Private Blocklist Lookups with Checklist

Dmitry Kogan
Stanford University

Henry Corrigan-Gibbs
MIT CSAIL

Abstract. This paper presents Checklist, a system for private blocklist lookups. In Checklist, a client can determine whether a particular string appears on a server-held blocklist of strings, without leaking its string to the server. Checklist is the first blocklist-lookup system that (1) leaks no information about the client's string to the server, (2) does not require the client to store the blocklist in its entirety, and (3) allows the server to respond to the client's query in time *sublinear* in the blocklist size. To make this possible, we construct a new two-server private-information-retrieval protocol that is both asymptotically and concretely faster, in terms of server-side time, than those of prior work. We evaluate Checklist in the context of Google's "Safe Browsing" blocklist, which all major browsers use to prevent web clients from visiting malware-hosting URLs. Today, lookups to this blocklist leak partial hashes of a subset of clients' visited URLs to Google's servers. We have modified Firefox to perform Safe-Browsing blocklist lookups via Checklist servers, which eliminates the leakage of partial URL hashes from the Firefox client to the blocklist servers. This privacy gain comes at the cost of increasing communication by a factor of 3.3×, and the server-side compute costs by 9.8×. Checklist reduces end-to-end server-side costs by 6.7×, compared to what would be possible with prior state-of-the-art two-server private information retrieval.

1 Introduction

This paper proposes a new system for *private blocklist lookups*. In this setting, there is a client, which holds a private bitstring, and a server, which holds a set of blocklisted strings. The client wants to determine whether its string is on the server's blocklist, without revealing its string to the server.

This blocklist-lookup problem arises often in computer systems:

- Web browsers check public-key certificates against blocklists of revoked certificates [55, 60, 61].
- Users of Google's Password Checkup and the "Have I Been Pwned?" service check their passwords against a blocklist of breached credentials [52, 62, 65, 84, 86].
- Antivirus tools check the hashes of executed binaries against blocklists of malicious software [26, 58, 67].
- Browsers and mail clients check URLs against Google's Safe Browsing blocklist of phishing sites [9, 38, 43].

A simple approach for blocklist lookups is to store the blocklist on the server and have the client send its query string to the server. However, the string that the client is checking against the blocklist is often *private*: the client wants to hide from the server which websites she is visiting, or which passwords she is using, or which programs she is running.

Another approach is to store the entire blocklist on the client [28]. Maintaining a client-side blocklist offers maximal client privacy, since the server learns nothing about which strings the client checks against the blocklist. The downside is that the client must download and store the entire blocklist—consuming scarce client-side bandwidth and storage. Chrome uses this approach for its certificate-revocation blocklist [60]. Unfortunately, client-side resource constraints limit the size of client-side blocklists: Chrome's revocation blocklist (as of May 2021) covers under 1,200 revoked certificates out of millions of revoked certificates on the web [53], and thus provides suboptimal protection against revoked certificates.

A hybrid approach is also possible: the client stores a *compressed* version of the blocklist, which allows the client to perform most blocklist lookups locally, at a modest storage cost. The compressed blocklist, like a Bloom filter [11], can return false-positive answers to blocklist queries. When the local blocklist gives a positive answer, the client queries the server to check whether a local positive is a true positive. The Safe Browsing API client uses this technique. The limitation of this strategy is that the client's queries to the server still leak some information about the client's private string. In particular, the Safe Browsing client's queries to the server allow the server to make a good guess at which URL the client is visiting [9, 38, 45, 72].

Existing techniques for private blocklist lookups are inadequate. Keeping the blocklist on the client in its entirety is infeasible when the blocklist is large. Querying a server-side blocklist leaks sensitive client data to the server.

This paper presents the design and implementation of Checklist, a new privacy-respecting blocklist-lookup system. Using Checklist is less expensive, in terms of total communication, than maintaining a client-side blocklist. And, unlike conventional server-side blocklists, Checklist leaks *nothing* about the client's blocklist queries to the system's servers. We achieve this privacy property using a new high-throughput form of two-server private information retrieval. Checklist requires only a modest amount of server-side computation:

in a blocklist of n entries, the amortized server-side cost is $O(\sqrt{n})$ work per query. Concretely, a server can answer client queries to the three-million-entry Safe Browsing blocklist in under half a core-millisecond per query on average. Our new PIR scheme reduces the server-side compute costs by 6.7×, compared with a private-blocklist scheme based on existing PIR protocols.

To our knowledge, Checklist is the first blocklist-lookup system that (1) leaks no information about the client's string to the server, (2) does not require the client to store the blocklist in its entirety, and (3) achieves per-query server-side computation that is *sublinear* in the blocklist size.

