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. We describe the design, implementation, and evaluation of *Popcorn*, a media delivery system that hides clients' consumption (even from the content distributor). Popcorn relies on a powerful cryptographic primitive: private information retrieval (PIR). With novel refinements that leverage the properties of PIR protocols and media streaming, Popcorn scales to the size of Netflix's library (8000 movies) and respects current controls on media dissemination. The dollar cost to serve a media object in Popcorn is $3.87\times$ that 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 are people deeply uncomfortable with exposing their media diet, in particular to a centralized media server that can be targeted by either hacking or subpoena. They argue that, philosophically, freedom requires the ability to consume privately [94] and that, practically, access to a person's consumption profile can reveal the person's sexual orientation, political leanings, cultural affiliations, etc. [80, 81, 96].[1] And although many people may in fact *want* to expose their consumption to gain recommendations, there may still be content that they want to consume without others' knowledge. Another camp counters that media often exists within a commercial framework, and that people who create it and services that distribute it need to be compensated 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?*

No answer is likely to apply to all media delivery systems, as they differ widely. YouTube's library, for instance, is large, continuously updated, freely distributed, and supported by advertising. Netflix's library is comparatively small, updated infrequently [7], subject to strict content protection, and supported by paid subscriptions. This paper explicitly targets Netflix-like systems, and adopts the following requirements:

1. *Hide requests comprehensively and provably.* We want to hide consumption from both a network eavesdropper [9, 46] and the content distributor, and avoid the risk of heuristic solutions [20].
2. *Make it affordable even at scale.* Our system should dispense privacy at an attractive price point. The cost should be within a small multiple of what customers pay to access content today.
3. *Respect current controls on content dissemination.* Our solution must be compatible with the existing commercial, legal, and policy regime (copyright, controls on content dissemination, etc.) so as not to fundamentally reorient digital rights.

At first blush, Tor [41] and other anonymity systems [3, 72] (which conceal *who* consumes content) satisfy the above requirements. However, these solutions conflict with commercial media delivery: now Netflix would have to rely on the altruism of Tor nodes. Moreover, the capacity, latency, and reliability on a Tor network is unlikely to match the requirements of Netflix.

Thus, Popcorn turns to a large body of cryptographic protocols known as *Private Information Retrieval*, or *PIR* (§2.2). These protocols [31, 47, 69, 84, 109] allow clients (content consumers) to request content from servers (content distributors) without the servers being able to infer which items the clients requested.

Applying these protocols, however, raises several challenges (§3): the linear overhead of PIR (to respond to a request, the server must compute over its entire library, or else it would learn 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 surrounding PIR protocol choice (one type of PIR, called CPIR [69], needs only one server, but the overhead is high; another, called ITPIR [31], involves lightweight operations but demands non-colluding servers and hence separate administrative domains, which, among other things, threatens content protection). There is a large and inspiring body of work (§7) addressing some of these issues [14, 15, 27, 33, 35, 37–39, 45, 51, 52, 56, 57, 60, 74, 76, 78, 86, 104, 106, 110], but prior implementations suitable for media delivery at the scale we target levy prohibitive demands on 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—

[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.

with resource overhead that translates to 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 (§4). First, Popcorn combines 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 amortizes the cost of PIR by batching requests from the large number of concurrent users retrieving content at any given time; by leveraging the properties of media streaming, Popcorn forms large batches without introducing playback delays or interruptions. Third, Popcorn exploits the ability to encode a media object in multiple ways (e.g., by changing its bitrate) to meet the fixed-size-object requirement of PIR.

We experimentally evaluate Popcorn for a Netflix-like workload (10,000 concurrent clients, each streaming different content at 4 Mbps from a library of 8192 movies [1] with an average length of 90 minutes). Popcorn's overheads are high when compared to a non-private baseline: for each request, Popcorn consumes $1080\times$ more computational resources, about $14\times$ more I/O bandwidth, and $2\times$ longer network transfers. However, since CPU is cheap and Popcorn is engineered to conserve the more expensive resources (I/O and network), these overheads, when translated to dollars, are manageable: Popcorn's per-request cost, in terms of dollars, is $3.87\times$ 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 [30]). 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 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*. We assume that a mapping, between the integers $1, \ldots, n$ and the names of the objects in L, is known to the distributor and the consumers. Therefore, a consumer can select a specific object by supplying the corresponding integer.

Threat model. We consider an attacker (for example, the content distributor or a network eavesdropper) trying to infer what object the consumer is accessing. The attacker has full access to the network and to the content of the consumers' requests, but for two restrictions. First, we do not consider side-channel attacks that, for example, use knowledge of where individual consumers pause playback, or of their concurrent web browsing activity. Second, we assume the existence of two non-colluding servers that the distributor can use to serve content. To satisfy this assumption in practice, one can pick servers from separate administrative domains (e.g., from different CDNs [13]). We discuss this topic further in Section 8.

We assume, as do today's media delivery systems [17, 42], that the client-side media decode and display environment can defeat content consumers intent on copying and redistributing content beyond what the distributor allows.

Finally, we treat content integrity as an orthogonal problem that 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. To privately retrieve object $O_b = L[b]$, the client invokes $\mathsf{Query}(b)$ to produce k query vectors $q_1, \ldots, q_k$, one for each server, and forwards q_j to server S_j ($1 \leq j \leq k$). Each S_j replies with $a_j = \mathsf{Answer}(q_j, L)$. Finally, the client computes $O_b = \mathsf{Decode}(a_1, \ldots, a_k)$ by applying the decode algorithm to the servers' responses.

