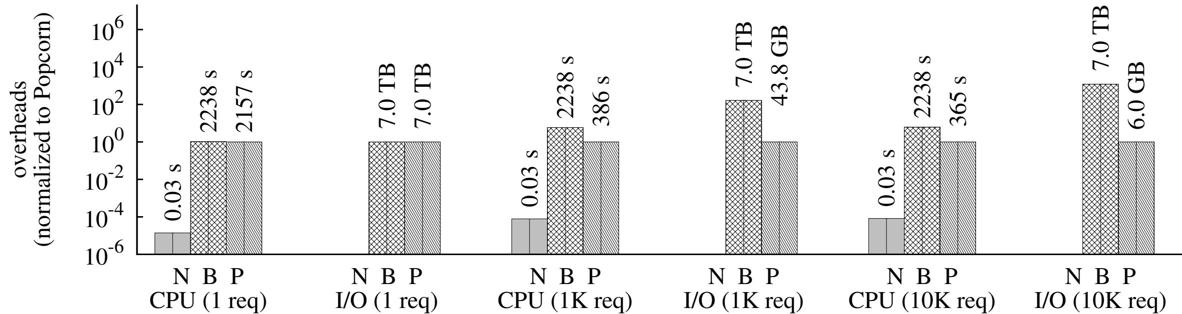

Figure 12—Per-request server-side resource use of NoPriv (N), BaselinePIR (B), and Popcorn (P). The bar heights represent resource use normalized to Popcorn. The labels indicate the absolute values. Provisioning AWS instances for BaselinePIR for more than a few requests is financially prohibitive, so we take the per-request overhead for 1 request and assume it remains constant with the number of concurrent requests.

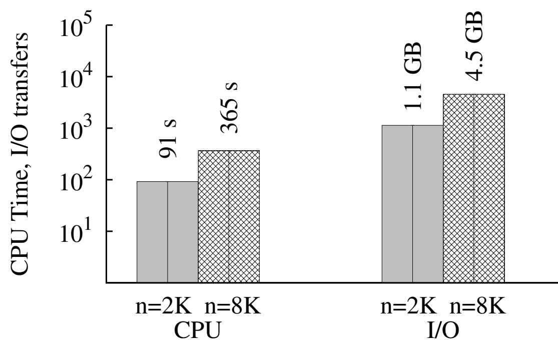

Figure 13—Change in per-request resource use of Popcorn with the number of objects. The I/O transfers are in the unit of MBs and the CPU time is in seconds. The left bar is for $n = 2048$ and the right for $n = 8192$. The y-axis is log-scaled.

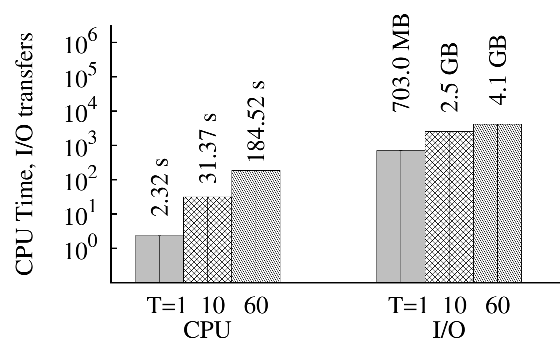

Figure 14—Change in per-request resource use of the online server in Popcorn with the length of objects. The I/O transfers are in the unit of MBs and the CPU time is in seconds. The bars correspond to objects with a playing time of 1, 10, and 60 minutes. The y-axis is log-scaled.

space do not change with the number of objects.

Overhead versus playing time of objects Figure 14 shows the per-request server-side CPU time and library related I/O transfers for the online server of Popcorn for the three configurations of 1 minute, 10 minute and 60 minute objects while keeping the number of objects (8192) and the request rate (10000 per hour) fixed.

As expected, CPU work appears to scale linearly with the length of the object. With respect to I/O transfers, we quantify the ratio of data that Popcorn reads per request to the size of the object: for 1 minute objects the ratio is 93, and decreases to 34 for 10 minute objects, and 9 for 60 minute objects. Popcorn's design is such that the batches for the first few columns are small but grow for later columns (§4.3). Therefore, when objects are small and have only a few segments, Popcorn is relatively inefficient in terms of I/O transfers.

