Scalable and private media consumption with Popcorn

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

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

Abstract This paper describes the design, implementation, and evaluation of *Popcorn*, a media content delivery system that provably hides clients' media consumption. Popcorn relies on a powerful cryptographic primitive: private information retrieval (PIR). With novel refinements that leverage the properties of PIR protocols and of media streaming, we have developed 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 3.87× the per-request cost of a non-private system.

1 Introduction and motivation

This paper describes a Netflix-like media delivery system, Popcorn, that provably hides *what* is consumed by its users, at scale and at low (dollar) cost.

Popcorn is motivated by a fundamental tension in the ecosystem of online media consumption. In one camp, there are people who are deeply uncomfortable exposing their media diet; a particular worry is a centralized media server (e.g., Netflix), as it is a target for capture, whether by hacking or subpoena. Philosophically, privacy advocates observe that freedom requires the ability to consume privately [88]; practically, an entity with access to a person's consumption profile can infer the person's sexual orientation, political leanings, private cultural affiliations, etc. [74, 75, 90].[1] And although many people may in fact *want* to expose their consumption, to gain recommendations—there may 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 advances a new design point in the realm of private 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. YouTube's library, for instance, is large, continuously updated, freely distributed, and powered by advertising. Netflix's library is comparatively small, updated infrequently [5], subject to strict content protection, and powered by paid subscriptions. This paper explicitly targets Netflix-like systems, and adopts the following requirements:

1. *Hide requests comprehensively and provably.* "Comprehensive" means hiding consumption not only from a network eavesdropper [6, 42] but also from the content distributor. "Provable" means that we wish to avoid the risk [17] of heuristic solutions.
2. *Make it affordable even at scale.* Our system should dispense privacy at an attractive price point. 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 (concealing *who* creates/consumes content, such as Tor [38]), satisfy the above requirements. However, these solutions conflict with media delivery: a content provider would have to relinquish its control of content delivery to Tor nodes. Meanwhile, the aggregate bandwidth—and the latency and reliability—on a Tor network is unlikely to accommodate every Netflix user. Moreover, a service like Netflix would have to rely on the altruism of Tor.

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 [28, 43, 64, 78, 102] allow clients (content consumers) to request content from servers (content distributors) without the servers being able to infer which items the clients requested.

These protocols are powerful, but applying them requires overcoming several challenges (§3): the linear overhead of PIR (responding to a query requires the server to compute over its entire library; otherwise the server would know what the client was *not* interested in); the strict deadlines of media delivery; variable object sizes (PIR assumes all objects are the same size); and a tension between PIR protocol choice and content protection (one type of PIR, called CPIR [64], needs only one server, but the overhead is high; another, called ITPIR [28], involves lightweight

[1] To be clear, we are not challenging the trustworthiness of commercial media services. The issue is that collecting the information in the first place creates the risk of exposure.

operations but demands non-colluding servers and hence separate administrative domains). As we discuss in Section 7, there is a large body of work that attempts to address some of these issues [11, 12, 15, 24, 30, 32, 34–36, 41, 47, 48, 52, 53, 55, 57, 68, 70, 72, 80, 93, 97, 99, 101, 103]. These many intriguing developments notwithstanding, prior implementations applicable to media delivery systems at the scale that we target levy prohibitive demands in terms of I/O and CPU resources.

Popcorn eases these demands substantially. It provably hides media consumption, scales to the size of Netflix, and respects current controls on media dissemination with a significantly reduced resource overhead, which translates into a manageable dollar cost. To do so, Popcorn cherry-picks techniques from the literature on PIR and media on demand, and works through the systems ramifications of tailoring them to the context at hand.

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 stored at a single server and retrieved using the heavier-weight CPIR. Second, Popcorn batches requests from the large numbers of concurrent users streaming content at any given time to amortize the costs of PIR. It forms large batches without introducing playback delays or interruptions by leveraging streaming. Third, Popcorn exploits the ability to encode a media object in multiple ways (e.g., by changing its bitrate) to meet the fixed-size requirement of PIR.

We experimentally evaluate Popcorn for a Netflix-like workload (10,000 concurrent clients, each streaming at 4 Mbps from a library of 8192 movies with an average length of 90 minutes). Popcorn's overheads are high when compared to a non-private baseline: for each request, Popcorn consumes 1080× more computational resources, ≈14× more I/O bandwidth, and incurs 2× in network transfers. However, since CPU is cheap and Popcorn is engineered to conserve the more expensive resources (I/O and network), the overheads when translated to dollars are manageable: Popcorn's per-request cost, in terms of dollars, is 3.87× that of the baseline.

Though promising, Popcorn has several limitations (§8). It requires non-colluding servers. Its overheads grow with the library size; this precludes scaling to media libraries that have more than a few tens of thousands of media files. (YouTube, for example, has millions [27].) It does not support forward seeking. In addition, the current prototype lacks features that would be required in a full-fledged deployment: online library updates, deployment via CDNs, elasticity, adaptive streaming, royalty payments, and advertising and recommendations. Some of these have natural solutions; others require further research.

2 Setting and 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 studio), delegates to the distributor (e.g., an online streaming service like Netflix) the tasks of disseminating content and charging consumers.

We model the content kept by the distributor as a collection L of n objects; we call L the *library*. 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 providing the distributor with the corresponding integer.

Threat model We consider three types of threats: to consumer privacy, to content protection, and to content integrity. The first threat describes an attacker (the content distributor or a network eavesdropper) trying to infer what object the consumer is accessing. We assume that the attacker has full access to the network and to the content of the consumer's requests but do not consider side-channel attacks, such as using knowledge of where a customer pauses playback, or of his/her concurrent web browsing activity. The second threat considers a consumer that is trying to copy and redistribute content beyond what is allowed by the distributor. Today's media delivery systems rely on the existence of a client-side trusted environment in which code executes correctly and content cannot be further distributed once consumed [14, 39]. We make the same assumption and hence do not consider this issue further in Popcorn. The final threat focuses on a distributor serving incorrect content to the consumer. We treat content integrity as an orthogonal problem, which undermines correctness (§2.2) but not privacy. The literature offers standard solutions to guarantee content integrity (content hashing, etc.).

2.2 Private Information Retrieval (PIR)

The high-level goal of PIR protocols aligns with that of Popcorn: they 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 (see Figure 1). To privately retrieve object $O_b = L[b]$, the client invokes Query(b) to produce 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

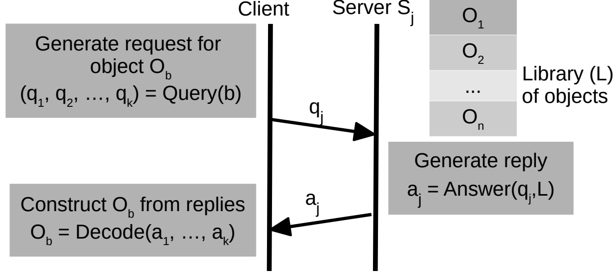

Figure 1—The structure of a PIR protocol.

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.

We want three properties from a PIR protocol:

- ***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.
- ***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 two such PIR protocols.

2.3 Computational PIR (CPIR) protocols

CPIR protocols [64] require only a single, computationally bound server ($k = 1$). They are commonly constructed using additively (not fully [44]) homomorphic public key cryptosystems. 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, pk is a public key, sk is a secret key, Dec is the decryption algorithm, and Enc is the (randomized) encryption algorithm. Examples of cryptosystems used in CPIR are the Paillier [79] and the lattice-based Ring-LWE [19] cryptosystems.

Figure 2 depicts a CPIR protocol due to Ostrovsky and Skeith [78]. It meets the three properties (§2.2), as we informally argue:

- ***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 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 S sees is $q = (pk, c_1, \ldots, c_n)$. If 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 properties of the underlying encryption scheme.
- ***Communication efficiency.*** The length of the server's reply is $(\ell/y) \cdot |c|$ bits, where ℓ/y is the number of ciphertexts in the reply and $|c|$ is the size (in bits) of a ciphertext. $(\ell/y) \cdot |c|$ is comparable to ℓ, the size of object O_b, if the expansion ratio, $|c|/y$, of the underlying additively homomorphic cryptosystem is small.[2] The client's request contains n ciphertexts and is thus $|c| \cdot n$ bits. When $\ell \gg n$ (as will be the case in our context) and $|c|$ is a small constant (e.g., 2048 in many Paillier implementations), $|c| \cdot n$ is much smaller than ℓ.

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 y-bit integers:
  // $L \in (\{0,1\}^y)^{n \times (\ell/y)}$
  for $j = 1$ to ℓ/y **do**
    $r_j \leftarrow \prod_{i=1}^{n} {c_i}^{L_{i,j}}$
  return $a = (r_1, \ldots, r_{\ell/y})$

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

Figure 2—A computational PIR (CPIR) protocol based on an additively homomorphic cryptosystem (Gen, Enc, Dec) and due to Ostrovsky and Skeith [78]. (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.

2.4 Information-theoretic PIR (ITPIR) protocols

ITPIR protocols [28] use more than one server ($k > 1$), and assume that they do not collude; thus, in practice, the servers must belong to different administrative domains.

Figure 3 shows the CGKS [28] ITPIR protocol. It meets the three properties of PIR (§2.2):

- ***Correctness.*** The output of Decode is $\bigoplus_{j=1}^{k} a_j$, which equals $\bigoplus_{j=1}^{k} (q_j \cdot L)$. By properties of the field $\mathbb{F}_2$ (that addition is XOR and that multiplication distributes over addition), $\bigoplus_{j=1}^{k} (q_j \cdot L) = (\bigoplus_{j=1}^{k} q_j) \cdot L = e_b \cdot L = L[b]$.
- ***Privacy.*** Each server in $S_1, \cdots, S_{k-1}$ sees a randomly generated query vector, and therefore each server (and all of them combined) cannot learn any information about b. Server S_k sees q_k, which is constructed by

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

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
// q is a row vector, L a logical matrix: $L \in \{0,1\}^{n \cdot \ell}$
return $q \cdot L$ // product over the two-element field $\mathbb{F}_2$

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

Figure 3—The ITPIR protocol of CGKS [28]. n is the number of objects in library L, and ℓ is the length of each object. k is the total number of servers.