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

```
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 $\ell/y$ 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 1—A computational PIR (CPIR) protocol based on an additively homomorphic cryptosystem (Gen, Enc, Dec) and due to Stern [101]. (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.

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 [69] require only a single, computationally bound server ($k = 1$). They are commonly constructed using additively (not fully [48]) 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, $+$ represents addition of two plaintext messages, $\cdot$ is a binary operation (for example, addition, multiplication, etc.) on the ciphertexts, (pk, sk) is a (public, private) key pair generated using the key generation algorithm Gen, Dec is the decryption algorithm, and Enc is the encryption algorithm. Note that Enc is randomized; thus, repeatedly encrypting the same plaintext produces different ciphertexts. Examples of cryptosystems used in CPIR are the Paillier [85] and the lattice-based Ring-LWE [22].

Figure 1 depicts a CPIR protocol, due to Stern [101], that meets the three properties (§2.2):

- ***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 $c_b = \mathsf{Enc}(pk,\ 1)$), then S could likewise guess which entry is the encryption of 1 (versus 0)—which would contradict the properties of the underlying encryption scheme.

```
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 $\ell$ bits
    // $q$ is a row vector, $L$ a logical matrix: $L \in \{0,1\}^{n \times \ell}$
    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 2—The ITPIR protocol of CGKS [31]. n is the number of objects in library L, and ℓ is the length of each object. k is the total number of servers. (In Popcorn, k=2.)

- ***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.

2.4 Information-theoretic PIR (ITPIR) protocols

ITPIR protocols [31] 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 2 shows the CGKS [31] 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 $S_1, \ldots, S_{k-1}$ 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 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 combined length of the servers' reply is $k \cdot \ell$ bits. In Popcorn, we set $k = 2$ to keep this comparable to ℓ, the size of an object. A client's request consists of k n-bit-long query vectors, which is much smaller than ℓ when k is small.

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

	I/O	CPU	content prot. (ITPIR)	resists collusion	object sizes	pricing, reco
XPIR [14]		◐	●	●		
RAID-PIR [35]					◐	
Percy++ [51]		◐	◐	◐	◐	●
Popcorn	●	◐	●		◐	

Figure 3—Prior PIR-oriented works (rows) and which media-related challenges they address (columns), assuming two servers for ITPIR-based works. ● means that the work addresses the challenge; ◐ means that it partially addresses the challenge.

3 Challenges of applying PIR

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

- *Resources.* The I/O and CPU resources required to serve a single request are proportional to the size of the library. Batching requests should help amortize some of this overhead, but it 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]
- *Billing, access control, recommendations.* For business reasons, media services may need to support access control, pricing policies (tiers, etc.), targeted advertising, and recommendations. Yet, private retrieval conflicts with all of this functionality.

Subsets of these challenges have been addressed before (Figure 3). Popcorn aims mainly at the resource consumption issue, via the architecture and design described next.

4 Architecture and design of Popcorn

Figure 4 depicts Popcorn's architecture. 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.

[3]The state of the art CPIR implementation is XPIR, which is based on the Ring-LWE cryptosystem. XPIR can process data at 22 Gbps on a machine with 4 physical (and 8 virtual) cores [14], while the CGKS ITPIR implementation in Percy++ [51], based on cheaper XOR operations, can process data at 152 Gbps on comparable hardware.

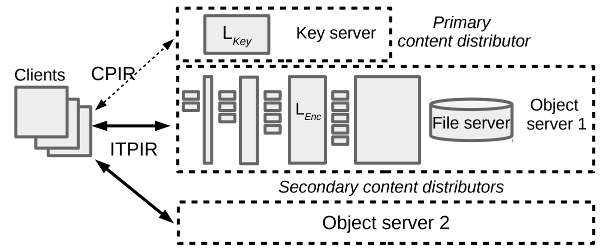


Figure 4—Popcorn's architecture. Popcorn uses two servers for ITPIR. Each object server stores all of the columns in the library (Figure 5), and is distributed over multiple physical machines.

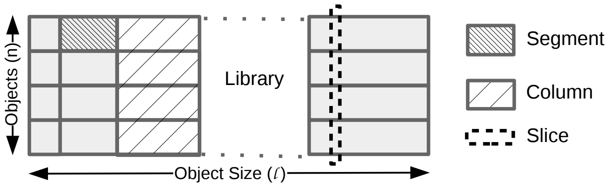


Figure 5—Popcorn terminology. Each column is stored by two ITPIR instances (one from each object server). Columns are divided into slices, which are assigned to physical machines.

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 today's DRM implementations [2, 4, 88], where clients contact two separate servers, one for encrypted video and one for decryption keys.

Media objects are split into *segments*—contiguous pieces of media 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 (we revisit this assumption in §4.4). The library is partitioned into *columns* (Figure 5); a column is a union of corresponding segments, across all objects. Therefore, a column's size is n times that of 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 *chunk*. Each machine is responsible for one or more slices.

To retrieve an object, the client fetches a decryption key from the key server and the encrypted object from the object servers. The latter step proceeds in two overlapping phases. In the first phase, the client sends, in parallel, a query vector to all machines in both object servers. 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, and pushing the resulting chunks to a file server (one per object server; Figure 4) that 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 (Figure 2). 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 Symmetric PIR (SPIR) or 1-out-of-n Oblivious Transfer (OT). Section 7 discusses these alternatives.

4.2 Batching

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

To reduce costs, Popcorn's machines, which are oblivious to the content of queries, process queries in *batches*, and perform a single I/O pass over a slice for all of the queries in a batch. Batching thus amortizes I/O overhead and lets Popcorn exploit sequential transfer bandwidth.