6.3 Dollar-cost analysis

The previous subsection showed that Popcorn takes a significant step towards reducing I/O and computation overhead when there are many concurrent requests and large objects These results provide the foundation for achieving privacy at low cost, a cost which we now quantify for our target use case of Netflix.

	experimental configuration				per-request costs ($)		
	#1	#2	#3	#reqs	server	network	total
NoPriv	0	0	2	10000	0.0	0.02	0.02
BaselinePIR	0	0	2	3	1.127	0.04	1.17
Popcorn	11	35	12	10000	0.005	0.04	0.05

Figure 15—Per-request estimated dollar cost for Popcorn, NoPriv and BaselinePIR. #1, #2, #3 refers to the type of AWS instance from Figure 10.

Method Our cost analysis uses the pricing model of Amazon EC2 (Figure 10). EC2 does not report per-resource cost (except for network transfers), but instead reports instance cost in dollars per hour. We take the instances used for the configuration of $n = 8192$ objects, $\ell = 450$MB ($T = 60$ minutes at $\mu = 1$Mbps) and 10000 concurrent requests from the previous subsections, and estimate a per-request dollar cost.

Specifically, we multiply, for each used instance, its hourly cost by the time it was used, and then divide the resulting total machine cost by the number of requests to get per-request machine cost. We add network transfer cost to the per-request machine cost to get the total per-request cost. To ensure that we do not overestimate the cost of NoPriv, perhaps because instances are not used optimally in the experiments, we assume a machine cost of 0 for NoPriv. Similarly, the cost analysis for Popcorn and BaselinePIR is only an estimate as instances may be underutilized or may not be the optimal choice for our experiments. But we note that our method does not underestimate the cost of Popcorn.

NoPriv To give NoPriv maximum benefit, we disregard machine cost. The per-request cost is then determined by the network transfer cost, which is approximately 2 cents (450MB per request at 5 cents per GB).

BaselinePIR Given that BaselinePIR is bottlenecked on I/O, we use the I/O-optimized EC2 instances to provision the servers. Specifically, we choose 2 type 3 instances (one for each object server). With this setup, each server can serve a maximum of 3 concurrent requests (in fact, servers could serve only 2 requests smoothly; we round up to be optimistic to BaselinePIR). This setup costs $1.13 per request (2 times $1.69/hour/instance divided by 3 requests an hour), rising to $1.17 after adding the network cost.

Popcorn Appropriately provisioning Popcorn requires provisioning both the offline and the online server. For the offline server, we use the machine allocation outlined in §6.1 (18 type 2 instances) plus the additional instances (6 of type 3) required to buffer and serve replies. The online server must satisfy the provisioning requirements outlined in §4.3, which allows for flexibility in how one chooses the number and size of columns. By choosing $\alpha = 2$, we get five columns for the online server, and by using the same process as described in §6.1, we get an EC2 instance allocation consisting of 10 type 1 instances, 17 type 2 instances, and 6 type 3 instances. To this we add another instance of type 1 for the key server. The setup described above yields a per request cost of $0.004 which, when combined with the $0.04 per-request network cost yields a total per-request cost of $0.05.

Summary Figure 15 summarizes the results of our analysis. With our estimates, we find the per-request cost for a no-privacy baseline to be $0.02, while the per-request cost of Popcorn is $0.05 (of which 80% is network cost). Providing privacy in Popcorn thus leads to an estimated 2.5× increase in dollar cost, in line with the affordability requirement in the introduction. Moreover, as in NoPriv, network cost is now the dominating factor.

Limitations Our cost analysis has two main limitations. First, it assumes that all objects are of the same size. It hence fails to consider the additional storage space and bandwidth cost of having to pad objects. Second, our calculations exclusively focus on the server-side cost, and fail to capture the additional cost at the client-side, both in terms of the additional computing to XOR the replies from the ITPIR servers, and bandwidth overhead of having to communicate with multiple object servers.

7 Related Work

Improving the performance of PIR. The computational challenges of PIR have been obvious since its introduction, and have since been mitigated in several ways.

The first response consists of distributing the work, either by moving it to the cloud [23, 53] or by distributing it among clients, in peer-to-peer fashion [64]. Neither approach, unfortunately, meets the demands of our setting. A simple back-of-the-envelope analysis shows that the $14 needed on Amazon EC2 to process a 1TB library after cleverly tuning an efficient CPIR protocol to run MapReduce [53] translates, in our setting, to a 10× to 100× increase in the cost of serving media content. And while, after partitioning the library among clients, processing queries in a peer-to-peer fashion helps address the load imposed by CPIR on server resources, it increases latency and defeats content protection.