At the heart of Checklist is a new "offline/online" private-information-retrieval scheme [12, 27, 71]. These schemes use client-specific preprocessing in an offline phase to reduce the computation required at query (online) time. On a blocklist with n entries and with security parameter $\lambda \approx 128$, our scheme requires the servers to perform work $O(\sqrt{n})$ per query, on average. This improves the $O(\lambda\sqrt{n})$ per-query cost of schemes from prior work [27] and amounts to a roughly 128-fold concrete speedup. In addition, prior offline/online schemes do not perform well when the blocklist/database changes often (since even a single-entry change to the blocklist requires rerunning the preprocessing step). By carefully structuring the blocklist into a cascade of smaller blocklists, we demonstrate that it is possible to reap the benefits of these fast offline/online private-information-retrieval schemes even when the blocklist contents change often. In particular, in a blocklist of n entries, our scheme requires server-side computation $O(\log n)$ per blocklist update per client, whereas a straightforward use of offline/online private-information-retrieval schemes would yield $\Omega(n)$ time per update per client.

Limitations. First, since Checklist builds on a two-server private-information-retrieval scheme, it requires two independent servers to maintain replicas of the blocklist. The system protects client privacy as long as *at least one* of these two servers is honest (the other may deviate arbitrarily from the prescribed protocol). In practice, two major certification authorities could run the servers for certificate-revocation blocklists. Google and Mozilla could run the servers for the Safe-Browsing blocklist. An OS vendor and antivirus vendor, such as Microsoft and Symantec, could each run a server for malware blocklists. Second, while Checklist reduces *server-side* CPU costs, compared with a system built on the most communication-efficient prior two-server PIR scheme [15] (e.g., by 6.7× when used for Safe Browsing), Checklist increases the *client-to-server* communication (by 2.7×) relative to a system based on this earlier PIR scheme. In applications in which client resources are extremely scarce, Checklist may not be appropriate. But for applications in which server-side costs are important, Checklist will dominate. We discuss these and other deployment considerations in Section 8.

Experimental results. We implemented our private blocklist-lookup system in 2,481 lines of Go and 497 lines of C. In addition, we configure the Firefox web browser to use our private blocklist-lookup system to query the Safe Browsing blocklist. (By default Firefox makes Safe-Browsing blocklist queries to the server via the Safe Browsing v4 API, which leaks a 32-bit hash of a subset of visited URLs to Google's servers.) Under a real browsing workload, our private-blocklisting system requires 9.4× more servers than a non-private baseline with the same latency and increases total communication cost by 3.3×. We thus show that it is possible to eliminate a major private risk in the Safe Browsing API at a manageable cost.

Contributions. The contributions of this paper are:

- A new two-server offline/online private-information-retrieval protocol that reduces the servers' computation by a factor of the security parameter $\lambda \approx 128$.
- A general technique for efficiently supporting database updates in private-information-retrieval schemes that use database-specific preprocessing.
- A blocklist-lookup system that uses these new private-information-retrieval techniques to protect client privacy.
- An open-source implementation and experimental validation of Checklist applied to the Safe Browsing API. (Our code is available on GitHub [1].)

2 Goals and overview

2.1 Problem statement

In the private-blocklist-lookup problem, there is a *client* and one or more blocklist *servers*. The blocklist $\mathcal{B}$ is a set of strings, of which each server has a copy. We assume, without loss of generality, that the strings in the blocklist are all of some common length ℓ (e.g., 256 bits). If the strings are longer or shorter, we can always hash them to 256 bits using a collision-resistant hash function, such as SHA256.

Initially, the client may download some information about the blocklist from the servers. Later on, the client would like to *lookup* strings in the blocklist: the client holds a string $X \in \{0,1\}^\ell$ and, after interaction with the servers, the client should learn whether or not the string X is on the servers' blocklist (i.e., whether $X \in \mathcal{B}$). In addition, the servers may add and remove strings from the blocklist over time. We do *not* attempt to hide the blocklist from the client, though it is possible to do so using an extension described in Section 8.2.

The goals of such a system, stated informally, are:

- **Correctness.** Provided that the client and servers correctly execute the prescribed protocol, the client should receive correct answers to its blocklist queries, except with some negligible failure probability.
- **Privacy.** In our setting, there are two blocklist servers and as long as one of these servers executes the protocol faithfully, an adversary controlling the network and the other blocklist server learns nothing about the queries

that the client makes to the blocklist (apart from the total number of queries).

Formally, the adversarial server should be able to simulate its view of its interaction with the client and the honest server, given only the system's public parameters and the number of queries that the client makes.