Batching also reduces *computational* (not just I/O) overhead by leveraging the observation that the PIR computation required for a batch of requests can be expressed as matrix multiplication ($q \cdot L$ in Figure 2 can be replaced by $Q \cdot L$, where Q is a matrix whose rows are query vectors). Previous work [21, 74] (covered by the Percy++ row in Figure 3) has used this observation to incorporate sub-cubic algorithms [32, 61] that reduce the total number of operations required by PIR. Popcorn, by contrast, chooses block matrix multiplication [71], 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).[4] However, there is a tension between large

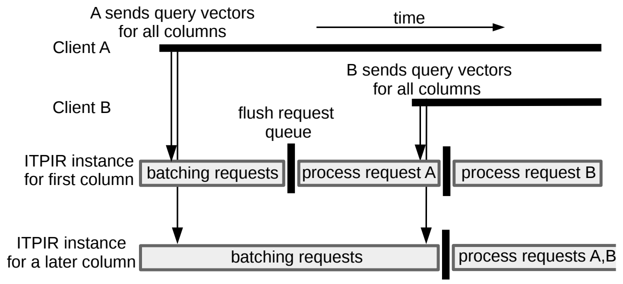

A sends query vectors for all columns
time
Client A
B sends query vectors for all columns
Client B
flush request queue
ITPIR instance for first column
batching requests
process request A
process request B
ITPIR instance for a later column
batching requests
process requests A,B

Figure 6—Batching at an object server in Popcorn. Requests to the initial column from two clients A,B are in separate batches as the processing cycle for this column is short. The requests to a later column (sent alongside the requests to the first) can be batched. This arrangement is inspired by Pyramid Broadcasting [105].

batch sizes, which seem to require synchronizing clients, and meeting the deadlines of real-time media delivery. Popcorn resolves this tension as follows.

To begin with, each ITPIR instance loops over its assigned column (§4) continuously. Since a client can begin playback only after decoding the response for the first column, Popcorn uses a "narrow" first column to keep this initial delay short. Column width, however, increases quickly in Popcorn, making later columns wide. The crucial intuition is that *wide columns imply good batching opportunities*: a batch comprises all requests that reached an ITPIR instance during its previous loop interval, and wider columns imply longer loop intervals.

Figure 6 depicts this arrangement. It is inspired by Pyramid Broadcasting (PB) [105] (see also [8, 58]), wherein an object is divided into increasingly-sized pieces, each served on a separate broadcast channel that loops over the piece. Differences are as follows. Whereas PB targets network bandwidth efficiency (and clients must buffer), Popcorn aims to reduce server-side I/O (and the buffer is at the server); one can view Popcorn's arrangement as the I/O subsystem using PB to "broadcast" to the next layer in the pipeline (the XORs). Furthermore, in Popcorn, a server's work (the XORs) depends on the number of clients (unlike in a broadcast setting). Finally, in Popcorn, each instance is distributed over multiple physical machines. These differences lead to a design and analysis that owe a debt to PB but are specific to our context.

Details. We start with two simplifying assumptions, which we revisit later: that a single ITPIR 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

[4]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.

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? Consider only instance 1: in the worst case, a client initiates consumption just after instance 1 begins its processing cycle. The client cannot download until the current processing cycle has terminated (which takes time T_1) and the first slice of the next cycle is processed (for an additional ϵ). Smooth playback simply requires the overall delay ($T_1 + \epsilon$) to be less than d, matching our conditions. Once playback begins, the client has t_1 additional time before it needs the second segment. Generalizing, in the worst case for instance i (i.e., the client's initial request arrives just as a processing cycle begins), as long as T_i is no larger than $d - \epsilon + \sum_{j=1}^{i-1} t_j$ (which is exactly what our conditions 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? Recall that, for more effective batching, Popcorn needs segment widths to increase: we are then seeking the maximum t_i for each instance i .

Let μ be the playback rate, P_i the rate at which XOR operations are processed by the ith instance, R_i the I/O bandwidth available to the instance, and b_i 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, the column's data (n segments, each of size $t_i \cdot \mu$) 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, 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:

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

(see Appendix A). Note that the $\{t_i\}$ increase exponentially in size, 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 across 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. P_i, the XOR processing throughput for instance i, increases with i because so does the batch size b_i; specifically, if λ is the overall rate at which clients initiate requests for objects, then $b_i = \lambda T_i$. Moreover, the per-machine XOR processing throughput for the ith instance, $p_i(\cdot)$, is a function of the batch size because cache locality in block matrix multiplication (§4.2) (and hence throughput) improves with a a bigger batch size. Thus, the task of processing the XOR operations for instance i can be handled by $P_i/p_i(b_i) = \mu \cdot n \cdot b_i/p_i(b_i)$ machines.

To account for the striping, we need to modify the earlier analysis of startup delay, smooth playback, etc.: 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 both the $\{T_i\}$ and $\{t_i\}$ are computed accordingly.[5]

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 max is controlled by the first term, then the given instance is bottlenecked by I/O (and the CPU resource is sometimes idle); if by the second, 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 the buffer space for each instance to equal the number of requests in service times the size of a segment, i.e., $\sum_{i=1}^{I} b_i \cdot (t_i \cdot \mu)$. 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 β. To this end, T_i should be upper-bounded by $d - \epsilon \cdot k_i - \beta + \sum_{j=1}^{i-1} t_j$, and the $\{t_i\}$ computed to be consistent with T_i.

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}. Let a *cohort* denote the set of clients who initiate a request (for the first chunk of a media file) in an epoch. Then, the entire cohort moves through the slices, as it were, together. Each cohort needs enough machines to meet two requirements: (a) $\mu \cdot n$ I/O bandwidth, and

[5]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.

(b) $\mu \cdot n \cdot \lambda \cdot T_{\text{epoch}}$ XOR processing throughput (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 processing throughput for a batch size of $\lambda \cdot T_{\text{epoch}}$. Here, T_{epoch} must be upper-bounded by $d - \epsilon \cdot k - \beta$ 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 the simplifying and optimistic assumption that both schemes use machines that make the two terms of the respective maxes equal, so that no resources are idle (we will revisit this assumption in §6.2 and §6.4). Then, the total I/O bandwidth required by the cohort scheme is $H \cdot \mu \cdot n$, which is considerably larger than what Popcorn needs ($I \cdot \mu \cdot n$, where $I \ll H$).