A second line of investigation, motivated by a paper of Sion and Carbunar [70]—which noted that the cost of CPIR may be worse than the naive solution of transferring the entire library—has focused on reducing the computational load of CPIR [7, 8, 56, 75] not by reducing the number of operations performed by the server, but by cryptographic protocols that require less expensive operations. Their performance, however, is still insufficient for our purposes: while the authors of the recent XPIRe system [7] report that "...it is possible to privately receive an HD movie from a Netflix-like database (with 35K movies) with enough throughput to watch it in real time ...", their claim assumes that the server dedicates a full processor and a storage medium with 30Gbps of read bandwidth to *each* request—an intensive use of resources that does not match the economics of commercial streaming services.

Yet another response has been to circumscribe, in exchange for better performance, the portion of the library for which the privacy guarantees hold. For libraries that can be thought of as a matrix, bbPIR [77] allows users to specify a submatrix (called a bounding box) from which bits can be privately retrieved using CPIR. This can be useful for efficiently implementing privacy-preserving location-based services: the larger the bounding box, the higher the privacy, but also the higher the processing and network costs. Similarly, RAID-PIR [21] proposes to partition the library and run PIR on the partition of interest to reduce the computational and network overhead of PIR. Olumofin and Goldberg [60] put this approach on sound theoretical grounds by rigorously quantifying the degree of privacy lost under such circumscribing.

Perhaps the most direct way to reduce the overhead of PIR is to genuinely reduce the work that servers need to perform.

Lueks and Goldberg [50], building on earlier theoretical work by Beimel et al. [13] and Ishai et al. [43], show that one can achieve sub-linear server-side *computation* by efficiently processing batches of requests from multiple clients. As we had discussed in Section 4.2, Popcorn is inspired by this work and batches requests at multiple stages of its protocol to reduce not only computational cost, but, crucially, I/O cost.

Finally, one can improve performance through dedicated hardware [10, 42, 78], as first proposed by Smith and Safford [71]: a client connects to a secure coprocessor that (obliviously with respect to the server) retrieves the requested object from the library hosted by the server and delivers it to the client, using schemes similar to ORAM [38] (discussed below) and/or oblivious permutations [42]. These schemes, however, are not a good fit for our context, as they require the secure coprocessor to channel all requests (a likely bottleneck in applications like ours, with high request rates) and to have enough memory to store several large media objects, when the IBM 4764 coprocessor has only 64 MB of storage.

Besides reducing the computational overhead of PIR, a large body of literature focuses on reducing the communication overhead of PIR ([31, 62] survey these prior works). A work in this sub-area that is related to ours is that of Devet et al. [24]. This work proposes composing CPIR and ITPIR, however, the composition takes a different form and serves the purpose of communication efficiency. Specifically, Devet et al. [24] compose the two protocols hierarchically; their protocol uses ITPIR to select a sub-library and multiple rounds of CPIR to select an object within a sub-library. The work targets an environment in which $n \gg \ell$, and aims to save network overhead. As noted in Section 2.2, this is the opposite of our scenario. In particular, we assume media objects are large, so we are not concerned with PIR's network overhead. The techniques used Devet et al. to save network overhead would induce prohibitive computational cost in our setting.

Improving the robustness of PIR. A separate research direction investigates how to effectively prevent Byzantine servers [46] running an ITPR protocol from successfully returning an incorrect answer to a client's query. Research on this problem, first posed by [14], has pursued the dual goals of identifying the highest number of faulty servers that can be tolerated and of reducing the computational burden that fault-tolerance imposes on clients [25, 36]. Correctness concerns are orthogonal to our proposed work, which focuses on protecting client privacy. Nevertheless, while Popcorn will not mask Byzantine servers, it allows clients to detect when they have received the wrong object in response to their query.

Protecting library content in PIR. When using untrusted servers to run ITPIR, the content distributor is faced with the problem of protecting the library's content from unauthorized use and distribution (the issue is explained in Section 3). Gertner et al. [33], who first introduce the problem, propose protecting the content by storing at auxiliary servers independent random data that, when XORed, produces the library's content. Huang et al. [41] protect library content kept at untrusted servers by first encrypting it, and then using a threshold signature scheme [22] for serving keys for the encrypted object: only if more than a tunable threshold of servers collude can the library content be disclosed. Library content protection is also a focus in Popcorn. Indeed, in light of the significant economic payoff that colluding servers may reap in uncovering the library's content, Popcorn takes the extreme position that content protection should be collusion-proof. Section 4.1 discusses how Popcorn achieves its goal through a novel combination of CPIR and ITPIR.