XORing unit vector e_b with the one-time pad $q_1 \oplus \cdots \oplus q_{k-1}$. By the properties of one-time pads, S_k can learn information about e_b only by learning the one-time pad (or by colluding with all other servers).

- ***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 applying PIR

Though PIR is promising, there are a number of challenges in applying it to large-scale media consumption:

- *Resources.* Under PIR, the I/O and CPU resources required to serve a single request are proportional to the size of the library. One might guess that batching requests to amortize some of this overhead would help, but this technique is in tension with the next issue.
- *Strict deadlines.* Media delivery has stringent latency requirements: initial delay must be small, and the delivery must obey real-time constraints.
- *Variable object sizes.* Object sizes vary as a function of encoding or playback time. However, PIR assumes objects of identical size.
- *Content protection in ITPIR vs. CPIR.* Content creators may be loath to disseminate the content beyond its original distribution channel. Yet ITPIR requires multiple non-colluding servers, and hence multiple administrative domains, necessitating such dissemination. CPIR, on the other hand, requires only a single server; however, its computational cost is significantly higher.[3]

[3]The state of the art CPIR implementation is XPIR, which is based on the Ring-LWE cryptosystem. XPIR can process data at a rate of $\approx$5.5 Gbps per core [11], while the CGKS ITPIR implementation in Percy++ [47], based on cheaper XOR operations, can achieve a processing rate of $\approx$61 Gbps per core on comparable hardware (§6).

	I/O	CPU	content prot. (ITPIR)	resists collusion	object sizes	pricing, reco
XPIR [11]		◐	●	●		
RAID-PIR [32]					◐	
Percy++ [47]		◐	◐	◐	◐	●
Popcorn	●	◐	●		◐	

Figure 4—Prior PIR-oriented works (rows) and which media-related challenges they address (columns), assuming two or fewer servers. ● means that the work addresses the challenge; ◐ means that the work partially addresses the challenge.

- *Billing, access control, recommendations.* For business reasons, media services may need to support access control, pricing policies (tiers, etc.), targeted advertising, and recommendations. Yet, the hiding in PIR conflicts with all of this functionality.

Subsets of these challenges have been addressed before, as depicted in Figure 4. Popcorn principally aims at the resource consumption issue, via the architecture and design described in the next section.

4 Architecture and design of Popcorn

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 two *secondary content distributors*, each in separate administrative domains. The primary content distributor maintains a *key server*. Each secondary content distributor maintains an *object server* that is distributed over multiple physical machines.

The key server delivers the per-object keys using CPIR; the object servers deliver encrypted objects using ITPIR (§4.1). The distinction between key and object servers maps to a similar distinction in today's DRM implementations [1, 2, 82], where clients contact two separate servers, one for encrypted video and one for decryption keys.

Each media object is split into *segments*; a segment is a contiguous piece of a media object containing, for example, a few seconds or minutes of a video. Segment sizes vary (§4.3). Each object is presumed to have the same decomposition into segments (Section 4.4 revisits this assumption). As depicted in Figure 6, the library is partitioned into *columns*; a column is a union of corresponding segments, across all objects. Therefore, a column's size is n times larger than any segment it contains.

Each column is stored and served by two independent ITPIR *instances* (one for each object server); different instances use separate physical machines. Columns are further sub-divided into *slices*, which are the work units assigned to physical machines. A slice is 1 MB "wide" and n items "high"; we sometimes refer to 1 MB as a

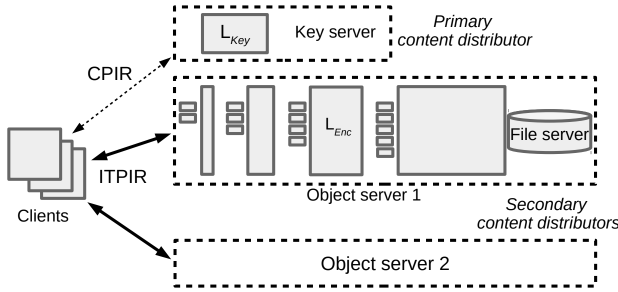

Figure 5—Architecture of Popcorn. Each object server stores all of the columns in the library (Fig. 6), and is distributed over multiple physical machines.

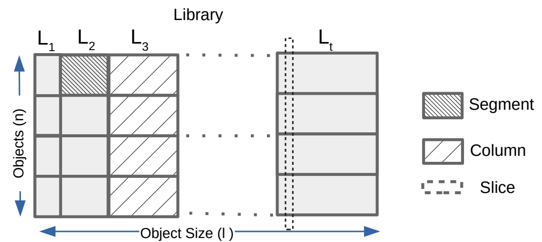

Figure 6—Popcorn terminology. Each column is stored by two ITPIR instances (one from each object server). Columns are divided into slices, which are assigned to physical machines.

chunk. Each machine is responsible for one or more slices.

To retrieve an object, the client fetches the object's decryption key from the key server and the encrypted object from the object servers; the latter proceeds in two overlapping phases. In the first phase, the client sends, in parallel, a query vector to all machines in both object servers.[4] On receiving a request, a machine adds the query vector to a request queue. Each machine services its queue by looping over its slices, computing chunk-sized ITPIR replies for every pending request. Each machine pushes the resulting chunks to a single (distributed) file server (Fig. 5), which retains the chunks until they are requested by clients. In the second phase, the client downloads these ITPIR-encoded chunks at the appropriate playback times, and applies Decode (Fig. 3). This phase overlaps with the server-side generation of replies.

4.1 Composing ITPIR and CPIR

As stated earlier, 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).

As an alternative to CPIR, the key server could use a Symmetric PIR (SPIR) scheme or a 1-out-of-n Oblivious Transfer (OT) protocol. Section 7 discusses these alternatives.

4.2 Batching

Popcorn uses the CGKS ITPIR scheme described in Section 2.4, as the computational overhead is low (by the standards of PIR), owing to inexpensive operations (XORs). Still, because ITPIR queries are dense—on average, half of the entries are set to 1 (Fig. 3)—responding to a query requires the machine serving a slice to read from storage $n/2$ chunks on average and then XOR them. This taxes I/O bandwidth, memory bandwidth, and CPU cycles.

To reduce costs, Popcorn's machines process queries in *batches*: they combine multiple queries, and perform a single sequential I/O pass over a slice. This exploits sequential transfer bandwidth and amortizes the cost of the pass over all of the queries in the batch.

Batching also reduces *computational* (not just I/O) overhead. Specifically, 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). Previous work [18, 68] has applied this observation to incorporate sub-cubic algorithms [29, 56] (resulting in fewer total operations than would be done without batching). Popcorn, by contrast, chooses block matrix multiplication [65], which, though it does not affect the total number of operations, leverages cache locality. One can view the resulting access pattern as batching at the CPU-memory interface.

4.3 Specializing batching for media delivery

Given the considerations in the previous subsection, Popcorn has an interest in increasing batch sizes (at least up to a point).[5] However, there is a tension between large batch sizes, which seems to require synchronizing clients, and meeting the deadlines imposed by real-time media delivery. This subsection describes how Popcorn addresses the tension.

Popcorn's high-level solution is as follows. To begin with, each ITPIR instance loops over its assigned column (§4) continuously. The first column is "narrow", and—since the client can begin playback after decoding the response for this column—the initial delay is short. But the column widths increase quickly; as a result, there are not many columns, which implies that some of them are wide. The crucial intuition is that *wide columns im-*

[4] This reflection of the query vector adds network overhead (a few hundred KB in the worst case). This cost could be dramatically reduced by installing a single reflector node in each object server.

[5] Above a certain batch size, there is no advantage: I/O is no longer a bottleneck, and the CPU benefits of using matrix multiplication stop increasing. However, there is also no disadvantage, so for simplicity, Popcorn does not bound batch sizes.

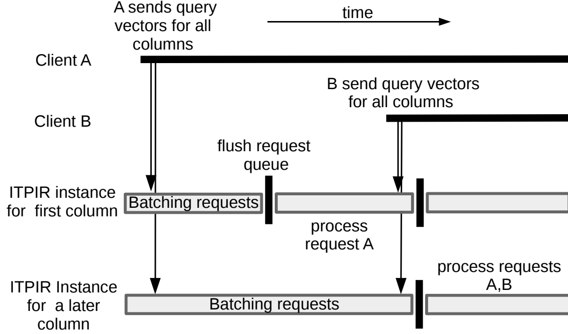

Figure 7—Batching at an object server in Popcorn. Requests for the initial segment from two clients A,B are in separate batches as the processing cycle for the initial column is short. The requests for a later segment (sent alongside the requests for the first segment) can be batched. This arrangement is inspired by Pyramid Broadcasting [98].

ply good batching opportunities: a wide column implies a long loop interval for the instance, and a batch comprises all requests that arrived during the previous (long) interval.

Figure 7 depicts the high-level arrangement. It is inspired by Pyramid Broadcasting (PB) [98], wherein increasingly-sized pieces of a media object are served on separate broadcast channels. However, the details of our setup are different: a Popcorn server's work depends on the number of clients (unlike in broadcasting), Popcorn relies on server-side buffering (PB relies on client-side buffering), and Popcorn is concerned with provisioning machines (PB concentrates on allocating bandwidth). These and other differences lead us to a solution that owes a debt to PB but is specific to our context.

Details. We start with two simplifying assumptions, which we revisit later: that a single instance is handled by a single machine, and that there is no network delay or loss. Define an *instance processing cycle* as the duration of one iteration of an instance's loop. Within this cycle, an instance traverses each slice in turn, performing Answer for all queries that arrived during the prior cycle.

We want all clients to experience smooth playback. To this end, suppose that we are willing to impose startup delay d. Suppose further that $T_1 \leq d-\epsilon$, where T_1 denotes the processing cycle for the first instance, and ϵ is the time for the instance to handle a single slice. Likewise, define T_i as the processing cycle for the ith instance ($i > 1$), and suppose that for all such instances, $T_i \leq d - \epsilon + \sum_{j=1}^{i-1} t_j$, where t_j is the playback time of segment j.