Efficiency. In our setting, the two key efficiency metrics are:

- *Server-side computation*: The amount of computation that the servers need to perform per query.
- *Total communication*: The number of bits of communication between the client and blocklist servers.

Since clients typically make many queries to the same blocklist, we consider both of these costs as amortized over many queries and many blocklist updates (additions and removals).

Using a client-side blocklist minimizes server-side computation, but requires communication linear in the number of blocklist updates. Using standard private-information-retrieval protocols [15, 24, 39, 59] minimizes communication but requires per-client server-side computation linear in the blocklist size. Checklist minimizes the server-side computation without the client having to download and store the entire blocklist.

2.2 Design overview

Checklist consists of two main layers: the first layer allows private lookups to *static array-like* databases. The second layer adds support for dynamic dictionaries that allow private *key-value* lookups and efficient *updates*. We now explain the design of each layer.

Private lookups. A straightforward way to implement private lookups is to use private information retrieval (PIR) [15,23,24]. With standard PIR schemes, the running time of the server on each lookup is linear in the blocklist size n. In contrast, recent "offline/online" PIR schemes [27] reduce the server's online computational cost to $\lambda\sqrt{n}$, after the client runs a linear-time preprocessing phase with the server. During this preprocessing phase, the client downloads a *compressed representation* of the blocklist. These offline/online PIR schemes are well suited to our setting: the client and server can run the (relatively expensive) preprocessing step when the client first joins Checklist. Thereafter, the server can answer private blocklist queries from the client in time sublinear in the blocklist length—much faster than conventional PIR.

To instantiate this paradigm, we construct in Section 4 a new offline/online PIR scheme that achieves a roughly 128-fold speedup over the state of the art, in terms of server-side computation. (Asymptotically, our new scheme reduces the servers' online time to roughly $\sqrt{n}$ from $\lambda\sqrt{n}$, where $\lambda \approx 128$ is the security parameter.)

As with many PIR schemes, our protocol requires two servers, and it protects client privacy as long as at least one server is honest.

Dynamic dictionaries. Offline/online PIR schemes allow the server to answer client queries at a low cost *after* the client and server have run a relatively expensive preprocessing phase. One hitch in using these schemes in practice is that the client and server have to rerun the expensive preprocessing step whenever the server-side blocklist (database) changes. If the blocklist changes often, then the client and server will have to rerun the preprocessing phase frequently. The recurring cost of the preprocessing phase may then negate any savings that an offline/online PIR scheme would afford.

The second layer of our system, described in detail in Section 5, reaps the efficiency benefits of offline/online PIR schemes, even in a setting in which the blocklist changes frequently. Our high-level approach, which follows a classic idea from the data-structures literature [10] and its applications in cryptography [20, 41, 70, 79, 83], is to divide the length-n blocklist into $O(\log n)$ buckets, where the bth bucket contains at most 2^b entries. The efficiency gains come from the fact that only the contents of the small buckets, for which preprocessing is inexpensive, change often. The large buckets, for which preprocessing is costly, change rarely. We use preexisting techniques [23] to support key-value lookups to the database, rather than array-like lookups.

With this strategy, the amortized cost per blocklist update is $O(\log n)$. In contrast, a naïve application of offline/online PIR would lead to $\Omega(n)$ amortized cost per update.

3 Background

This section summarizes the relevant background on private information retrieval.

Notation. For a natural number n, the notation $[n]$ refers to the set $\{1, 2, \ldots, n\}$. All logarithms are base 2. We ignore integrality concerns and treat expressions like $\sqrt{n}$, $\log n$, and m/n as integers. The expression $\mathrm{negl}(\cdot)$ refers to a function whose inverse grows faster than any fixed polynomial. For a finite set S, the notation $r \xleftarrow{R} S$ refers to choosing r independently and uniformly at random from the set S. For $p \in [0, 1]$, the notation $b \xleftarrow{R} \mathsf{Bernoulli}(p)$ refers to choosing the bit b to be "1" with probability p and "0" with probability $1 - p$. For a bit $b \in \{0, 1\}$, we use $\bar{b}$ to denote the bit $1 - b$. For two equal-length bit strings $X, Y \in \{0, 1\}^\ell$, we use $X \oplus Y \in \{0, 1\}^\ell$ to refer to their bitwise XOR.

3.1 Private information retrieval (PIR)

In a private information retrieval (PIR) system [24, 25], a set of servers holds identical copies of an n-row database. The client wants to fetch the ith row of the database, without leaking the index i of its desired row to the servers. We work in the *two-server setting*, in which the client interacts with two database replicas. The system protects the client's privacy as long the adversary controls at most one of the two servers.