SPIR schemes add an additional facet to content protection by preventing dishonest clients from learning information about the content of a database beyond what is contained in the records they retrieved [34]. While Popcorn does not use a SPIR scheme to privately download keys from the key server, we note that it is possible to transform any PIR protocol into an SPIR protocol using Oblivious Transfer (OT) [26, 57].

Alternatives to PIR for privacy. *Obfuscation* [12, 28, 66] defends clients' privacy by accompanying their requests with dummy requests that serve as cloaking or cover traffic. Compared to PIR, this approach requires less processing at clients and servers, but at significantly higher network cost. In our setting, matching the degree of privacy (the number of objects among which a request is hidden) offered by PIR would require issuing a prohibitive number of dummy requests.

Rather than concealing the content being consumed, *anonymity* hides the identity of the consumer [27, 47]. We see anonymity as complementary: unlike PIR that hides consumption, it hides metadata such as login times, download frequency, etc. However, anonymity based solutions can reveal access patterns, which, in combination with other background information, may not protect a user's media consumption [58].

Oblivious RAM (ORAM) [38, 51, 54, 72] algorithms also allow a client to conceal its access patterns. However, they are not directly applicable to media delivery systems as they rely on the assumption that the entity that downloads data can also modify data at the server.

Searchable Symmetric encryption (SSE) and protocols based on it [17, 18, 44, 65] are yet another alternative to hiding access patterns, however, unlike PIR, these protocols allow for a controlled amount of leakage in the form of data-access and query patterns.

8 Discussion, limitations and future work

We learned four important lessons from Popcorn:

1. Privacy is achievable under current content control and dissemination policies.
2. The costs are not necessarily prohibitive, even at scale.
3. We can quantify what these costs are.

4. Cryptography protocols traditionally viewed as expensive, can, with careful system design and implementation, be used in practical systems.

Popcorn is only a proof-of-concept and as such lacks functionality. Here we discuss ways to extend Popcorn to support the following: (1) updates to the library, (2) variable size objects, (3) variable object quality and client bandwidth, (4) more complex pricing models, and (5) targeted ads and recommendations.

Library updates. Popcorn should permit updating objects online. The time scale, given our setting of Netflix, is hourly or daily changes; we are not targeting the case of constant flux (as in YouTube). To handle updates, Popcorn must address two technical issues: first, updates invalidate precomputed replies at offline object servers (§4.4). A potential solution is to *recompute incrementally*; the XOR operation makes such incremental computation possible for insertions and deletions of rows (objects) in the library matrix. Second, correctness (§2.2) is lost if the object servers have different versions, which could happen during a transition. A potential solution would be for object servers to associate *generation numbers* with each version of the library.

Variable object sizes. Popcorn's design assumes that all the objects in the library have the same decomposition into segments. In reality, media objects have different sizes and playback times. A solution is to *logically pad objects*: each object is considered to be the length of the largest object in the given library. This padding is not literal: if the server is aware that an object has been logically padded, it can very efficiently handle queries against the padded piece. Unfortunately, this design forces the client to download as much data as the size of the largest file.

Variations in quality and bandwidth. Different clients demand different quality, bitrate, and tracks (subtitles, language, etc.). As a consequence, a modern media file is encoded in many different ways. Popcorn can handle variation in bandwidth and desired quality among clients by having different libraries for different bitrate encodings, sending query vectors to all of such libraries but downloading the corresponding segment only from one. This increases the cost of server side computation, but does not affect the network cost, which as we saw is the dominant factor.

Pricing models. Popcorn's current prototype can support a subscription-based pricing model in which the primary content distributor can charge a flat fee to an accessor of the key server. More advanced pricing models can be supported by modifying the way in which keys are served in Popcorn. Transforming the keyserver's CPIR protocol to be symmetric [26, 34, 57] would allow Popcorn to support pay-per-view, while support for different price for different objects could be added by relying on Priced Oblivious Transfer (POT) protocols [9, 15]. Finally, support for tiered pricing could be added by adapting Priced Symmetric PIR (PSPIR) protocols [40] to our setting (where keys are served from a single server).