Under these conditions, we claim that any client, *regardless of when it joins*, experiences smooth playback. Why? Considering only instance 1, the worst case is that a client initiates consumption just after instance 1 began its processing cycle. This client cannot download until the processing cycle has expired (which takes time T_1) and the first slice was handled (time ϵ). As a result, the delay is $T_1 + \epsilon$, which, under the conditions above, means that the delay is less than d, as required. Next, consider the moment that playback begins: the client has t_1 further time units before it needs the second segment. Generalizing, in the worst case for instance i (the client's initial request arrived just as a processing cycle was beginning), as long as T_i finishes in $d - \epsilon + \sum_{j=1}^{i-1} t_j$ time units (which the conditions above guarantee), then the first slice of the ith instance will be ready, and playback will be smooth.

But how should the $\{t_i\}$ be set? Recalling the aforementioned intuition, Popcorn arranges for the segment widths to increase. An analysis follows; we are seeking the maximum t_i for each instance i.

Let μ be the playback rate, P_i be the processing throughput (the rate at which the CPU XORs data) available to the ith instance, R_i be the I/O bandwidth available to the instance, and b_i be the batch size (the number of requests accumulated in a cycle of time T_i). To upper-bound t_i, we match load to capacity, for both I/O and CPU. For I/O transfers, the column's data (the segment size of $t_i \cdot \mu$ times the number of items, n) is upper-bounded by the amount of data that the instance can read in one cycle: $t_i \cdot \mu \cdot n \leq T_i \cdot R_i$. For CPU transfers, the picture is similar, except that the total work scales with b_i, the number of clients being served: $t_i \cdot \mu \cdot n \cdot b_i \leq T_i \cdot P_i$. These inequalities lead to:

$$t_i \leq T_i \cdot \left(\frac{\min\{R_i,\, P_i/b_i\}}{\mu \cdot n} \right).$$

Assume that for all i, $\min\{R_i,\, P_i/b_i\} \geq \mu \cdot n$ (we will arrange for this in "Provisioning," below). Then, the foregoing bounds (on the $\{T_i\}$ and on load) imply that for all i, we can set:

$$t_i = T_i = 2^{i-1} \cdot (d - \epsilon).$$

(This is derived in Appendix A.) Observe that the $\{t_i\}$ increase exponentially, as desired. In particular, approximately half of the file is covered by the final segment.

Provisioning is driven by the earlier assumption that $\min\{R_i,\, P_i/b_i\} \geq \mu \cdot n$ for all i. To meet the requirements on R_i and P_i, Popcorn uses multiple machines per instance and aggregates their resources, by striping slices over them. If r_i is the per-machine I/O bandwidth for the machines used for the ith instance, then the I/O for instance i can be handled with $R_i/r_i = \mu \cdot n/r_i$ machines. For the "XOR work", P_i increases with i because the batch size b_i varies; specifically, if λ is the overall rate at which clients initiate requests for movies, then $b_i = \lambda T_i$. Moreover, the per-machine "XOR bandwidth" for the ith instance, $p_i(\cdot)$, is modeled as a function of the batch size because a bigger batch size implies better cache locality in block matrix multiplication (§4.2), which increases "XOR bandwidth". Thus, the XOR work for instance i can be handled with $P_i/p_i(b_i) = \mu \cdot n \cdot b_i/p_i(b_i)$ machines.

To account for the striping, the earlier analysis of startup delay, smooth playback, etc. must be modified: if resources from k_i machines are aggregated for the ith instance, then each machine takes $\epsilon \cdot k_i$ time instead of ϵ to handle a slice. As a result, the inequality $T_i \leq d - \epsilon + \sum_{j=1}^{i-1} t_j$ becomes $T_i \leq d - \epsilon \cdot k_i + \sum_{j=1}^{i-1} t_j$, and the $\{T_i\}, \{t_i\}$ are computed accordingly.[6] Then, the total number of machines, across all I instances, is: $\mu \cdot n \cdot \sum_{i=1}^{I} \max\{1/r_i,\ \lambda T_i/p_i(\lambda T_i)\}$. Notice that if the first term controls the max, then the given instance is bottlenecked by I/O (and the CPU resource is sometimes idle), whereas if the second term controls, then the instance is bottlenecked by CPU work (and the I/O resource is sometimes idle). Later (§6.1), we will obtain estimates empirically for r_i and $p_i(\cdot)$.

Popcorn must also provision for the file server machines (§4). The file server requires that, for each instance, there is buffer space equal to the number of requests in service times the size of the reply. This equals $\sum_{i=1}^{I} b_i \cdot (t_i \cdot \mu)$. The file server also requires I/O bandwidth equal to the rate at which reply data is produced and consumed: $2 \cdot \sum_{i=1}^{I} b_i \cdot \mu$ (assuming $t_i = T_i$).

Finally, we have been assuming no burstiness or delay in the network. To account for network fluctuation, we must allow for clients to build up a playback buffer, of some time length b. To this end, T_i should be upper-bounded by $d - \epsilon \cdot k_i - b + \sum_{j=1}^{i-1} t_j$, and the $\{t_i\}$ computed accordingly.

Discussion. To understand the savings and amortization from Popcorn's batching, consider a naive batching scheme, in which time is divided into *epochs* of length T_{epoch}. Define all clients who initiate a request (for the first chunk of a media file) in an epoch as a single *cohort*. Then, the entire cohort "moves through the slices" together. We can calculate the provisioning requirements for this setup: each cohort needs enough machines to satisfy its requirements of (a) $\mu \cdot n$ I/O bandwidth, and (b) $\mu \cdot n \cdot \lambda \cdot T_{\text{epoch}}$ "XOR bandwidth" (here $\lambda \cdot T_{\text{epoch}}$ is the cohort's batch size). If $H = T/T_{\text{epoch}}$ is the total number of cohorts (where T is the total playback time), then the total number of machines is $\mu \cdot n \cdot \sum_{i=1}^{H} \max\{1/r,\ \lambda \cdot T_{\text{epoch}}/p(\lambda \cdot T_{\text{epoch}})\}$, where r is the per-machine I/O bandwidth, and $p(\lambda \cdot T_{\text{epoch}})$ is per-machine "XOR bandwidth" for a batch size of $\lambda \cdot T_{\text{epoch}}$. Here, T_{epoch} must be upper-bounded by $d - \epsilon \cdot k - b$ to meet the startup delay requirements, where k is the number of machines for a cohort.

To compare the cohort batching scheme to Popcorn, we make a simplifying and optimistic assumption: that both the schemes have machines such that the two terms of the max are equal, and there are no idle resources. (Later, in Sections 6.2 and 6.4, we will revisit this assumption.) Then, the total I/O bandwidth required in the cohort scheme is $H \cdot \mu \cdot n$, which is considerably larger than is required in Popcorn: compare to $I \cdot \mu \cdot n$, where $I \ll H$.

To compare the computational resources, consider the number of machines required for the two schemes: the cohort scheme requires $\mu \cdot n \cdot \lambda \cdot T/p(\lambda \cdot T_{\text{epoch}})$, which can be written as $\mu \cdot n \cdot \lambda \cdot \sum_{i=1}^{I} T_i/p(\lambda \cdot T_{\text{epoch}})$; Popcorn requires $\mu \cdot n \cdot \lambda \cdot \sum_{i=1}^{I} T_i/p_i(\lambda \cdot T_i)$. An unequivocal comparison is not possible. However, if we assume that $p(\cdot) = p_i(\cdot)$ for all i, then Popcorn has lower computational demands. This is because $T_i \approx 2^{i-1} \cdot T_{\text{epoch}}$ (by our earlier analysis) and because $p(\cdot)$ is monotonically increasing. In essence, Popcorn has larger batches, so (holding machine type configuration constant) the locality benefit is more pronounced (§4.2), lowering its computational requirements relative to the naive batching scheme.

4.4 Handling variable-sized objects

The design has so far assumed objects to be of equal size. A naive solution is to pad all objects to the size of the longest one. However, this can be very inefficient; for example, in Netflix, the longest movie is approximately 6 hours, and the average movie length is 1.5 hours, so network transfers would increase by 4×.

Popcorn's solution combines padding and compression. It chooses a representative object O_{avg} from the library (for example, the object closest to the average media length) and makes all other objects O_{avg} in size: objects smaller than O_{avg} are padded. Longer objects, meanwhile, are compressed by reducing the bitrate.

Both padding and compression are potentially problematic. Padding could waste resources, but by using it in combination with compression, Popcorn ensures that objects will be padded only up to O_{avg}, limiting costs. For compression, we consider two cases. First, if the object is up to 30% bigger than O_{avg}, then the compression required to make it O_{avg} is small, and the consequent degradation in quality is likely to be tolerable. Indeed, several studies [62, 91] suggest that small variations in video bitrate have a limited impact on user satisfaction.

The second case is that the object is much larger than O_{avg}; a trade-off then arises between quality and on-demand consumption. On the one hand, Popcorn can compress these objects aggressively; however, doing so will result in significant quality degradation. On the other hand, Popcorn, like others [32, 52], can divide up these objects; however, the client would then have to download each division as if it were a separate movie, which means delaying consumption or downloading far ahead of time (if the separate divisions were downloaded all at once, then an attacker could guess that a longer object is being consumed).

[6] The computation must resolve a circular dependency as T_i is expressed in terms of k_i, which itself depends on the segment size, with a bigger segment requiring more machines. We resolve this circularity by repeating the process of speculatively setting a k_i, calculating T_i, and then refining the speculated value of k_i using the obtained value of T_i.

Popcorn is affordable when it serves large media files to many concurrent clients.	§6.2
Popcorn's per-request dollar cost is 3.87× of a system without privacy for workloads with ≥10K concurrent clients.	§6.3
Popcorn integrates well with existing web technology. It can play DRM-encoded media within modern web browsers.	§6.5

Figure 8—Summary of main evaluation results.