In traditional PIR schemes, the servers must take a linear scan over the entire database in the process of answering each client query. In the standard setting of PIR, in which the

servers store the database in its original form, this linear-time server-side computation is inherent [7].

Offline/online PIR. This linear-time cost on the servers is a performance bottleneck, so recent work [12, 27, 30, 71] constructs "offline/online" PIR schemes, which move the servers' linear-time computation to an offline preprocessing phase. Offline/online PIR schemes work in two phases:

- **In the offline phase**, which takes place before the client decides which database row it wants to fetch, the client downloads a *hint* from one of the PIR servers. The hint's size is sublinear in the size of the full database, though generating it still takes the server time that is at least linear in the size of the database.
- **In the online phase**, which takes place once the client decides which database row it wants to fetch, the client uses its hint to issue a query to the PIR servers. The servers' responses to the queries allow the client to reconstruct the database row it is interested in, as well as to update its hint in preparation for future queries. The total communication and the server's work in this step are sublinear in the database size.

There are two benefits to using offline/online PIR schemes:

1. *Lower latency.* The amount of online computation that the servers need to perform to service a client query is sublinear in the database size, compared with linear for standard PIR schemes. This lower online cost can translate into lower perceived latency for the client.
2. *Lower total amortized cost.* Since each client can reuse a single hint for making many online queries, the servers' work per query is also sublinear in the database size, compared with linear for standard PIR schemes.

3.2 Puncturable pseudorandom set

To construct our PIR schemes, we will use *puncturable pseudorandom sets* [27, 78], for which there are simple constructions from any pseudorandom generator (e.g., AES-CTR).

Informally, a puncturable pseudorandom set gives a way to describe a pseudorandom size-$\sqrt{n}$ subset $S \subset \{1, \ldots, n\}$ using a short cryptographic key sk. (The set size is a tunable parameter, but in this paper we always take the subset size to be $\sqrt{n}$.) Furthermore, it is possible to "puncture" the key sk at any element $i \in S$ to get a key sk_p that is a concise description of the set $S' = S \setminus \{i\}$. The important property of the punctured key is that it hides the punctured element, in a strong cryptographic sense. That is, given only sk_p, an adversary cannot guess which was the punctured element with better probability than randomly guessing an element from $[n] \setminus S'$. This notion of puncturing comes directly from the literature on puncturable pseudorandom functions [13, 16, 51, 57, 76].

The full syntax and definitions appear in prior work [27], but we recall the important ideas here. More formally, a punctured pseudorandom set consists of the following algorithms, where we leave the security parameter implicit:

- $\mathsf{Gen}(n) \to \mathsf{sk}$. Generate a random puncturable set key sk.
- $\mathsf{GenWith}(n, i) \to \mathsf{sk}$. Given an element $i \in [n]$, generate a random puncturable set key sk such that the element $i \in \mathsf{Eval}(\mathsf{sk})$.
- $\mathsf{Eval}(\mathsf{sk}) \to S$. Given an unpunctured key sk, output a pseudorandom set $S \subseteq [n]$ of size $\sqrt{n}$. (Or, given a punctured key sk_p, output a pseudorandom set of size $\sqrt{n} - 1$.)
- $\mathsf{Punc}(\mathsf{sk}, i) \to \mathsf{sk}_\mathsf{p}$. Given a set key sk and element $i \in \mathsf{Eval}(\mathsf{sk})$, output a punctured set key sk_p such that $\mathsf{Eval}(\mathsf{sk}_\mathsf{p}) = \mathsf{Eval}(\mathsf{sk}) \setminus \{i\}$.

We include the formal correctness and security definitions for puncturable pseudorandom sets in Appendix A.

Constructions. Prior work [27] constructs puncturable sets from any pseudorandom generator $G\colon \{0,1\}^\lambda \to \{0,1\}^{2\lambda}$ (e.g., AES in counter mode) such that: (a) the set keys are λ bits long and (b) the punctured set keys are $O(\lambda \log n)$ bits long. Furthermore, the computation cost of Eval consists almost entirely of $O(\sqrt{n})$ invocations of the PRG.

4 PIR with faster online time

In this section, we describe our new two-server offline/online PIR protocol. Throughout this section, we view the database as a static array; in Section 5 we handle the case of a key-value database that changes over time.

Compared with the best prior two-server scheme [27], ours improves the servers' online time and the online communication by a multiplicative factor of the security parameter λ. Since we typically take $\lambda \approx 128$ in practice, this improvement gives roughly a 128-fold improvement in communication and online computation cost.