Targeted ads and recommendation services Commercial streaming services use knowledge of customers' media diet to target advertisements or formulate recommendations, and these functions contribute to distributors' revenue. In its current form, Popcorn does not support targeted advertisement or recommendations. This limitation is not fundamental: existing work [11] shows how one can use PIR itself, in combination with other techniques, to deliver advertisements obliviously to an ad broker.

Acknowledgments

We thank Allen Clement, Alan Dunn, Yuval Ishai, Jaeyeon Jung, Brad Karp, Sangman Kim, Michael Z. Lee, James Mickens, Vitaly Shmatikov, and Emmett Witchel for feedback and comments that improved this draft. The Texas Advanced Computing Center (TACC) at UT supplied computing resources for an earlier version of this work. This work was supported by NSF grants 1040083, 1048269, 1409555, and 1055057; and a Google European Doctoral Fellowship.

References

[1] Digital Rights Management. `http://msdn.microsoft.com/en-us/library/cc838192%28VS.95%29.aspx`.
[2] Microsoft PlayReady. `http://www.microsoft.com/playready/`.
[3] Netflix USA: Complete Instant Streaming List of all Movies and TV Shows. `http://netflixusacompletelist.blogspot.com/`.
[4] The 2014 Pulitzer Prize Winners, Public Service: The Guardian US and The Washington Post. `http://www.pulitzer.org/works/2014-Public-Service`.
[5] Today's Media File Sizes – What's Average? `http://www.filecatalyst.com/todays-media-file-sizes-whats-average`.
[6] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang. Unreeling netflix: Understanding and improving multi-CDN movie delivery. In *INFOCOM*, 2012.
[7] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIRe: Private Information Retrieval for Everyone. Cryptology ePrint Archive, Report 2014/1025, 2014.
[8] C. Aguilar-Melchor and P. Gaborit. A lattice-based computationally-efficient private information retrieval protocol. *Cryptol. ePrint Arch.*, 446, 2007.
[9] W. Aiello, Y. Ishai, and O. Reingold. Priced oblivious transfer: How to sell digital goods. In *International Conference on the Theory and Application of Cryptographic Techniques*, pages 119–135. Springer-Verlag, 2001.
[10] D. Asonov. *Querying databases privately: a new approach to private information retrieval*, volume 3128. Springer, 2004.
[11] M. Backes, A. Kate, M. Maffei, and K. Pecina. ObliviAd: Provably secure and practical online behavioral advertising. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.
[12] E. Balsa, C. Troncoso, and C. Diaz. OB-PWS: Obfuscation-based private web search. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[13] A. Beimel, Y. Ishai, and T. Malkin. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *Journal of Cryptology*, 17(2):125–151, 2004.

[14] A. Beimel and Y. Stahl. Robust information-theoretic private information retrieval. In *Security in Communication Networks*, pages 326–341. Springer, 2003.

[15] J. Camenisch, M. Dubovitskaya, and G. Neven. Unlinkable priced oblivious transfer with rechargeable wallets. In *Proceedings of the 14th International Conference on Financial Cryptography and Data Security*, FC'10, pages 66–81, Berlin, Heidelberg, 2010. Springer-Verlag.

[16] J. Cappos. Avoiding theoretical optimality to efficiently and privately retrieve security updates. In *Financial Cryptography and Data Security (FC)*, pages 386–394. Springer, 2013.

[17] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rou, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. Cryptology ePrint Archive, Report 2014/853, 2014. `http://eprint.iacr.org/`.

[18] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. Cryptology ePrint Archive, Report 2013/169, 2013. `http://eprint.iacr.org/`.

[19] X. Cheng, C. Dale, and J. Liu. Statistics and social network of youtube videos. In *16th International Workshop on Quality of Service (IWQoS)*, pages 229–238. IEEE, 2008.

[20] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM (JACM)*, 45(6):965–981, 1998.

[21] D. Demmler, A. Herzberg, and T. Schneider. RAID-PIR: Practical multi-server PIR. In *Workshop on Privacy in the Electronic Society (WPES)*, 2014.

[22] Y. G. Desmedt. Threshold cryptography. *European Transactions on Telecommunications*, 5(4):449–458, 1994.

[23] C. Devet. Evaluating private information retrieval on the cloud. Technical Report 5, University of Waterloo, 2013.

[24] C. Devet and I. Goldberg. The best of both worlds: Combining information-theoretic and computational PIR for communication efficiency. Technical Report 7, University of Waterloo, 2014.