The Netflix catalog [5] indicates that the majority of movies have a similar size: 85% of the objects are between 60 and 120 minutes, with the majority clustered around the average movie length of 92 minutes. Given this distribution, movies between 92 and 120 minutes require, on average, 10% compression, which does not degrade viewing quality prohibitively. Similarly, the padding for objects between 60 and 92 minutes will be small. The impact of objects at either extreme will be limited: 8% of the movies are below 60 minutes, and will require significant padding; 6% are between 120 and 150 minutes, making them candidates for aggressive compression (29% on average); 1% are over 150 minutes, making them candidates for splitting.

5 Implementation

Popcorn leverages existing PIR implementations: the key server uses the XPIR [11] implementation of the CPIR protocol of Figure 2, and can be instantiated using either the Ring-LWE homomorphic [19] or the Paillier [79] cryptosystems. It is a total of 4,500 lines of C++ code. For our object servers, we borrow the CGKS ITPIR implementation of Percy++ [47] (one of the fastest implementations for two-server ITPIR[7]), and modify it to support the techniques mentioned in Section 4. The resulting server-side code is 6,500 lines of C++. Finally, we implement two versions of the client-side library: one in C++ (2,500 lines), which we use for our experimental evaluation (§6.2), and one in JavaScript (500 lines), which we use to demonstrate compatibility with modern web browsers (§6.5).

6 Evaluation

Our evaluation answers the following questions:

1. When is Popcorn affordable?
2. What is the price of Popcorn's privacy guarantees?
3. Can we use Popcorn to watch a movie encoded using an existing DRM scheme on a modern web browser?

Figure 8 summarizes our evaluation results.

Method and setup. We compare Popcorn to two baseline systems: *NoPriv* and *BaselinePIR*. NoPriv uses an Apache web server to serve chunks of objects and models modern media delivery systems that use HTTP caching at CDN edge servers [10]. BaselinePIR is a modified version of Percy++ [47] CGKS: the servers store the library L as slices and process ITPIR queries directed at individual slices. This is essentially Popcorn without the techniques in Section 4. For both PIR systems, we experiment with only one object server and multiply the measurements by two; we do this to reduce the financial cost of our experimental evaluation.

Our workload is modeled on existing media delivery services [95]: clients arrive according to a Poisson process (e.g., $C{=}10K$ clients arrive in $T{=}90$ minutes), and each client issues a request for an object. For NoPriv, our experiments request the same (average-sized) object every time; this gives NoPriv the maximum benefit of server-side caching. For Popcorn and BaselinePIR, the server's work is completely oblivious to the request distribution. (For concreteness, we choose a Zipfian distribution with $\theta = 0.8$.)

For the three systems, we measure server- and client-side resource usage in terms of CPU time (by instrumenting code with `clock()`), I/O transfers and storage (using `iostat`), and network transfers (via `/proc/net/dev`).

Our experimental testbed is a single availability zone

[7]Another choice would have been the CGKS implementation from RAID-PIR [32] (§7).

	type	vCPUs	RAM (GB)	SSDs (# × GB)	cost/hr
c3.8xl	1	32	60	2 × 320	$0.6281
i2.4xl	2	16	122	4 × 800	$0.8451
i2.8xl	3	32	244	8 × 800	$1.6902

Figure 9—Hourly cost of reserved Amazon EC2 machines, for the types of machines used in our experiments. Machines starting with "c" are compute-optimized; those starting with "i" are I/O-optimized.

	Throughput (Gbps)
On c3.8xl	
Sequential read	6.4
Random mixed rw	2.1
block matrix multiplication	488–4968
On i2.4xl	
Sequential read	12.6
Random mixed rw	8.0
block matrix multiplication	488–2512
On i2.8xl	
Sequential read	23.3
Random mixed rw	16.0
block matrix multiplication	432–4608

Figure 10—Throughput of basic operations in Popcorn—reading a column slice (§4.3), reading and writing 1 MB sized chunks, and computing block matrix multiplication on a slice (§4.2)—on machines listed in Figure 9. Throughput of block matrix multiplication depends on the size of the query matrix (§4.2, §4.3), so we present a range; the smallest value in the range corresponds to performance with a matrix consisting of a single query vector, and the largest corresponds to throughput with a matrix that contains 4,096 query vectors.

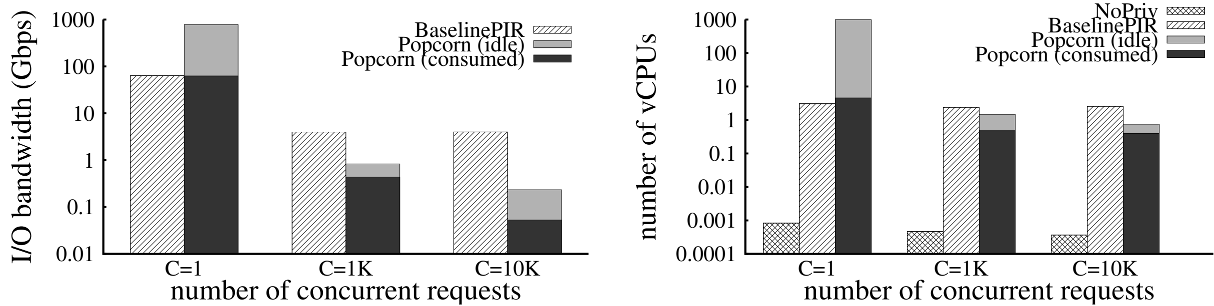

Figure 11—Per-request server-side resource use of NoPriv, BaselinePIR, and Popcorn, versus number of concurrent requests C. Depending on whether I/O or CPU is the bottleneck (§4.3), the other resource idles. Both idle and consumed resources are depicted for Popcorn; For the baselines, only the resources consumed are reported. The y-axis is log-scaled. I/O bandwidth for NoPriv is not depicted as it is always zero (see text).

within Amazon's EC2; Figure 9 lists the machine types that we choose for our experiments.

6.1 Provisioning resources using microbenchmarks

Popcorn. Determining a machine allocation for Popcorn is done in two steps: (1) benchmarking the basic operations in Popcorn (reading a slice of a column, block matrix multiplication, and reading and writing chunks of ITPIR replies), and (2) combining these results with the provisioning analysis in §4.3.

As an example, consider provisioning the first ITPIR instance of a Popcorn object server for a Netflix-like workload: C=10,000 clients streaming from a library of n=8,192 media files each with an average playing time of T=90 minutes at playback rate μ=4 Mbps, and an initial delay of d=15 seconds.[8] From the analysis in Section 4.3, this instance must have a processing cycle $T_1 \leq d - \epsilon \cdot k_1$. For our example, ϵ=2 (this is the time to process or consume a 1 MB chunk at μ=4 Mbps), and we speculatively set k_1=3 (this corresponds to three i2.4xl machines), which gives $T_1 \leq 15-2\cdot3 = 9$ seconds. Thus, the instance is given a segment of t_1=T_1=9 seconds and has a batch size of $b_1 = (C/T) \cdot T_1 = 17$. Furthermore, it requires storage capacity of $n \cdot t_1 \cdot \mu = 36$ GB, read bandwidth $R_1 = n \cdot \mu = 32$ Gbps, and processing throughput $P_1 = b_1 \cdot n \cdot \mu = 544$ Gbps.

Our microbenchmarks (Figure 10) suggest that the requirements on R_1 and P_1 can be met by provisioning three machines of type i2.4xl. Had we required a different number of machines than three, then, as described in Section 4.3, we would have had to adjust k_1 and repeat the provisioning process described above.

BaselinePIR. A natural choice is to store the entire library using the fewest number of machines: we, for example, vertically stripe the Netflix-like workload's ≈21 TB library across four i2.8xl machines. We measure the number of requests that can be serviced by this setup, along with each request's resource consumption. We use this data to extrapolate BaselinePIR's resource consumption for a workload that involves a larger number of requests (e.g., to support 2× concurrent clients, we multiply resource costs by two). We do this to further reduce the financial cost of our experimental evaluation.

[8]We think that 15 seconds of delay before playing a long video is tolerable. During this time the server could display a generic advertisement or public service announcement (existing services commonly display 15 or 30 second advertisements [9]).

6.2 Per-request overheads of Popcorn

We first consider the overheads of the object servers; the overheads of the key server are described toward the end of this subsection.

To understand when Popcorn is affordable, we run three sets of experiments in which we vary: the number of concurrent requests (C), the number of objects (n), and the playing time of objects (T). We find that Popcorn incurs modest costs when the library size is modest (≈8K media files), object sizes are large (≈90 minutes), and there are many concurrent clients (≥10,000). Fortunately, these parameter settings are in line with the workloads of Netflix-like systems (§8).

Before proceeding, we note that Popcorn's provisioning method can leave resources idle (§4.3), so we report both the consumed and provisioned resources. We focus on the consumed resources in this subsection and account for the idle resources in the next subsection.

Overhead versus number of concurrent requests. We run Popcorn and its baselines with C=$\{1, 1\text{K}, 10\text{K}\}$ while keeping n=8,192, T=90 min, μ=4 Mbps, and d=15 seconds. Figure 11 summarizes the per-request server-side resource costs.

I/O overheads. When C=1, Popcorn's consumed I/O bandwidth matches that of BaselinePIR as there is no opportunity to amortize costs by batching requests. However, as the request rate increases, Popcorn amortizes its I/O transfers via batching (§4.3): the per-request amortized

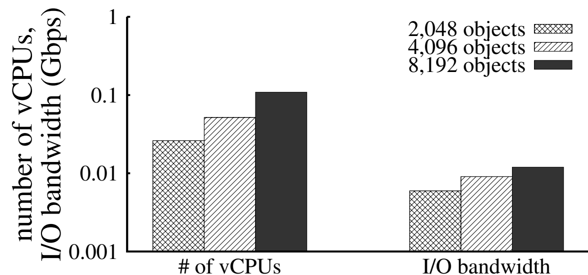

Figure 12—Per-request consumed resources in Popcorn versus the number of objects. The y-axis is log scaled.