In terms of computational resources, the cohort scheme needs $\mu \cdot n \cdot \lambda \cdot T/p(\lambda \cdot T_{\text{epoch}}) = \mu \cdot n \cdot \lambda \cdot \sum_{i=1}^{I} T_i/p(\lambda \cdot T_{\text{epoch}})$ machines; Popcorn requires instead $\mu \cdot n \cdot \lambda \cdot \sum_{i=1}^{I} T_i/p_i(\lambda \cdot T_i)$ machines. Neither scheme is the clear-cut winner; however, if we assume that $p(\cdot) = p_i(\cdot)$ for all i, then Popcorn has lower computational demands, because (a) $T_i \approx 2^{i-1} \cdot T_{\text{epoch}}$ (by our earlier analysis) and (b) $p(\cdot)$ is monotonically increasing. In essence, Popcorn has larger batches, so (holding machine type configuration constant) the benefit of locality is more pronounced (§4.2), lowering Popcorn's computational requirements relative to the naive batching scheme.

4.4 Handling variable-sized objects

The design has so far assumed equally sized objects. A naive solution would be to pad all objects to the size of the longest one. However, this would, for Netflix, cause a $4\times$ increase in network transfers: the average movie is approximately 1.5 hours while the longest is 6, and clients would have to download the padding in full (doing otherwise would reveal the true object size).

Popcorn's solution instead chooses a representative object O_{avg} from the library (for example, the object closest to the average media length) and pads smaller objects to that size. Longer objects, up to a cutoff, are compressed down to O_{avg}'s size, by reducing the bitrate;[6] the longest objects are split into several files.

We note that reducing the bitrate is likely to be tolerable, as variations of up to 30% (roughly) in video bitrate have a limited impact on user satisfaction [43, 67, 97]. (Other factors, such as playback interruptions and startup times, instead have substantial impact.) Objects that cannot be tolerably compressed must be divided up (as in other systems [35, 56]). 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).

The Netflix catalog [1] 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. Movies between 92 and 120 minutes will require 23% compression in the worst case and 10% on average; similarly, the padding for objects between 60 and 92 minutes will be small to moderate. The impact of objects at either extreme will be limited: 8% of the movies are shorter than 60 minutes, and will require significant padding; 5% are between 120 and 135 minutes, making them candidates for aggressive compression (32% in the worst case and 27% on average) though potentially at the cost of lowering user satisfaction; and 2% are over 135 minutes, making them candidates for splitting. We think that splitting is not a huge limitation, because we hypothesize that people usually plan ahead to watch long movies.

5 Implementation

Our prototype implements the design in Section 4, except for large file splitting (§4.4). It leverages existing PIR implementations: the key server uses the XPIR [14] implementation of the CPIR protocol in Figure 1. For the object servers, we borrow the CGKS ITPIR implementation of Percy++ [51][7] and modify it to support the techniques in Section 4. The total server-side code is 11K lines of C++. We implement two versions of the client-side library: one in C++ (2500 lines), which we use for experiments (§6.2), and one in JavaScript (500 lines), which we use to show 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 7 summarizes our evaluation results.

Method and setup. We compare Popcorn to three baselines. *NoPriv*, *BaselinePIR*, and *BaselinePIR++*. NoPriv serves object chunks from an Apache web server, modeling modern media delivery systems that use HTTP

[6]Regardless of an object's bitrate, a client must issue chunk download requests at a constant rate (e.g., one request every 1 MB/μ seconds, where 1 MB is a chunk's size and μ is O_{avg}'s bitrate); otherwise, chunk download patterns would leak information.

[7]Percy++'s CGKS ITPIR implementation is one of the fastest implementations for two-server ITPIR. An alternative is the CGKS implementation from RAID-PIR [35] (§7).

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 7—Summary of main evaluation results.

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

caching at CDN edge servers [13]. BaselinePIR is a modified version of Percy++ [51] CGKS: the servers store the library L as slices and process ITPIR queries directed at them. This is essentially Popcorn without the techniques of §4. BaselinePIR++ additionally batches requests using cohort batching (§4.3) to reduce both I/O and CPU costs. For all PIR systems, we experiment with one object server and multiply the measurements by two (to reduce the financial cost of our experimental evaluation).

Our workload is modeled on existing media delivery services [102]: clients arrive according to a Poisson process (e.g., C=10K clients arrive in T=90 minutes). All clients in NoPriv request the same (average-size) object, giving this baseline the maximum benefit of server-side caching. The server's work in Popcorn, BaselinePIR, and BaselinePIR++ is oblivious to the request distribution (we select a Zipfian distribution with θ=0.8).

For the four 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 within Amazon's EC2, and is described in Figure 8.

6.1 Provisioning resources using microbenchmarks

Popcorn. Machine provisioning for Popcorn involves two steps: (1) benchmarking the basic operations (details in Figure 9), and (2) combining the results with the provisioning analysis in §4.3.

Consider, for example, 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=8192 media files with average playing time of T=90 minutes, playback rate of μ=4 Mbps, and startup delay of d=15 seconds.[8] The processing cycle of this

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

Figure 9—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 8. The latter value depends on the size of the query matrix (§4.2, §4.3), so we report a range: from a query matrix consisting of a single query vector to one that contains 4096 query vectors.

instance must be $T_1 \le d - \epsilon \cdot k_1$ (§4.3). For our example, ϵ=2 (the time to process or consume a 1 MB chunk at μ=4 Mbps), and we speculatively set k_1=3, which gives $T_1 \le 15 - 2 \cdot 3 = 9$ 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 XOR processing throughput $P_1 = b_1 \cdot n \cdot \mu = 544$ Gbps.

Our microbenchmarks (Figure 9) indicate that these requirements can be met by three i2.4xl machines. If the microbenchmarks had indicated a different number, then, as described in §4.3, we would have had to adjust k_1 (which was speculatively set) and repeat the provisioning process described above.