Specifically, on a database of n records, of length ℓ bits each, and security parameter λ, existing PIR schemes have online communication $O(\lambda^2 \log n + \lambda\ell)$ and online server time $O(\lambda\ell\sqrt{n})$, measured in terms of main-memory reads and evaluations of a length-doubling PRG. We bring the online communication cost down to $O(\lambda \log n + \ell)$ bits and the online server-side computation time down to $O(\ell\sqrt{n})$ operations (dominated by the cost of $O(\sqrt{n})$ AES operations and $O(\sqrt{n})$ random-access ℓ-bit database lookups). Concretely, these costs are modest—less than a kilobyte of communication and under 150 microseconds, even for blocklists with millions of entries.

In terms of the preprocessing phase, similarly to previous work [27], our protocol uses $\lambda\ell\sqrt{n}$ bits of communication and requires the server to do $O(\lambda\ell n)$ work per client.

4.1 Definition

A two-server offline/online PIR scheme for a database $\mathcal{D} = (D_1, \ldots, D_n)$ of n records, of length ℓ bits each, consists of the following four algorithms. We leave the security parameter implicit and, here and henceforth, assume that n and ℓ are bounded by some polynomial in the security parameter.

$\mathsf{Hint}(\mathcal{D}) \to h$. The first database server uses the Hint algorithm to generate a preprocessed data structure h that a client can later use to privately query the database $\mathcal{D}$. The Hint algorithm is randomized, and the first server must run this algorithm once per client.

$\mathsf{Query}(h, i) \to (\mathsf{st}, q_0, q_1)$. The client uses the Query algorithm to generate the PIR queries it makes to the database servers. The algorithm takes as input the hint h and the database index $i \in [n]$ that the client wants to read. The algorithm outputs client state st and PIR queries q_0 and q_1, one for each server.

$\mathsf{Answer}(\mathcal{D}, q) \to a$. The servers uses Answer, on database $\mathcal{D}$ and client query q to produce an answer a.

$\mathsf{Reconstruct}(\mathsf{st}, a_0, a_1) \to (h', D_i)$. The client uses state st, generated at query time, and the servers' answers a_0 and a_1 to produce a new hint h' and the database record $D_i \in \{0,1\}^\ell$.

We sketch the correctness and privacy definitions here for the case in which the client makes a single query. Prior work gives the (lengthy) definitions for the multi-query setting [27].

Correctness. If an honest client interacts with honest servers, the client recovers its desired database record. We say that an offline/online PIR scheme is *correct* if, for all databases $\mathcal{D} = (D_1, \ldots, D_n)$ and all $i \in [n]$, the probability

$$\Pr\left[D_i' = D_i : \begin{array}{rl} h & \leftarrow \mathsf{Hint}(\mathcal{D}) \\ (\mathsf{st}, q_0, q_1) & \leftarrow \mathsf{Query}(h, i) \\ a_0 & \leftarrow \mathsf{Answer}(\mathcal{D}, q_0) \\ a_1 & \leftarrow \mathsf{Answer}(\mathcal{D}, q_1) \\ (\_, D_i') & \leftarrow \mathsf{Reconstruct}(\mathsf{st}, a_0, a_1) \end{array} \right]$$

is at least $1 - \mathrm{negl}(\lambda)$, on (implicit) security parameter λ.

Security. An attacker who controls either one of the two servers learns nothing about which database record the client is querying, even if the attacker deviates arbitrarily from the prescribed protocol. More formally, for a database $\mathcal{D} = (D_1, \ldots, D_n)$ and $i \in [n]$, define the probability distributions

$$\mathsf{View}_{\mathcal{D},0,i} := \left\{ h, q_0 : \begin{array}{rl} h & \leftarrow \mathsf{Hint}(\mathcal{D}) \\ (\_, q_0, \_) & \leftarrow \mathsf{Query}(h, i) \end{array} \right\},$$

capturing the "view" of the first server, and

$$\mathsf{View}_{\mathcal{D},1,i} := \left\{ q_1 : \begin{array}{rl} h & \leftarrow \mathsf{Hint}(\mathcal{D}) \\ (\_, \_, q_1) & \leftarrow \mathsf{Query}(h, i) \end{array} \right\},$$

capturing the "view" of the second server.

An offline/online PIR scheme is *secure* if, for every number of records n, record length ℓ, database $\mathcal{D}$, servers $s \in \{0,1\}$, and $i, j \in [n]$ the distributions $\mathsf{View}_{\mathcal{D},s,i}$ and $\mathsf{View}_{\mathcal{D},s,j}$ are computationally indistinguishable. This definition implicitly captures security against an adversarial server that colludes with additional clients in the system, since the adversary can simulate the responses of the honest server to such clients.