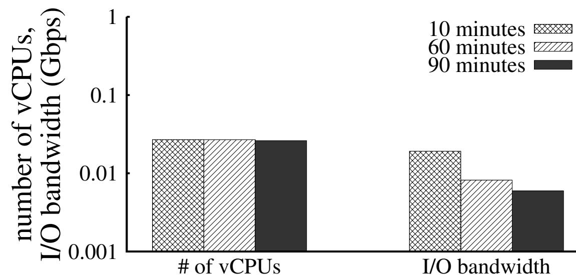

Figure 13—Per-request consumed resources in Popcorn versus the length of objects. The y-axis is log-scaled.

I/O bandwidth decreases from ≈63 Gbps (for C=1) to 53 Mbps (for C=10K), a reduction of 1190×.

Surprisingly, BaselinePIR's per-request I/O bandwidth also reduces (by 16×) when there are concurrent requests; this owes to requests hitting the file system cache. There are no I/O transfers in NoPriv as all requests hit the same (cached) object.

CPU overheads. For a single request, Popcorn consumes 50% more CPU than BaselinePIR, as the overhead of parallelizing block matrix multiplication (over multiple cores) in Popcorn (§4.2) is charged to a single request. As the number of concurrent requests increases, Popcorn's CPU overheads decrease; the per-request CPU consumption decreases by ≈11× when the number of concurrent requests increases from 1 to 10,000. We think that this is at least partly due to the increase in cache locality from block matrix multiplication over bigger batch sizes.[9] Furthermore, the 36 minutes of per-request CPU time for C=10K matches the performance of the matrix multiplication microbenchmark (42 TB of data processed in 36 minutes gives a throughput of 159 Gbps for a single CPU, which is consistent with the throughputs reported in Figure 10).

However, Popcorn's per-request CPU consumption is much higher than NoPriv (1080× for C=10K): the Apache web server in NoPriv serves 1 MB chunks of an object, which requires almost no server-side processing, while Popcorn must XOR n objects on average to serve a single object.

Network and storage overheads (not depicted in the figures). Both BaselinePIR and Popcorn incur a two-fold network overhead over NoPriv: there are two servers in Popcorn and BaselinePIR from which a client downloads content. With respect to storage, each instance of an object server in Popcorn needs buffer space equal to its segment size times its batch size (§4.3). This buffer space across all instances equals ≈15.4 TB, or ≈1.6 GB per concurrent request, which is 0.6× the size of an object.

Overhead versus number of objects. To understand how Popcorn's resource costs vary as a function of library size, we experiment with $n=\{2048, 4096, 8192\}$ while keeping the other parameters fixed (C=10K, T=90 min, μ=1 Mbps,[10] and d=15 seconds). Figure 12 depicts the results. As expected, Popcorn's per-request CPU and I/O bandwidth consumption, even though amortized, is proportional to n. Network downloads and server-side storage overheads (not depicted) do not change with the number of objects.

Overhead versus playing time of objects. To understand how Popcorn's overheads vary as a function of the playing time of objects, we experiment with $T=\{10, 60, 90\}$ minutes while keeping the other parameters fixed (n=2,048, μ=1Mbps, d=15 seconds, and C=10K). Figure 13 shows the results. We find that the number of CPUs consumed per-request (for the duration of the request) does not change for the three configurations, while the consumed I/O bandwidth decreases with the length of the object. Popcorn's efficiency (in terms of the consumed I/O bandwidth) thus improves for bigger objects. However, we note that idle I/O bandwidth (not depicted in the figure) likewise increases for bigger objects (§4.3).

Overheads of the key server. Recall that Popcorn uses XPIR [11] as its CPIR implementation (§5). Since XPIR does not batch requests, the per-request overheads of the key server depend only on the number of keys. We use a single machine of type c3.8xl for the key server. We find that for 8,192 keys, the server-side CPU time to privately retrieve a key is 3 seconds; there are zero I/O transfers because the library is only 128 KB, which fits in memory. Thus, as expected, the key server is not a performance bottleneck for Popcorn. We further note that the end-to-end time to retrieve a key is much less than the startup delay of d =15 seconds.

[9] In a separate experiment, we measured the percentage of cache misses for block matrix multiplication (§4.2) using CPU performance counters, and found that it reduces from 48% for a query matrix with a single request to less than 2% for a query matrix with 2^{10} requests.

[10] To reduce the financial cost of EC2 experiments, this and subsequent experiments set μ=1 Mbps instead of 4 Mbps. The change scales down the experiments; the qualitative results are unaffected.

	experimental configuration				per-request costs ($)		
	#reqs	#1	#2	#3	machine	network	total
NoPriv	10K	–	–	–	–	0.016	0.016
Popcorn	1	2	60	0	77.943	0.032	77.975
Popcorn	1K	17	50	4	0.09	0.032	0.122
Popcorn	10K	185	32	32	0.03	0.032	0.062

Figure 14—Estimated per-request dollar cost for NoPriv and Popcorn. #1, #2, and #3 refer to the type of AWS EC2 machines from Figure 9.

system	description
XPIR [11]	fastest CPIR implementation
XPIR+	XPIR with naive batching (§4.2)
BaselinePIR	XPIR composed with CGKS ITPIR (§4.1)
BaselinePIR++	BaselinePIR with naive batching (§4.2)
Popcorn	§4.1 + §4.2 + §4.3

Figure 15—Comparison points. "Naive batching" refers to an instantiation of batching, as described in Section 4.2, with the cohort batching scheme described in §4.3.

Client-side overheads. Compared to NoPriv, Popcorn's client consumes additional CPU and network bandwidth (since the Popcorn client has to generate PIR queries, download content from two object servers, and also decode them). For n=8192 objects, T=90 minutes, and μ=4 Mbps, we find that Popcorn's client consumes 10 CPU seconds (compared to NoPriv's 1.7 CPU seconds), and 25 MB of network upload bandwidth (compared to NoPriv's 11 MB).

6.3 Dollar-cost analysis

The previous subsection showed that Popcorn significantly reduces CPU and I/O consumption over BaselinePIR, especially when there are many concurrent requests and large objects. These improvements provide the foundation for achieving privacy at low cost, a cost which we now quantify for our target use case of Netflix.

Method. Our cost estimations use (1) the pricing model of Amazon EC2 (Figure 9) to estimate the per-request machine cost, and (2) the pricing model of CDNs ($0.006 per GB) [84] to compute per-request network cost. (We choose these pricing models because they are public—although in an actual deployment, a service could receive wholesale, bulk, or negotiated prices.) We use a Netflix-like workload in our calculations: n=8192 media files, T=90 minutes, μ=4 Mbps with varying number of concurrent clients.

Figure 14 summarizes our results. We find that Popcorn's per-request cost is within a small multiple of NoPriv for a workload with C=10K concurrent clients.

NoPriv. To give NoPriv the maximum benefit, we disregard its machine cost. The per-request cost is then determined solely by the network transfer cost, and is ≈ $0.016 (i.e., 90 minutes × 4 Mbps × $0.006/GB).

Popcorn. We use the method described in Sections 6.1 and 4.3 to provision EC2 machines. The total per-request cost is derived by combining (1) the per-request machine cost, computed by dividing total machine cost by the total number of requests, and (2) the per-request network cost. This method charges Popcorn for both consumed and idle resources (Figure 11).

For the Netflix-like library and C=10K, the per-request cost is $0.062 (the per-request machine cost is $0.03; the per-request network cost is $0.032). Popcorn thus increases dollar cost 3.87× over NoPriv, which we believe is in line with our initial affordability requirement (§1). Importantly, Popcorn's low cost is premised on many clients accessing the system concurrently: the per-request machine cost decreases with the number of concurrent clients. It is $78 for C=1 and $0.09 for C=1K.

BaselinePIR. Quantifying BaselinePIR's dollar cost is challenging: if we were to use the same method as we do for Popcorn, it would be unfair to BaselinePIR since that would charge for all the allocated resources, both consumed and idle, in an experiment. To address this, we create a per-resource pricing model and use it estimate the dollar cost of BaselinePIR. The next subsection provides details.

6.4 Further comparisons

In the previous subsection, we estimated the dollar costs of Popcorn and NoPriv. This subsection estimates the dollar costs of BaselinePIR and other natural baselines (e.g., XPIR [11]). The estimates for BaselinePIR are based on experimental data from Section 6.2; for other baselines, we use either the microbenchmarks in Figure 10 or reported performance (e.g., for XPIR) to extrapolate their end-to-end performance. Future work is to estimate their costs using end-to-end performance measurements.

We compare Popcorn to the following additional baseline systems: XPIR [11] (a CPIR scheme), XPIR+, and BaselinePIR++. Figure 15 summarizes these systems. XPIR+ is a hypothetical extension to XPIR; it uses the cohort batching scheme (§4.2) to reduce I/O costs (but does not implement block matrix multiplication). Similarly, BaselinePIR++ is a hypothetical extension to BaselinePIR (§6); it uses the same cohort batching scheme (which reduces I/O costs) and block matrix multiplication (which reduces CPU costs).

We compare these systems for a Netflix-like workload: a library with n=8,192 movies, T=90 minute average playback time, and μ=4 Mbps playback rate. We set startup delay d to 15 seconds, except for the systems using the cohort batching scheme, for which we vary d.