BaselinePIR. To use the fewest possible machines, we stripe the approximately 21 TB library of our Netflix-like workload across machines with highest storage capacity (that is, i2.8xl in our testbed).

To reduce the financial cost of our experimental evaluation, we measure the number of requests that can be serviced by this setup, along with each request's resource consumption, and extrapolate the results to workloads with a larger number of requests (e.g., to support 2× concurrent clients, we double resource costs).

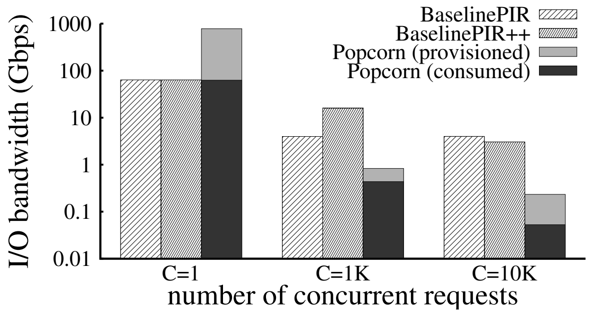
BaselinePIR++. As in Popcorn, we use the microbenchmarks in Figure 9 and the provisioning analysis for the cohort batching scheme (§4.3).

6.2 Per-request overheads of Popcorn

To understand when Popcorn is affordable, we run experiments varying 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 moderate (≈8K media files), object sizes are large (≈90 minutes), and there are many concurrent clients (≥10,000). Fortunately, these settings are consistent 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

[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 [12]).



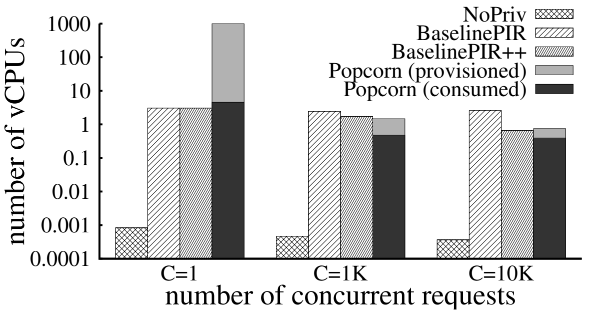


Figure 10—Per-request server-side resource use (log-scaled) of Popcorn and the baselines with varying concurrent requests C. If I/O is the bottleneck, there are idle CPU cycles and vice versa (§4.3). For Popcorn, we depict both the provisioned and consumed resources; for the baselines, we depict only the latter. We do not depict I/O usage for NoPriv as it is always zero (see text).

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 $C=\{1, 1K, 10K\}$ while keeping $n=8192$, $T=90$ min, $\mu=4$ Mbps, and $d=15$ seconds. Figure 10 summarizes the per-request server-side resource costs.

I/O overheads. When $C=1$, the I/O bandwidth Popcorn consumes matches that of BaselinePIR and BaselinePIR++, as there is no opportunity to batch requests. However, as the request rate increases, batching lets Popcorn amortize its I/O transfers (§4.3): the per-request amortized I/O bandwidth decreases from ≈ 63 Gbps (for $C=1$) to 53 Mbps (for $C=10K$), a reduction of 1190×. Surprisingly, concurrent requests, by hitting the file system cache, also reduce BaselinePIR's per-request I/O bandwidth (by 16×). As expected, BaselinePIR++'s per-request I/O bandwidth reduces by the cohort batch size. Finally, 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 $\approx 11\times$ when the number of concurrent requests increases from 1 to 10,000. We hypothesize that this stems from 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, consistent with the throughputs reported in Figure 9).

However, Popcorn's per-request CPU consumption is much higher than NoPriv (1080× for $C=10K$): for a single object, the Apache web server in NoPriv serves 1 MB chunks and requires almost no server-side processing, whereas Popcorn XORs n objects on average.

Network and storage overheads (not depicted in the figures). BaselinePIR, BaselinePIR++, and Popcorn incur a two-fold network overhead over NoPriv because clients download from two servers. 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). Across all instances, this 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. In Figure 11(a), we change the size of the library ($n=\{2048, 4096, 8192\}$) while keeping the other parameters fixed ($C=10K$, $T=90$ min, $\mu=1$ Mbps,[10] and $d=15$ seconds). 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 shown) do not change with n.

Overhead versus playing time of objects. In Figure 11(b), we change the playing time of objects ($T=\{10, 60, 90\}$ minutes) while keeping the other parameters fixed ($n=2048$, $\mu=1$ Mbps, $d=15$ seconds, and $C=10K$). As T increases, the per-request CPU consumption is unaffected. Also, with increasing T, the per-request I/O consumption decreases; on the other hand, idle I/O bandwidth (not depicted in the figure) increases (§4.3).

Overheads of the key server. Recall that Popcorn uses XPIR [14] 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 (and not on the number of concurrent requests C). We use a single machine of type c3.8xl for the key server. For a library with 8192 keys, it takes three seconds of server-side CPU time to privately retrieve a key; there are no I/O transfers

[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 $\mu=1$ Mbps instead of 4 Mbps. The change scales down the experiments; the qualitative results are unaffected.

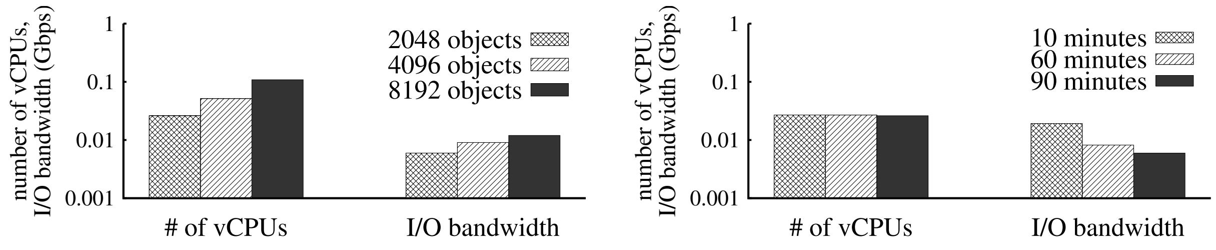


Figure 11—Popcorn per-request resource use (log-scaled) as a function of the number (left) and length (right) of objects.