[25] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *USENIX Security Symposium (SEC)*, pages 269–283, 2012.

[26] G. Di Crescenzo, T. Malkin, and R. Ostrovsky. Single database private information retrieval implies oblivious transfer. In *EUROCRYPT*, pages 122–138, 2000.

[27] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The second-generation onion router. In *USENIX Security Symposium (SEC)*, 2004.

[28] J. Domingo-Ferrer, A. Solanas, and J. Castellà-Roca. h(k)-private information retrieval from privacy-uncooperative queryable databases. *Online Information Review*, 33(4):720–744, 2009.

[29] Electronic Frontier Foundation. NSA spying on Americans. `https://www.eff.org/nsa-spying`.

[30] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten. SPORC: Group collaboration using untrusted cloud resources. In *Symposium on Operating Systems Design and Implementation (OSDI)*, 2010.

[31] W. Gasarch. A survey on private information retrieval. In *Bulletin of the European Associateion for Theoretical Computer Science (EATCS)*, 2004.

[32] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, volume 9, pages 169–178, 2009.

[33] Y. Gertner, S. Goldwasser, and T. Malkin. A random server model for private information retrieval. In *Randomization and Approximation Techniques in Computer Science*, pages 200–217. Springer, 1998.

[34] Y. Gertner, Y. Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes. In *ACM Symposium on the Theory of Computing (STOC)*, pages 151–160, 1998.

[35] P. Gill, M. Arlitt, Z. Li, and A. Mahanti. Youtube traffic characterization: a view from the edge. In *Internet Measurement Conference (IMC)*, pages 15–28, 2007.

[36] I. Goldberg. Improving the robustness of private information retrieval. In *IEEE Symposium on Security and Privacy (S&P)*, pages 131–148, 2007.

[37] I. Goldberg. Percy++ project on SourceForge, 2012.

[38] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[39] R. Henry, Y. Huang, and I. Goldberg. One (block) size fits all: PIR and SPIR over arbitrary-length records via multi-block PIR queries. In *Network and Distributed System Security Symposium (NDSS)*, 2013.

[40] R. Henry, F. Olumofin, and I. Goldberg. Practical PIR for electronic commerce. In *Computer and communications security (CCS)*, 2011.

[41] Y. Huang and I. Goldberg. Outsourced private information retrieval. In *Workshop on Privacy in the Electronic Society (WPES)*, 2013.

[42] A. Iliev and S. Smith. Private information storage with logarithmic-space secure hardware. In *Information Security Management, Education and Privacy*, pages 201–216. Springer, 2004.

[43] Y. Ishai, E. Kushilevitz, R. Ostrovsky, and A. Sahai. Batch codes and their applications. In *ACM Symposium on the Theory of Computing (STOC)*, pages 262–271, 2004.

[44] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. Cryptology ePrint Archive, Report 2013/720, 2013. `http://eprint.iacr.org/`.

[45] E. Kushilevitz and R. Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *Symposium on Foundations of Computer Science (FOCS)*, pages 364–364, 1997.

[46] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM TOPLAS*, 1982.

[47] M. Z. Lee, A. M. Dunn, B. Waters, E. Witchel, and J. Katz. Anon-pass: Practical anonymous subscriptions. In *IEEE Symposium on Security and Privacy (S&P)*, pages 319–333, 2013.

[48] T. Lepoint and M. Tibouchi. Cryptanalysis of a (somewhat) additively homomorphic encryption scheme used in PIR. Cryptology ePrint Archive, Report 2015/012, 2015. `http://eprint.iacr.org/`.

[49] P. LLC. Analysis of netflix's security framework for 'watch instantly' service. `http://pomelollc.files.wordpress.com/2009/04/pomelo-tech-report-netflix.pdf`, Mar. 2009.

[50] W. Lueks and I. Goldberg. Sublinear scaling for multi-client

private information retrieval. Technical Report 19, University of Waterloo, 2014.

[51] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanovic, J. Kubiatowicz, and D. Song. Phantom: Practical oblivious computation in a secure processor. In *ACM SIGSAC Conference on Computer & Communications Security*, pages 311–324, 2013.

[52] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish. Depot: Cloud storage with minimal trust. *Transactions on Computer Systems (TOCS)*, 29(4):12, 2011.