For the different systems, we report both the server-side resource consumption and the per-request estimated dollar cost. The resource consumption is determined experimen-

	# vCPUs	I/O bandwidth	Network transfers relative to NoPriv	Dollar cost relative to NoPriv
XPIR [11]	11.6	64 Gbps	5×	265×
XPIR+ (C=1)	11.6	64 Gbps	5×	265×
XPIR+ (C=1K)	11.6	26.6 Gbps	5×	118×
XPIR+ (C=1K, d=60)	11.6	5.96 Gbps	5×	37×
XPIR+ (C=1K, d=600)	11.6	0.58 Gbps	5×	16×
XPIR+ (C=10K)	11.6	2.66 Gbps	5×	24×
XPIR+ (C=10K, d=60)	11.6	0.59 Gbps	5×	16×
XPIR+ (C=10K, d=600)	11.6	0.058 Gbps	5×	13.5×
BaselinePIR (C=1)	3.1	64 Gbps	2×	256×
BaselinePIR (C=1K)	2.4	4 Gbps	2×	19×
BaselinePIR (C=10K)	2.5	4 Gbps	2×	19×
BaselinePIR++ (C=1)	2.1	64 Gbps	2×	255×
BaselinePIR++ (C=1K)	1.5	26.6 Gbps	2×	108×
BaselinePIR++ (C=1K, d=60)	0.84	5.96 Gbps	2×	26×
BaselinePIR++ (C=1K, d=600)	0.44	0.58 Gbps	2×	4.6×
BaselinePIR++ (C=10K)	0.55	2.66 Gbps	2×	12.8×
BaselinePIR++ (C=10K, d=60)	0.44	0.59 Gbps	2×	4.6×
BaselinePIR++ (C=10K, d=600)	0.41	0.058 Gbps	2×	2.5×
Popcorn (C=1)	4.6–992	63–781 Gbps	2×	253×–4873×
Popcorn (C=1K)	0.5–1.47	0.43–0.83 Gbps	2×	4×–7.6×
Popcorn (C=10K)	0.4–0.74	0.053–0.23 Gbps	2×	2.5×–3.87×

Figure 16—Per-request resource consumption and estimated dollar-cost of prior PIR implementations, their hypothetical extensions, and Popcorn, assuming two or fewer servers. Startup delay d for a system is set to 15 seconds unless specified otherwise.

tally or analytically, depending on whether we have end-to-end experimental data (§6.2). For both BaselinePIR and Popcorn, we take the resource consumption reported in Figure 11. (For Popcorn, we take both the consumed and provisioned resources.) For BaselinePIR++, we calculate CPU resource consumption using the empirical estimates of block matrix multiplication performance $p(\cdot)$ (Figure 10), and I/O bandwidth consumption using the expression $2 \cdot (n \cdot \mu)/b_{\text{cohort}}$, where the factor of two captures the two object servers, and b_{cohort} is the cohort's batch size and equals $\lambda \cdot (d - \epsilon)$ (§4.3). For XPIR+, we calculate CPU resource consumption using XPIR's reported performance and I/O bandwidth consumption using the expression $2 \cdot (n \cdot \mu)/b_{\text{cohort}}$ (it is twice an object server's because XPIR's preprocessed library is twice the size of the original [11]).

To estimate dollar cost for only the consumed resources, we derive a pricing model from Amazon EC2's machine cost model (Figure 9); the description and derivation are in Appendix B. The derived model charges CPU at \$0.0076/hour and I/O bandwidth at \$0.042/Gbps-hour. For network transfers, we use the CDN pricing model [84] that charges at \$0.006 per GB. We take these values and multiply them with each system's resource consumption (CPU, I/O bandwidth, and network transfers) to get per-request dollar cost.

Figure 16 depicts the comparison. We note the following:

- XPIR is a natural baseline that has been proposed in the literature. But it doesn't explicitly describe a batching scheme; if we use it without batching, the costs are high (265× NoPriv). Adding a naive batching scheme significantly reduces its costs (by a factor of approximately eleven when there are 10K concurrent clients and 15 seconds of startup delay).
- Using ITPIR for object delivery (by composing CPIR with ITPIR (§4.1)) reduces the costs further (by a factor of approximately two for C=10K, d=15). The benefits come from reduced CPU and network overhead. However, the downside is the dependence on the non-collusion assumption of ITPIR.
- Increasing the startup delay (and thus the batch size of the cohort) can further reduce costs. For example, increasing the delay from 15 to 60 seconds reduces costs by 2.78× (this corresponds to a reduction from 12.8× NoPriv to 4.6× NoPriv).
- BaselinePIR++'s cost matches that of Popcorn (when C=10K) but requires a startup delay that is 40× higher (d=10 minutes in BaselinePIR++ vs. 15 seconds for Popcorn).

6.5 Compatibility study of Popcorn

Popcorn's additional logic at the client raises compatibility concerns with existing client-side technology (web browsers) and DRM systems. We find that Popcorn can successfully play DRM encoded content within a modern web browser: we implemented a JavaScript Popcorn client and successfully watched short videos in WebM format [7] (protected using WebM Encryption–AES with 128-bit keys [8]). Our prototype works on Chrome (ver-

sion 45.0.2454) and follows the DRM standards. It makes use of the HTML5 video tag, and HTML5 video extensions: the decoded ITPIR content is passed into the Media Source Extension (MSE) which forwards media chunks to the video player; the decoded CPIR response is passed into the Encrypted Media Extension (EME) interface, which decrypts the DRM protected content.

7 Related Work

Alternatives to PIR for privacy. The cryptographic literature on privacy is vast; we describe some of the primitives below. For the most part, these works offer guarantees and properties that are different from PIR.

Obfuscation [17, 40, 85] protects clients' privacy by introducing dummy requests that serve as cloaking or cover traffic. Compared to PIR, this approach requires less processing at clients and servers but 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 downloading the entire library.

Rather than concealing the content being consumed, *anonymity* hides the identity of the consumer [38, 66]. We see anonymity as complementary: 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 [74].

Oblivious RAM (ORAM) [49, 69, 71, 94] algorithms allow a client to conceal its access patterns from a storage server. Similarly, searchable symmetric encryption (SSE) schemes (see [25, 26] for a survey of this area) are yet another solution for private data retrieval from a remote database. However, they are geared to a different setup: a client outsourcing its encrypted data to a server.

Nevertheless, recent works [59, 81] enhance the above setup: they let clients privately retrieve data from a remote database owned by a *different* entity. Unlike PIR, these protocols allow for a controlled amount of leakage in the form of data-access and query patterns. Unlike us, they assume that the server does not collude with clients (e.g., in Popcorn the server can pretend to be a new customer of the streaming service). If the server can collude with a client, it can issue queries for each media file in the system, monitor access patterns, and decode all other clients' queries.

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

One response is to distribute the work, either by moving it to the cloud or dividing it among clients [34, 70, 80]. This reduces latency via parallelization but not the total computational burden.

Another response is to reduce the computational load of CPIR by either using GPUs [30, 72], or cheaper cryptographic operations [11, 12, 41, 97, 103]. Taken together, these works refute the point of Sion and Carbunar [92], that the cost of CPIR is likely to be worse than the naive solution of transferring the entire library. Of these works, XPIR [11] is the fastest (by a good margin). However, its single request cost for media delivery is still higher than desirable; a detailed comparison with Popcorn is in Section 6.4 and Figure 16.

Yet another response has been to circumscribe, in exchange for better performance, the portion of the library for which the privacy guarantees hold [76, 77, 99]. As an example, for libraries that can be thought of as a matrix, bbPIR [99] 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.

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 [68], building on earlier theoretical work by Beimel et al. [18] and Ishai et al. [58], show that one can achieve sub-linear server-side computation by efficiently processing batches of requests from multiple clients. As discussed in Section 4.2, Popcorn is inspired by this work; it uses batching at multiple stages of its protocol but tailored for media delivery. Another recent system, RAID-PIR [32], based on the implementation of upPIR [24], reduces the server-side work, first, by storing and processing only a *fraction* of the library at each ITPIR server and, second, by encoding multiple requests from the same client in a single query. In principle, these techniques compose with Popcorn but would improve performance only for more than two servers, or when clients issue multiple simultaneous requests. Currently, Popcorn assumes exactly two servers and that individual clients request objects sequentially.

Finally, one can improve performance with trusted hardware [15, 57, 67, 93, 101]: 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. These schemes, however, are not a good fit for our context, as they require assuming trust in the trusted hardware's manufacturer.

A large body of literature focuses on instead reducing the communication overhead of PIR [43, 78]. Unlike Popcorn, these protocols target an environment in which $n \gg \ell$. In that context, Devet et al. [35] propose a technique that, like Popcorn, composes CPIR and ITPIR. Unlike Popcorn, the composition is hierarchical (ITPIR selects a sub-library, and iterations of CPIR select an object) and minimizes communication costs.

Protecting library content in PIR. The tension between ITPIR and content protection has been noted before. Gertner et al. [45] introduce the problem and propose two solutions, both of which, at a high level, protect the content by storing at untrusted servers independent random data (e.g., two servers store random data that XORs to the library content). Goldberg's ITPIR protocol [48] has a similar protection property as [45], but it uses fewer servers. Huang et al. [55] protect library content kept at untrusted servers by first encrypting it, and then using a threshold signature scheme [33] to serve keys for the encrypted object. Under all of the aforementioned schemes, the library content can be disclosed if more than a threshold of untrusted servers collude. By contrast, Popcorn keeps content protection collusion-proof, by composing CPIR and ITPIR (§4.1).

Symmetric PIR (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 [46]. While Popcorn does not use a SPIR scheme to privately download keys from the key server, because it assumes an honest client (§2.1), we note that it can reduce that trust by transforming its CPIR protocol into an SPIR protocol [37, 73].

Relatedly, 1-out-of-N oblivious transfer (OT) [20, 73] provides the same content protection property as SPIR but, unlike SPIR, can have network overhead linear in the size of the library. On the one hand, this overhead would not be costly in our experiments: in accordance with WebM Encryption (§6.5), our keys are 128 bits, which, for n=8192 objects, yields a library of only 128 KB. On the other hand, the linear overhead can be large in general (e.g., if the key server embeds keys within DRM licenses, or if the number of keys grows); for this reason, we do not use OT for Popcorn's key server.

Handling variable sized objects in PIR. A naive solution is to pad every object to the size of the longest, and download (the equivalent of) the longest object from each server. Prior work [32, 52] avoids this solution by (a) concatenating small objects (e.g., a few objects form one row of the library), and (b) splitting large objects over multiple rows of the library and using *multi-row queries* that retrieve (secretly) many rows in a single query. The reduced communication cost is close to the optimal: the size of the longest object in the library. However, this cost is still high, especially if a smaller object is being retrieved. An alternative is to download different rows (of an object) as independent objects, possibly at the cost of increasing the consumption delay [32]. Popcorn uses this technique for objects that are divided over multiple rows, but in addition, reduces the number of such objects by using a combination of compression and padding (§4.4).