	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 12—Estimated per-request dollar cost for NoPriv and Popcorn. #1, #2, and #3 refer to the type of AWS EC2 machines from Figure 8.

because the 128 KB library fits in memory. Thus, as expected, the key server is not a performance bottleneck for Popcorn. Moreover, the end-to-end time to retrieve a key is much less than the startup delay of $d=15$ seconds.

Client-side overheads. Compared to NoPriv, Popcorn's client consumes additional CPU and network bandwidth (because it has to generate and decode PIR queries, and download content from two object servers). For $n=8192$ objects, $T=90$ minutes, and $\mu=4$ Mbps, we find that Popcorn's client (run on a single vCPU of c3.8xl type machine) 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 the baseline PIR systems, at least for large objects and high load. These improvements provide the foundation for achieving privacy at low cost, a cost that we now quantify.

Method. We use the pricing model of Amazon EC2 (Figure 8) to estimate the per-request machine cost, and the pricing model of CDNs ($0.006 per GB) [90] to compute per-request network cost. We choose these pricing models because they are public—though, 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, $\mu=4$ Mbps with varying number of concurrent clients. Figure 12 summarizes our results. We find that Popcorn's per-request cost is within a small multiple of NoPriv for a workload with $C=10$K 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 $\approx$ $0.016 (i.e., 90 minutes $\times$ 4 Mbps $\times$ $0.006/GB).

Popcorn. We provision EC2 machines as described in §6.1 and §4.3. 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 10). For the Netflix-like library and $C=10$K, the per-request cost is $0.062 (the per-request machine cost is $0.03; the per-request network cost is $0.032).[11] Popcorn thus increases dollar cost 3.87$\times$ over NoPriv, 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 $C=1$K.

BaselinePIR and BaselinePIR++. Since we might have provisioned these systems wastefully, we do not estimate their dollar cost using the machine-based pricing model, which charges for both the consumed and idle resources. Instead, we use a per-resource pricing model to estimate the dollar cost of these systems, as described next.

6.4 Further comparisons

In this subsection, we estimate the dollar cost of BaselinePIR, BaselinePIR++, XPIR [14], and XPIR++, a hypothetical extension to XPIR that uses cohort batching (§4.2) to reduce I/O costs (but does not use matrix multiplication). Figure 13 summarizes these alternatives.

The estimates for BaselinePIR, BaselinePIR++, and Popcorn are based on experimental data from §6.2; for XPIR and XPIR++, we calculate CPU resource consumption using XPIR's reported performance and I/O bandwidth consumption from the expression $2{\cdot}(n{\cdot}\mu)/b_{\text{cohort}}$ (the factor of two is due to XPIR's preprocessed library being twice the size of the original [14]).

We compare these systems for the Netflix-like workload of §6.3. We set the startup delay d to 15 seconds, except for the systems using cohort batching scheme, for which we vary d.

[11] The network cost can be reduced for a pricing model in which network transfers between (ITPIR) servers is cheaper than server to client transfers, by using the techniques of Riffle [70, Section 4.4] (§7).

system	description
XPIR [14]	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 13—Comparison points. "Naive batching" refers to an instantiation of batching, as described in Section 4.2, with the cohort batching scheme described in Section 4.3.

	# vCPUs	I/O bandwidth (Gbps)	Dollar cost relative to NoPriv
X [14]/X++ (C=1)	11.6	64	265×
X++ (C=1K)	11.6	26.6	118×
X++ (C=1K, d=60)	11.6	5.96	37×
X++ (C=1K, d=600)	11.6	0.58	16×
X++ (C=10K)	11.6	2.66	24×
X++ (C=10K, d=60)	11.6	0.59	16×
X++ (C=10K, d=600)	11.6	0.058	13.5×
B/B++ (C=1)	3.1	64	256×
B (C=1K)	2.4	4	19×
B (C=10K)	2.5	4	19×
B++ (C=1K)	1.7	16	66×
B++ (C=1K, d=60)	1.26	9.15	39×
B++ (C=1K, d=600)	0.49	0.54	4.5×
B++ (C=10K)	0.65	3	14×
B++ (C=10K, d=60)	0.49	0.59	4.7×
B++ (C=10K, d=600)	0.41	0.058	2.5×
P (C=1)	4.6–992	63–781	253×–4873×
P (C=1K)	0.5–1.47	0.43–0.83	4×–7.6×
P (C=10K)	0.4–0.74	0.053–0.23	2.5×–3.87×

Figure 14—Per-request resource consumption and estimated dollar-cost of XPIR (X), XPIR++ (X++), BaselinePIR (B), BaselinePIR++ (B++), and Popcorn (P). Network transfers are not shown; they are 5× NoPriv for X and X++, and 2× NoPriv for the other systems. For Popcorn, we present a range: the smallest value considers only the consumed resources, while largest value includes both consumed and idle resources. Startup delay d is 15 seconds unless specified otherwise.

We use a per-resource pricing model (derived in Appendix B) based on Amazon EC2's machine cost (Figure 8) and on the network cost of CDNs [90]. Our model charges CPU at $0.0076/hour, I/O bandwidth at $0.042/Gbps-hour, and network transfers at $0.006 per GB. Multiplied by each system's consumption of the corresponding resources, these values determine the per-request dollar cost (Figure 14).