4.2 Our scheme

Prior offline/online PIR schemes [27] natively have relatively large correctness error: the online phase fails with relatively large probability $\approx 1/\sqrt{n}$. To allow the client to recover its record of interest with overwhelming probability, the client and server must run the online-phase protocol λ times in parallel to drive the correctness error down to $(1/\sqrt{n})^\lambda = \mathrm{negl}(\lambda)$. Our improved PIR scheme (Construction 1) is slightly more complicated than those of prior work, but the benefit is that it has very low (i.e., cryptographically negligible) correctness error. Since our protocol has almost no correctness error, the parties need not repeat the protocol λ times in parallel, which yields our λ-fold performance gain.

Our main result of this section is:

Theorem. *Construction 1 is a computationally secure offline/online PIR scheme, assuming the security of the underlying puncturable pseudorandom set. On a database of $n \in \mathbb{N}$ records, each of length ℓ bits, and security parameter $\lambda \in \mathbb{N}$ (used to instantiate the puncturable pseudorandom set), the scheme has:*

- *offline communication $\lambda(\ell\sqrt{n} + 1)$ bits,*
- *offline time $O(\lambda \ell n)$,*
- *client query time $O(n)$ in expectation,*
- *online communication $2(\lambda + 1)\log n + 4\ell$ bits, and*
- *online server time $O(\ell\sqrt{n})$.*

We formally analyze the correctness and security of Construction 1 in Appendix B. Here, we describe the intuition behind how the construction works.

Offline phase. In the offline phase of the protocol, the first server samples $T = \lambda\sqrt{n}$ puncturable pseudorandom set keys $(\mathsf{sk}_1, \ldots, \mathsf{sk}_T)$. Then, for each $t \in [T]$, the server computes the parity of the database records indexed by the set $\mathsf{Eval}(\mathsf{sk}_t)$. If the database consists of n records $D_1, \ldots, D_n \in \{0,1\}^\ell$, then the t-th parity word is: $P_t = \bigoplus_{j \in \mathsf{Eval}(\mathsf{sk}_t)} D_j \in \{0,1\}^\ell$. The t keys $(\mathsf{sk}_1, \ldots, \mathsf{sk}_T)$ along with the T parity words $(P_1, \ldots, P_T)$ form the hint that the server sends to the client. If the server uses a pseudorandom generator seeded with seed to generate the randomness for the T invocations of Gen, the hint that the client stores consists of $(\mathsf{seed}, P_1, \ldots, P_T)$ and has length $\lambda + \lambda\ell\sqrt{n}$ bits.

The key property of this hint is that with overwhelming probability (at least $1 - 2^{-\lambda}$), each database record will be included in at least one of the parity words. That is, for every $i \in [n]$, there exists a $t \in [T]$ such that $i \in \mathsf{Eval}(\mathsf{sk}_t)$.

Online phase. In the online phase, the client has decided that it wants to fetch the ith record of the database, for $i \in [n]$.

The client's general strategy will be to obtain the parity words of the database records indexed by sets of indices S and $S \setminus \{i\}$. The client will then recover the value of the database record from the two parity words.

Our scheme uses two instantiations of this strategy. In the first case, the client will take the set S and its parity word P

Construction 1 (Our offline/online PIR scheme)**.** Parameters: database size $n \in \mathbb{N}$, record length $\ell \in \mathbb{N}$, security parameter $\lambda \in \mathbb{N}$, $T := \lambda\sqrt{n}$, puncturable pseudorandom set (Gen, GenWith, Eval, Punc) construction of Section 3.2 with universe size n and set size $\sqrt{n}$.

$\underline{\mathsf{Hint}(\mathcal{D}) \to h.}$

- For $t \in [T]$:
  - Sample a puncturable-set key $\mathsf{sk}_t \leftarrow \mathsf{Gen}(n)$.
    // To reduce the hint size, we can sample the
    // randomness for the T invocations of Gen *from a*
    // pseudorandom generator, whose seed we include
    // in the hint.
  - Set $S_t \leftarrow \mathsf{Eval}(\mathsf{sk}_t)$.
  - Compute the parity word $P_t \in \{0,1\}^\ell$ of the database records indexed by the set S_t.
    That is, let $P_t \leftarrow \bigoplus_{j \in S_t} D_j$.
- Output the hint as: $h \leftarrow \big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (P_1, \ldots, P_T)\big)$.