[53] T. Mayberry, E.-O. Blass, and A. H. Chan. PIRMAP: Efficient private information retrieval for MapReduce. In *Financial Cryptography and Data Security (FC)*, pages 371–385. Springer, 2013.

[54] T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining ORAM and PIR. 2014.

[55] C. McSherry and C. Cohn. Digital books and your rights: A checklist for readers. Electronic Frontier Foundation, `https://www.eff.org/files/eff-digital-books_0.pdf`, Feb. 2010.

[56] C. A. Melchor, B. Crespin, P. Gaborit, V. Jolivet, and P. Rousseau. High-speed private information retrieval computation on GPU. In *International Conference on Emerging Security Information, Systems and Technologies (SECUREWARE)*, pages 263–272, 2008.

[57] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *ACM Symposium on the Theory of Computing (STOC)*, pages 245–254, 1999.

[58] A. Narayanan and V. Shmatikov. Robust de-anonymization of large sparse datasets. In *IEEE Symposium on Security and Privacy (S&P)*, 2008.

[59] A. Narayanan and V. Shmatikov. Myths and fallacies of “personally identifiable information”. *Communications of the ACM*, 53(6):24–26, June 2010.

[60] F. Olumofin and I. Goldberg. Preserving access privacy over large databases. Technical Report 33, University of Waterloo, 2010.

[61] F. Olumofin and I. Goldberg. Revisiting the computational practicality of private information retrieval. In *Financial Cryptography and Data Security (FC)*, pages 158–172. Springer, 2012.

[62] R. Ostrovsky and W. E. Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *Public Key Cryptography (PKC)*, pages 393–411. Springer, 2007.

[63] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*. Springer-Verlag, 1999.

[64] S. Papadopoulos, S. Bakiras, and D. Papadias. pCloud: A distributed system for practical PIR. *IEEE Transactions on Dependable and Secure Computing*, 9(1):115–127, 2012.

[65] V. Pappas, F. Krell, B. Vo, V. Kolesnikov, T. Malkin, S. G. Choi, W. George, A. Keromytis, and S. Bellovin. Blind seer: A scalable private dbms. In *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, SP ’14, pages 359–374, Washington, DC, USA, 2014. IEEE Computer Society.

[66] D. Rebollo-Monedero and J. Forné. Optimized query forgery for private information retrieval. *IEEE Transactions on Information Theory*, 56(9):4631–4642, 2010.

[67] B. Schneier. The eternal value of privacy. *Wired*, May 2006. `http://archive.wired.com/politics/security/commentary/securitymatters/2006/05/70886`.

[68] A. Shraer, C. Cachin, A. Cidon, I. Keidar, Y. Michalevsky, and D. Shaket. Venus: Verification for untrusted cloud storage. In *Cloud computing security workshop (CCSW)*, pages 19–30. ACM, 2010.

[69] R. Singel. Netflix spilled your *Brokeback Mountain* secret, lawsuit claims. *Wired*, Dec. 2009. `http://www.wired.com/images_blogs/threatlevel/2009/12/doe-v-netflix.pdf`.

[70] R. Sion and B. Carbunar. On the practicality of private information retrieval. In *Network and Distributed System Security Symposium NDSS*, Mar. 2007.

[71] S. W. Smith and D. Safford. Practical server privacy with secure coprocessors. *IBM Systems Journal*, 40(3):683–695, 2001.

[72] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. *arXiv preprint arXiv:1106.3652*, 2011.

[73] J. Summers, T. Brecht, D. Eager, and B. Wong. Methodologies for generating HTTP streaming video workloads to evaluate web server performance. In *International Systems and Storage Conference (SYSTOR)*, page 2, 2012.

[74] R. Thurlow. RPC: Remote procedure call protocol specification version 2. RFC 5531, Network Working Group, 2009.

[75] J. Trostle and A. Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *Information Security*, pages 114–128. Springer, 2011.

[76] S. Viswanathan and T. Imielinski. Pyramid broadcasting for video-on-demand service. In *Proc. Multimedia Computing and Networking (MMCN)*, 1995.

[77] S. Wang, D. Agrawal, and A. El Abbadi. Generalizing PIR for practical private retrieval of public data. In *Data and Applications Security and Privacy*, pages 1–16. Springer, 2010.

[78] P. Williams and R. Sion. Usable PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2008.

[79] S. Yekhanin. Private Information Retrieval. *Communications of the ACM*, 53(4):68–73, Apr. 2010.