- The costs of XPIR are high (265× NoPriv), though adding a naive batching scheme (XPIR++) significantly reduces them (by ≈11× for C=10K, d=15).
- Using ITPIR for object delivery (in conjunction with CPIR (§4.1)) reduces the costs further (by ≈ 2× for C=10K, d=15). The disadvantage is that ITPIR requires non-colluding servers.
- Increasing the startup delay (and thus the batch size of the cohort) can further reduce costs. For example, increasing d from 15 to 60 seconds reduces costs by 3× (a reduction from 14× NoPriv to 4.7× NoPriv).
- BaselinePIR++ matches the cost of Popcorn (when C=10K) but requires a 40× higher startup delay (d=10 minutes in BaselinePIR++ vs. 15 seconds for Popcorn).

6.5 Compatibility study of Popcorn

To verify Popcorn's compatibility with modern Web browsers and DRM technology, we implemented a Popcorn client in JavaScript and used it to watch short videos in the WebM format [10] (protected using WebM Encryption [11]). Our prototype works on Chrome (version 45.0.2454), and makes use of the HTML5 video tag and extensions: the decoded ITPIR content is passed into the Media Source Extension interface, which forwards media chunks to the video player; the decoded CPIR response is passed into the Encrypted Media Extension interface, which decrypts the protected content.

7 Related Work

Alternatives to PIR for privacy. *Obfuscation* [20, 44, 91] protects clients' privacy by cloaking traffic with dummy requests. This approach requires less processing than PIR at clients and servers, but significantly higher network cost: matching PIR's degree of privacy (the number of objects among which a request is hidden) would require downloading the entire library.

Rather than the content being consumed, *anonymity* hides the identity of the consumer [41, 72]. This could be used to hide metadata (login times, download frequency, etc.), which is complementary to PIR. However, anonymity-based solutions can reveal access patterns that, combined with other background information, may disclose a user's media consumption [80].

Oblivious RAM (ORAM) [53, 75, 77, 100] algorithms conceal a client's access patterns from a storage server. Similarly, *searchable symmetric encryption* (SSE) (surveyed in [28, 29]) offers yet another solution for private data retrieval from a remote database. However, these solutions target a setup where the client outsources its encrypted data to a server.

Recent results [64, 87] 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.

Distributing the work, either by moving it to the cloud or by dividing it among clients [37, 76, 86], reduces latency but not the total computational burden.

GPUs [33, 78] and cheaper cryptographic operations [14, 15, 45, 104, 110] have reduced the computational load of CPIR, refuting the notion [98] that CPIR is likely to be more expensive than the naive solution of transferring the entire library. However, the single request cost for media delivery in XPIR [14], the fastest system employing these techniques, is still higher than desirable (see §6.4 and Figure 14 for a comparison with Popcorn).

Another path to better performance is to limit the privacy guarantees to only a portion of the library [82, 83, 106]. For example, bbPIR [106] allows users of libraries that can be thought of as a matrix to specify a submatrix (called a bounding box) from which bits can be privately retrieved using CPIR. This approach 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 [74], building on earlier theoretical work by Beimel et al. [21] and Ishai et al. [63], show that one can achieve sub-linear server-side computation by efficiently processing batches of requests from multiple clients. 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 [35], based on the implementation of upPIR [27], reduces 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. Popcorn's performance could potentially benefit from these techniques, but only when using more than two servers, or when clients issue multiple simultaneous requests. Currently, Popcorn assumes exactly two servers and that clients request objects sequentially.

Finally, performance can be improved with dedicated hardware [18, 62, 73, 99, 108], at the price of having to trust its manufacturer: a client can connect to a secure coprocessor that (obliviously to the server hosting the library) retrieves and delivers the requested object.

A large body of literature focuses on instead reducing the communication overhead of PIR [47, 84]. Unlike Popcorn, these protocols target an environment in which $n \gg \ell$. In that context, Devet et al. [38] 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.

In very recent work, Riffle [70], like Popcorn, targets the case $\ell \gg n$. Unlike Popcorn, Riffle focuses on peer-to-peer file transfers (as opposed to centralized streaming media). Riffle uses ITPIR, with $k > 2$, and focuses on reducing server-to-client network transfers (§2.4), by adding server-to-server transfers; this could potentially be composed with Popcorn.

Protecting library content in PIR. The tension between ITPIR and content protection has been noted before. Gertner et al. [49] 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 [52] has a similar protection property as [49], but it uses fewer servers. Huang et al. [60] protect library content kept at untrusted servers by first encrypting it, and then using a threshold signature scheme [36] to serve keys for the encrypted object. In all the above schemes, the library content can be disclosed if more than a threshold of untrusted servers collude. By composing CPIR and ITPIR (§4.1), Popcorn instead keeps content protection collusion-proof.

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 [50]. Popcorn currently assumes an honest client (§2.1) and thus does not use SPIR to privately download keys from the key server; however, it can reduce that trust by transforming its CPIR protocol into an SPIR protocol [40, 79].

1-out-of-N oblivious transfer (OT) [23, 79] provides the same content protection property as SPIR but, unlike SPIR, can have network overhead linear in the size of the library. In our experiments, this overhead would not be costly: WebM Encryption (§6.5) sets our keys to 128 bits, which, for n=8192 objects, yields a library of only 128 KB. However, the linear overhead can in general be large (e.g., if the key server embeds keys within DRM licenses; for this reason, Popcorn's key server does not use OT.