$\underline{\mathsf{Query}(h, i) \to (\mathsf{st}, q_0, q_1).}$

- Sample bit $\beta \xleftarrow{R} \mathsf{Bernoulli}(2(\sqrt{n}-1)/n)$.
- If $\beta = 0$: $(\mathsf{st}', q_0, q_1) \leftarrow \mathsf{QueryCommon}(h, i)$.
- If $\beta = 1$: $(\mathsf{st}', q_0, q_1) \leftarrow \mathsf{QueryRare}(i)$.
- Set $\mathsf{st} \leftarrow (h, \beta, \mathsf{st}')$
- Return (st, q_0, q_1)

$\underline{\mathsf{Answer}(\mathcal{D}, q) \to a.}$

- Parse the query q as a pair $(\mathsf{sk}_\mathsf{p}, r)$, where sk_p is a punctured set key and $r \in [n]$.
- Compute $S_\mathsf{p} \leftarrow \mathsf{Eval}(\mathsf{sk}_\mathsf{p})$ and compute the parity word $W \in \{0,1\}^\ell$ of the database records indexed by this set: $W \leftarrow \bigoplus_{j \in S_\mathsf{p}} D_j$.
- Return $a \leftarrow (W, D_r) \in \{0,1\}^{2\ell}$ to the client.

$\underline{\mathsf{Reconstruct}(\mathsf{st}, a_0, a_1) \to (h', D_i).}$

- Parse the state st as (h, β, st').
- Parse the answers as (W_0, V_0) and (W_1, V_1).
- If $\beta = 0$: *// Common case*
  - Parse the hint h as $\big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (P_1, \ldots, P_T)\big)$.
  - Parse the state st' as $(t, \mathsf{sk}_\mathsf{new})$
  - Set $D_i \leftarrow P_t \oplus W_1$.

  // The client updates the t-th component of the hint.
  - Set $\mathsf{sk}_t \leftarrow \mathsf{sk}_\mathsf{new}$ and $P_t \leftarrow W_0 \oplus D_i$.
  - Set $h' \leftarrow \big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (P_1, \ldots, P_T)\big)$.
- If $\beta = 1$: *// Rare case*
  - Parse the state st' as $\gamma \in \{0,1\}$
  - Set $D_i \leftarrow W_0 \oplus W_1 \oplus V_\gamma$.
  - Set $h' \leftarrow h$. *// The hint is unmodified.*
- Return (h', D_i).

$\underline{\mathsf{QueryCommon}(h, i) \to (\mathsf{st}', q_0, q_1).}$

// The client finds a set S_t in the hint that contains index i.
// The client asks the second server for the parity of the
// database records in $S_t \setminus \{i\}$.
// The client asks the first server for the parity of $\sqrt{n}-1$
// records indexed by a freshly sampled random set.
// The client also asks each server for the value of one extra
// database record.

- Parse the hint h as $\big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (P_1, \ldots, P_T)\big)$.
- Let $t \in [T]$ be a value such that $i \in \mathsf{Eval}(\mathsf{sk}_t)$.
  (If no such value t exists, abort.)
- Sample $\mathsf{sk}_\mathsf{new} \leftarrow \mathsf{GenWith}(n, i)$.
- Compute:

$$\begin{aligned} S_\mathsf{new} &\leftarrow \mathsf{Eval}(\mathsf{sk}_\mathsf{new}) & S_t &\leftarrow \mathsf{Eval}(\mathsf{sk}_t) \\ r_0 &\xleftarrow{R} S_\mathsf{new} \setminus \{i\} & r_1 &\xleftarrow{R} S_t \setminus \{i\} \\ \mathsf{sk}_{\mathsf{p}0} &\leftarrow \mathsf{Punc}(\mathsf{sk}_\mathsf{new}, i) & \mathsf{sk}_{\mathsf{p}1} &\leftarrow \mathsf{Punc}(\mathsf{sk}_t, i) \\ q_0 &\leftarrow (\mathsf{sk}_{\mathsf{p}0}, r_0) & q_1 &\leftarrow (\mathsf{sk}_{\mathsf{p}1}, r_1). \end{aligned}$$

- Set $\mathsf{st}' \leftarrow (t, \mathsf{sk}_\mathsf{new})$.
- Return (st', q_0, q_1).

$\underline{\mathsf{QueryRare}(i) \to (\mathsf{st}', q_0, q_1).}$

// The client asks each server for the parity of the database
// records indexed by a freshly sampled random set of $\sqrt{n}-1$
// indices such that the symmetric difference between the
// two sets contains i and one other random index r_γ.
// The client also asks server γ for the record at index r_γ.