Prior PIR implementations. Many of the CPIR and ITPIR works described above have been implemented. The Percy++ library [47] contains an implementation of the protocols from [12, 28, 34–36, 48, 53, 68]. Also, [52] is implemented as a fork of Percy++, RAID-PIR [32] is implemented on top of upPIR [24], and there are numerous CPIR implementations [11, 30, 41, 70, 72, 80, 87, 97, 99, 103], among which XPIR [11] is the fastest. Popcorn incorporates some of these implementations as modules: it uses the XPIR library for CPIR and borrows the CGKS ITPIR [28] code from Percy++. Sections 5 and 6 empirically or analytically compare Popcorn against these prior implementations.

8 Discussion, limitations, and future work

We evaluated Popcorn at the scale of a Netflix library, and found that the results are cautiously encouraging: compared to a baseline, I/O and CPU overhead are both lower (due to amortization, batching, and careful provisioning). And, although the overall resource cost is high, the *dollar* cost is manageable. Below, we discuss fundamental limitations of Popcorn, followed by limitations of the prototype and current design that require future work.

Fundamental limitations. Because Popcorn's overheads grow linearly with the number of objects, it has no hope of scaling to large libraries (like YouTube's).

A second issue is that collusion between object serving organizations compromises the privacy guarantee. On the one hand, for the very strongest (state-sponsored) adversaries, the assumption of no collusion is unrealistic: such an adversary can compromise both organizations (or already has). But Popcorn protects against many other adversaries. Queries are encrypted (over HTTPS) between the client and server; thus, compromising Popcorn's privacy guarantee requires attacking media servers or persistent storage in both object-serving organizations.

Third, Popcorn cannot support forward seeks during playback: such user actions alter the download pattern in a content-dependent way, thus revealing information.

Library updates. Popcorn does not currently support online updates, a feature that would be required in a real deployment. To support this, Popcorn should ensure that every ITPIR query executes on the same version of the library at both object servers. Standard solutions exist (e.g., generation numbers in concert with garbage collection).

Integration with CDNs. Running Popcorn on content delivery networks (CDNs) would present two main challenges: increased hardware provisioning at the CDN edge servers (power consumption, colocation space, etc.), and maintaining the utility of batching (§4) when running on a distributed infrastructure. Though increased hardware provisioning is non-trivial, we believe that it does not re-

quire a paradigm shift: Akamai already offers a distributed application service called EdgeComputing [31], which enables running CPU-intensive enterprise business web applications at edge servers. Moreover, Netflix recently installed custom built storage-optimized appliances at the edges.

Similarly, we believe that, though the CDN's distributed infrastructure will reduce opportunities for batching, enough concurrency will remain to make the service cost effective. Indeed, rough back-of-the-envelope calculations suggest that request rates for Netflix are already quite high (e.g., over 9,200 requests/90min/PoP[11]) and are growing fast [4]. This is not specific to Netflix: similar request rates (average of 6,000 requests in 90 minutes from within a single city) have been reported for other video on demand (VOD) systems [104].

Changes in load. Unless Popcorn is always wastefully provisioned for the peak load, load changes require care: the assignment of work units to machines depends on the number of clients (§4.3). A solution is to rely on virtual machines (VMs): give each VM a single slice, and then provide elasticity via VM migration or consolidation.

Variations in quality and bandwidth. Current media delivery services offer *adaptive streaming*: clients dynamically switch between different video quality levels to adjust to fluctuations. Popcorn can support this in two ways. The system can maintain an individual library for each video quality level. Each client sends query vectors to all libraries but downloads a given video chunk only from the appropriate one. This solution is simple but leads to a significant increase in server-side work as each individual library has to process each request. Alternatively, Popcorn could exploit layered coding [54, 61, 83, 89] or multiple description coding (MDC) [50, 86, 100]. There would be a single basic quality library that is accessed by all clients, with separate libraries for enhancement layers (better spatial resolution, bitrate, frame rate, etc.). The result would be adaptive streaming with server-side work that is proportional only to the size of the highest quality library.

Billing and accounting. Popcorn must enable the content distributor to charge consumers, pay royalties, and collect aggregate statistics. Popcorn's current prototype can support both subscription-based and pay-per-view pricing models by monitoring accesses to the key server. Furthermore, it by default works with a prepaid royalty model, where the distributor pays a fixed license fee upfront. However, in its current form, Popcorn does not support advanced pricing models (different prices for different objects, possibly in tiers) or advanced royalties models (e.g., based on number of views or aggregate statistics). However, we believe that these limitations are not fundamental (e.g., prior works [13, 22, 53, 96] have addressed them in different contexts). Future work is to investigate how these prior works can be composed with Popcorn, and what the implications of doing so are on performance and privacy guarantees.

Targeted ads and recommendation services. As with billing and accounting, in its current form, Popcorn does not support targeted advertisements or recommendations. Again, we believe that this limitation is not fundamental [16, 21, 23, 51, 60, 63]. Composing these works with Popcorn is an interesting research direction for future work.

[11] Assumes 10 billion hours watched in 3 months [3], requests are for a 90 minute video, and a total of 500 Points of Presence (PoP).

A Derivation of segment sizes

Recall the inequalities defined in Section 4.3:

$$t_i \leq T_i \cdot \alpha, \quad \text{where } \alpha = \frac{\min\{R_i, P_i/b_i\}}{\mu \cdot n}$$

$$T_i \leq d' + \sum_{j=1}^{i-1} t_j, \quad \text{where } d' = d - \epsilon$$

We consider the special case where both sides of the inequalities are equal. Combining both statements:

$$t_i = \left(d' + \sum_{j=1}^{i-1} t_j\right) \cdot \alpha$$

We show that $t_i = (d' \cdot \alpha \cdot (1+\alpha)^{i-1})$ is a solution to the above equation. Substituting on both sides:

$$d' \cdot \alpha \cdot (1+\alpha)^{i-1} = \left(d' + \sum_{j=1}^{i-1} d' \cdot \alpha \cdot (1+\alpha)^{j-1}\right) \cdot \alpha.$$

Summing the finite geometric series, and rearranging:

$$= \left(d' + d' \cdot \alpha \cdot \left(\frac{(1+\alpha)^{i-1} - 1}{\alpha}\right)\right) \cdot \alpha$$
$$= d' \cdot \alpha \cdot (1+\alpha)^{i-1}.$$

Setting $\alpha = 1$, we get $t_i = 2^{i-1} \cdot (d - \epsilon)$, as desired.

B Pricing model

Our high-level goal is to estimate the hourly cost of renting three resources on Amazon EC2: a vCPU, 1 GB of memory, and 1 Gbps of sequential read I/O bandwidth. To get the estimates, we make the simplifying assumption that the price of an EC2 machine depends only on these three resources. Of course, in practice, pricing machines is a complex process that depends on many factors (I/O performance for non-sequential workloads, cost of

the networking infrastructure, prices set by competitors, etc.); the values derived here should be treated as only estimates.

At a high level, our method is to use the specification of machines on Amazon EC2 and their corresponding prices to derive a system of linear equations; in these equations variables represent the unit cost of the resources mentioned above, coefficients represent the "quantity" of those resources in an Amazon EC2 machine, and the RHS will be the price of renting that machine.

We consider the machines in Figure 9 and an additional machine. We need this additional machine as the equation for i2.4xl is not linearly independent from that of i2.8xl, which leaves us with two equations to solve for three variables. To write the third equation, we pick a memory optimized machine that has 32 vCPUs, 244 GB of memory capacity, 2 SSDs with 320 GB capacity each (6.4 Gbps sequential read I/O bandwidth), and is rented out for $0.9822 per hour. Using these, we get the following equations:

$$32C + 60M + 6.4I = 0.6281$$
$$32C + 244M + 6.4I = 0.9822$$
$$32C + 244M + 23.3I = 1.6902,$$

where C is the hourly cost of renting a vCPU, M is the cost of renting 1 GB of memory for an hour, and I is the hourly cost for 1 Gbps of sequential read I/O bandwidth.

Solving for the unknowns in the equations, we get I=0.042, M=0.0019, and C=0.0076.

Acknowledgments

We thank Allen Clement, Alan Dunn, Yuval Ishai, Lon Ingram, 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 2015 Q1 Earnings Letter. http://files.shareholder.com/downloads/NFLX/47469957x0x821407/DB785B50-90FE-44DA-9F5B-37DBF0DCD0E1/Q1_15_Earnings_Letter_final_tables.pdf.

[4] Netflix Soars On Subscriber Growth. http://www.forbes.com/sites/laurengensler/2015/01/20/netflix-soars-on-subscriber-growth/.

[5] Netflix USA: Complete instant streaming list of all movies and TV shows. http://netflixusacompletelist.blogspot.com/.

[6] The 2014 Pulitzer Prize Winners, Public Service: The Guardian US and The Washington Post. http://www.pulitzer.org/works/2014-Public-Service.

[7] The WebM Project. http://www.webmproject.org/about/faq/.

[8] WebM Encryption. http://www.webmproject.org/docs/webm-encryption/.

[9] You are watching more web video ads than ever. http://allthingsd.com/20130215/you-are-watching-more-web-video-ads-than-ever/.

[10] 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 *IEEE International Conference on Computer Communications (INFOCOM)*, 2012.

[11] C. Aguilar-Melchor, J. Barrier, L. Fousse, and M.-O. Killijian. XPIR: Private Information Retrieval for Everyone. Cryptology ePrint Archive, Report 2014/1025, 2014.

[12] C. Aguilar-Melchor and P. Gaborit. A lattice-based computationally-efficient private information retrieval protocol. In *Western European Workshop on Research in Cryptology (WEWoRC)*, 2007.

[13] W. Aiello, Y. Ishai, and O. Reingold. Priced oblivious transfer: How to sell digital goods. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 2001.

[14] O. M. Alliance. DRM Architecture. http://technical.openmobilealliance.org/Technical/release_program/docs/DRM/V2_1-20081106-A/OMA-AD-DRM-V2_1-20081014-A.pdf, Mar. 2004.