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 [35, 56] 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 [35]. 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 protocols described above have been implemented. The Percy++ library [51] contains several of them [15, 31, 37–39, 52, 57, 74]. Also, [56] is implemented as a fork of Percy++, RAID-PIR [35] is implemented on top of upPIR [27], and there are numerous CPIR implementations [14, 33, 45, 76, 78, 86, 93, 104, 106, 110], among which XPIR [14] is the fastest. Popcorn incorporates some of these implementations as modules: it uses the XPIR library for CPIR and borrows the CGKS ITPIR [31] 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. We see three main limitations. First, because Popcorn's overheads grow linearly with the number of objects, it has no hope of scaling to YouTube-size libraries. Second, organizations that serve objects can collude to compromise Popcorn's privacy guarantee. Admittedly, an assumption of no collusion may be unrealistic against state-level adversaries that can compromise multiple organizations (or already have). 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. To support online updates, Popcorn should execute both CPIR and ITPIR queries on the same version of the key and object libraries, at the key server and 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: maintaining the utility of batching when running on a distributed infrastructure, and increased hardware provisioning at the CDN's edge servers. Though addressing the latter is non-trivial, we think that it does not require a paradigm shift: Akamai's EdgeComputing service [34] already 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 think 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 9200 requests/90min/PoP[12]) and are growing fast [6]. This is not specific to Netflix: similar request rates (average of 6000 requests in 90 minutes from within a single city) have been reported for other video on demand systems [111].

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. *Adaptive streaming* lets clients switch between different video quality levels to adjust to bandwidth fluctuations. Popcorn could support this feature in two ways. First, it could maintain an individual library for each quality level. Clients would send queries to all libraries but download a video chunk only from the appropriate one. (A concern is, does switching between libraries leak information? No, because the chunk download pattern and switches are "lined up" with a reference object, O_{avg} (§4.4).) This solution is simple, but asking each library to process every request would increase server-side work significantly.

Alternatively, Popcorn could exploit layered coding [59, 66, 89, 95] or multiple description coding (MDC) [54, 92, 107]. There would be a single basic quality library accessed by all clients, with separate libraries for enhancement layers (better spatial resolution, bitrate, frame rate, etc.). The server-side work would thus be 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. The current prototype can support both subscription-based and pay-per-view pricing models, by monitoring accesses to the key server. Furthermore, by default it works with a prepaid royalty model, where the distributor pays a fixed license fee up front. 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 think that these limitations are not fundamental, as prior works [16, 25, 57, 103] have addressed them in different contexts. Future work is to investigate the performance and privacy implications of composing these works with Popcorn.

Targeted ads and recommendation services. Popcorn does not currently support targeted advertisements or recommendations. Incorporating relevant prior work [19, 24, 26, 55, 65, 68] into Popcorn is a direction for future work.

[12]Assumes 10 billion hours watched in 3 months [5], 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 8 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:

$$\begin{aligned} 32C + 60M + 6.4I &= 0.6281 \\ 32C + 244M + 6.4I &= 0.9822 \\ 32C + 244M + 23.3I &= 1.6902, \end{aligned}$$

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 Carlos Aguilar-Melchor, Allen Clement, Alan Dunn, Bryan Ford, Yuval Ishai, Lon Ingram, Jaeyeon Jung, Brad Karp, Marc-Olivier Killijian, Sangman Kim, Michael Z. Lee, James Mickens, Simon Peter, Thomas Schneider, Vitaly Shmatikov, Emmett Witchel, and the anonymous reviewers 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; an AWS in Education Research grant; and a Google European Doctoral Fellowship.

References

[1] Alphabetical List - Fri, Apr 3, 2015. `http://usa.netflixable.com/2015/04/alphabetical-list-fri-apr-3-2015.html`.

[2] Digital Rights Management. `http://msdn.microsoft.com/en-us/library/cc838192%28VS.95%29.aspx`.

[3] Free haven's selected papers in anonymity. `http://freehaven.net/anonbib/`.

[4] Microsoft PlayReady. `http://www.microsoft.com/playready/`.

[5] Netflix 2015 Q1 Earnings Letter. `http://files.shareholder.com/downloads/NFLX/47469957x0x821407/DB785B50-90FE-44DA-9F5B-37DBF0DCD0E1/Q1_15_Earnings_Letter_final_tables.pdf`.

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

[7] New Movie Arrivals - Fri, Apr 3, 2015. `http://usa.netflixable.com/2015/04/new-movie-arrivals-fri-apr-3-2015.html`.

[8] Pyramid broadcast technique for video on demand. Lecture notes, `http://www.mathcs.emory.edu/~cheung/Courses/558-old/Syllabus/5-VoD/pyramid.html`.

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

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

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

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

[19] 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.

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

[21] 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.

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

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

[24] M. Burkhart and X. A. Dimitropoulos. Fast privacy-preserving top-k queries using secret sharing. In *International Conference on Computer Communication Networks (ICCCN)*, 2010.

[25] 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.

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

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

[28] 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.

[29] 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.

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

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

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

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

[34] 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.

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

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

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

[38] 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.

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

[40] 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.

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

[42] 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.

[43] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph,

A. Ganjam, J. Zhan, and H. Zhang. Understanding the impact of video quality on user engagement. *ACM SIGCOMM Computer Communication Review*, 41(4):362–373, 2011.

[44] 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.

[45] 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.

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

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

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

[49] 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.

[50] 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.

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

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

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

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

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

[56] 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.

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

[58] G. B. Horn, P. Knudsgaard, S. B. Lassen, M. Luby, and J. E. Rasmussen. A scalable and reliable paradigm for media on demand. *IEEE Computer*, 34(9):40–45, 2001.

[59] 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.

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

[61] 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.

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

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

[64] 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.

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

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

[67] 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.

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

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

[70] A. Kwon, D. Lazar, S. Devadas, and B. Ford. Riffle: An efficient communication system with strong anonymity. In *Privacy Enhancing Technologies Symposium (PETS)*, 2016.

[71] 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.

[72] 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.

[73] 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.

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

[75] 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.

[76] 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.

[77] 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.

[78] 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.

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

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

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

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

[83] 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.

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

[85] 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.

[86] 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.

[87] 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.

[88] 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.

[89] 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.

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

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

[92] 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.

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

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

[95] 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.

[96] 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`.

[97] 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.

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

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

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

[101] J. P. Stern. A new and efficient all-or-nothing disclosure of secrets protocol. In *International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, 1998.

[102] 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.

[103] 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.

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

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

[106] 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.

[107] 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.

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

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

[110] 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.

[111] 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.