[15] D. Asonov and J.-C. Freytag. Almost optimal private information retrieval. In *Workshop on Privacy Enhancing Technologies (PET)*, 2003.

[16] 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.

[17] E. Balsa, C. Troncoso, and C. Diaz. OB-PWS: Obfuscation-based private web search. In *IEEE Symposium on Security and Privacy (S&P)*, 2012.

[18] 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.

[19] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Advances in Cryptology—CRYPTO*, 2011.

[20] G. Brassard, C. Crepeau, and J.-M. Robert. All-or-nothing disclosure of secrets. In *Advances in Cryptology—CRYPTO*, 1987.

[21] M. Burkhart and X. A. Dimitropoulos. Fast privacy-preserving top-k queries using secret sharing. In

International Conference on Computer Communication Networks (ICCCN), 2010.

[22] J. Camenisch, M. Dubovitskaya, and G. Neven. Unlinkable priced oblivious transfer with rechargeable wallets. In *International Conference on Financial Cryptography and Data Security (FC)*, 2010.

[23] J. Canny. Collaborative filtering with privacy. In *IEEE Symposium on Security and Privacy (S&P)*, 2002.

[24] J. Cappos. Avoiding theoretical optimality to efficiently and privately retrieve security updates. In *International Conference on Financial Cryptography and Data Security (FC)*, 2013.

[25] D. Cash, J. Jaeger, S. Jarecki, C. Jutla, H. Krawczyk, M.-C. Rosu, and M. Steiner. Dynamic searchable encryption in very-large databases: Data structures and implementation. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[26] D. Cash, S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Highly-scalable searchable symmetric encryption with support for boolean queries. In *Advances in Cryptology—CRYPTO*, 2013.

[27] X. Cheng, C. Dale, and J. Liu. Statistics and social network of YouTube videos. In *International Workshop on Quality of Service (IWQoS)*, 2008.

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

[29] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progressions. *Journal of symbolic computation*, 9(3):251–280, 1990.

[30] W. Dai, Y. Doröz, and B. Sunar. Accelerating SWHE based PIRs using GPUs. Cryptology ePrint Archive, Report 2015/462, 2015.

[31] A. Davis, J. Parikh, and W. E. Weihl. Edgecomputing: Extending enterprise applications to the edge of the internet. In *International World Wide Web conference on Alternate track papers & posters (WWW Alt.)*, 2004.

[32] D. Demmler, A. Herzberg, and T. Schneider. RAID-PIR: Practical multi-server PIR. In *Cloud computing security workshop (CCSW)*, 2014.

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

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

[35] C. Devet and I. Goldberg. The best of both worlds: Combining information-theoretic and computational PIR for communication efficiency. In *Privacy Enhancing Technologies Symposium (PETS)*, 2014.

[36] C. Devet, I. Goldberg, and N. Heninger. Optimally robust private information retrieval. In *USENIX Security Symposium*, 2012.

[37] G. Di Crescenzo, T. Malkin, and R. Ostrovsky. Single database private information retrieval implies oblivious transfer. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 2000.

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

[39] Discretix Technologies. Secure implementations of content protection DRM schemes on consumer electronic devices. `http://www.discretix.com/wp-content/uploads/2013/02/secure_implementation_of_content_protection_schemes_on_consumer_electronic_devices.pdf`, Feb. 2013.

[40] 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.

[41] C. Dong and L. Chen. A fast single server private information retrieval protocol with low communication cost. In *European Symposium on Research in Computer Security (ESORICS)*, 2014.

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

[43] W. Gasarch. A survey on private information retrieval. *Bulletin of the European Association for Theoretical Computer Science (EATCS)*, 82:72–107, 2004.

[44] C. Gentry. Fully homomorphic encryption using ideal lattices. In *ACM Symposium on Theory of Computing (STOC)*, 2009.

[45] Y. Gertner, S. Goldwasser, and T. Malkin. A random server model for private information retrieval. In *International Workshop on Randomization and Approximation Techniques in Computer Science (RANDOM)*, 1998.

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

[47] I. Goldberg. Percy++ project on SourceForge. `http://percy.sourceforge.net/`.

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

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

[50] V. Goyal. Multiple description coding: compression meets the network. *IEEE Signal Processing Magazine*, 18(5):74–93, Sept. 2001.

[51] S. Guha, B. Cheng, and P. Francis. Privad: Practical privacy in online advertising. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[52] 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.

[53] R. Henry, F. Olumofin, and I. Goldberg. Practical PIR for electronic commerce. In *ACM Conference on Computer and Communications Security (CCS)*, 2011.

[54] U. Horn, K. Stuhlmüller, M. Link, and B. Girod. Robust internet video transmission based on scalable coding and unequal error protection. *Signal Processing: Image Communication*, 15(1):77–94, 1999.

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

[56] S. Huss-Lederman, E. M. Jacobson, J. Johnson, A. Tsao, and T. Turnbull. Implementation of Strassen's algorithm

for matrix multiplication. In *ACM/IEEE Conference on Supercomputing*, 1996.

[57] A. Iliev and S. Smith. Private information storage with logarithmic-space secure hardware. In *Information Security Management, Education and Privacy*, 2004.

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

[59] S. Jarecki, C. Jutla, H. Krawczyk, M. Rosu, and M. Steiner. Outsourced symmetric private information retrieval. In *ACM Conference on Computer and Communications Security (CCS)*, 2013.

[60] S. Jha, L. Kruger, and P. McDaniel. Privacy preserving clustering. In *European Symposium on Research in Computer Security (ESORICS)*, 2005.

[61] M. Johanson and A. Lie. Layered encoding and transmission of video in heterogeneous environments. In *ACM Multimedia (ACM-MM)*, 2002.

[62] J. Joskowicz and J. Ardao. Combining the effects of frame rate, bit rate, display size and video content in a parametric video quality model. In *Latin America Networking Conference (LANC)*, 2011.

[63] S. Katzenbeisser and M. Petković. Privacy-preserving recommendation systems for consumer healthcare services. In *International Conference on Availability, Reliability and Security (ARES)*, 2008.

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

[65] M. D. Lam, E. E. Rothberg, and M. E. Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.

[66] 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)*, 2013.

[67] J. R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *USENIX Conference on File and Storage Technologies (FAST)*, 2013.

[68] W. Lueks and I. Goldberg. Sublinear scaling for multi-client private information retrieval. In *International Conference on Financial Cryptography and Data Security (FC)*, February 2015.

[69] 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 Conference on Computer and Communications Security (CCS)*, 2013.

[70] T. Mayberry, E.-O. Blass, and A. H. Chan. PIRMAP: Efficient private information retrieval for MapReduce. In *International Conference on Financial Cryptography and Data Security (FC)*, 2013.

[71] T. Mayberry, E.-O. Blass, and A. H. Chan. Efficient private file retrieval by combining ORAM and PIR. In *Network and Distributed System Security Symposium (NDSS)*, 2014.

[72] 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)*, 2008.

[73] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *ACM Symposium on Theory of Computing (STOC)*, 1999.

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

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

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

[77] F. Olumofin, P. K. Tysowski, I. Goldberg, and U. Hengartner. Achieving efficient query privacy for location based services. In *Privacy Enhancing Technologies Symposium (PETS)*, 2010.

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

[79] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques (Eurocrypt)*, 1999.

[80] 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.

[81] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2014.

[82] Pomelo 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.

[83] H. M. Radha, M. Van der Schaar, and Y. Chen. The MPEG-4 fine-grained scalable video coding method for multimedia streaming over IP. *IEEE Transactions on Multimedia*, 3(1):53–68, 2001.

[84] D. Rayburn. CDN market trends: Pricing, growth and competitive landscape. In *Content Delivery Summit*, 2015.

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

[86] A. R. Reibman, H. Jafarkhani, Y. Wang, M. T. Orchard, and R. Puri. Multiple description coding for video using motion compensated prediction. In *International Conference on Image Processing (ICIP)*, 1999.

[87] F. Saint-Jean. Java implementation of a single-database computationally symmetric private information retrieval (cSPIR) protocol. Technical report, DTIC Document, 2005.

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

commentary/securitymatters/2006/05/70886.

[89] H. Schwarz, D. Marpe, and T. Wiegand. Overview of the scalable video coding extension of the H.264/AVC standard. *Transactions on Circuits and Systems for Video Technology*, 17(9):1103–1120, 2007.

[90] 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.

[91] K. D. Singh, Y. Hadjadj-Aoul, and G. Rubino. Quality of experience estimation for adaptive HTTP/TCP video streaming using H.264/AVC. In *Consumer Communications and Networking Conference (CCNC)*, 2012.

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

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

[94] E. Stefanov, E. Shi, and D. Song. Towards practical oblivious RAM. In *Network and Distributed System Security Symposium (NDSS)*, 2012.

[95] 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)*, 2012.

[96] V. Toubiana, A. Narayanan, D. Boneh, H. Nissenbaum, and S. Barocas. Adnostic: Privacy preserving targeted advertising. In *Network and Distributed System Security Symposium (NDSS)*, 2010.

[97] J. Trostle and A. Parrish. Efficient computationally private information retrieval from anonymity or trapdoor groups. In *International Security Conference (ISC)*, 2010.

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

[99] S. Wang, D. Agrawal, and A. El Abbadi. Generalizing PIR for practical private retrieval of public data. In *Working Conference on Data and Applications Security and Privacy (DBSec)*, 2010.

[100] Y. Wang and S. Lin. Error-resilient video coding using multiple description motion compensation. *IEEE Transactions on Circuits and Systems for Video Technology*, 12(6):438–452, 2002.

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

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

[103] X. Yi, M. G. Kaosar, R. Paulet, and E. Bertino. Single-database private information retrieval from fully homomorphic encryption. *IEEE Transactions on Knowledge and Data Engineering*, 25(5):1125–1134, 2013.

[104] H. Yu, D. Zheng, B. Y. Zhao, and W. Zheng. Understanding user behavior in large-scale video-on-demand systems. *ACM SIGOPS Operating Systems Review*, 40(4):333–344, 2006.