- Sample a random bit $\gamma \xleftarrow{R} \{0,1\}$.
- Sample $\mathsf{sk}_\mathsf{new} \leftarrow \mathsf{GenWith}(n, i)$.
- Compute:

$$\begin{aligned} S_\mathsf{new} &\leftarrow \mathsf{Eval}(\mathsf{sk}_\mathsf{new}) \\ r_\gamma &\xleftarrow{R} S_\mathsf{new} \setminus \{i\} & r_{\bar\gamma} &\xleftarrow{R} S_\mathsf{new} \setminus \{r_\gamma\} \\ \mathsf{sk}_{\mathsf{p}\gamma} &\leftarrow \mathsf{Punc}(\mathsf{sk}_\mathsf{new}, i) & \mathsf{sk}_{\mathsf{p}\bar\gamma} &\leftarrow \mathsf{Punc}(\mathsf{sk}_\mathsf{new}, r_\gamma) \\ q_\gamma &\leftarrow (\mathsf{sk}_{\mathsf{p}\gamma}, r_\gamma) & q_{\bar\gamma} &\leftarrow (\mathsf{sk}_{\mathsf{p}\bar\gamma}, r_{\bar\gamma}). \end{aligned}$$

- Set $\mathsf{st}' \leftarrow \gamma$.
- Return (st', q_0, q_1).

from the stored hint. In the second case, the client will choose S to be a fresh random set that contains i. The client chooses between the instantiations at random each time it wants to fetch a record from the database. We set the probability of each case such that the overall probability distribution of the client's queries hides the indices the client is interested in.

We now describe this in more detail.

Common case. Recall that at the start of the offline phase, the client holds the hint it received in the offline phase, which consists of a seed for a pseudorandom generator and a set of T hint words $(P_1, \ldots, P_T)$. The client's first task is to expand the seed into a set of puncturable pseudorandom set keys $\mathsf{sk}_1, \ldots, \mathsf{sk}_T$. (These are the same keys that the server generated in the offline phase.) Next the client searches for a key $\mathsf{sk}_t \in \{\mathsf{sk}_1, \ldots, \mathsf{sk}_T\}$ such that the index of the client's desired record $i \in \mathsf{Eval}(\mathsf{sk}_t)$.

At this point, the client holds a set $S_t = \mathsf{Eval}(\mathsf{sk}_t)$ of size $\sqrt{n}$, which contains the client's desired index i. The client also holds the parity word $P_t \in \{0,1\}^\ell$ of the database records indexed by S_t. The client sends the set $S_t \setminus \{i\}$ to the second server. (To save communication, the client compresses this set using puncturable pseudorandom sets.) The server returns the parity word W_1 of the database records indexed by this set $S_t \setminus \{i\}$. The client recovers its record of interest as:

$$P_t \oplus W_1 = \left(\bigoplus_{j \in S_t} D_j\right) \oplus \left(\bigoplus_{j \in S_t \setminus \{i\}} D_j\right) = D_i.$$

For security, it is critical that each server "sees" each set only once. Therefore, the client must not reuse the set S_t for any future queries. Therefore, the client also samples a replacement set S_{new} of $\sqrt{n}$ indices in $[n]$, one of which is i. The client then sends $S_{\mathsf{new}} \setminus \{i\}$ to the *first* server (again, compressed using puncturable pseudorandom sets), and the first server responds with the parity word W_0 of the database records indexed by this set. The client then replaces the set S_t in its hint with the new set S_{new} and updates the corresponding parity hint word to $P_{\mathsf{new}} \leftarrow W_0 \oplus D_i$.

In this first case, the sets that the client sends to the two servers *never* contain the index i of the client's desired database record. If the client would always use this query strategy, the servers would learn which database records the client is definitely *not* querying, effectively leaking $\approx 1/(\sqrt{n} \ln 2)$ bits of information about i. The next case prevents this leakage.

Rare case. With a small probability (roughly $2/\sqrt{n}$), the client must send a set containing its desired index i to each server. The client samples a random set S_{new} of $\sqrt{n}$ values in $[n]$, one of which is i. The client chooses a server $\gamma \xleftarrow{R} \{0,1\}$ at random and sends it $S_{\mathsf{new}} \setminus \{i\}$ (again, compressed), along with the index of a random element $r_\gamma \xleftarrow{R} S_{\mathsf{new}} \setminus \{i\}$. To the other server $\bar{\gamma} \coloneqq 1 - \gamma$, the client sends $S_{\mathsf{new}} \setminus \{r_\gamma\}$ and, to hide which server plays which role, a dummy value $r_{\bar{\gamma}}$.

Each server replies with the parity word W of the database records indexed by the set it has received. It also sends the value of the database record D_r. Now, the client can recover its record of interest as: $D_i = W_0 \oplus W_1 \oplus D_{r_\gamma}$, since $\forall \gamma \in \{0,1\}$, this sum is equal to

$$\left(\bigoplus_{j \in S_{\mathsf{new}} \setminus \{i\}} D_j\right) \oplus \left(\bigoplus_{j \in S_{\mathsf{new}} \setminus \{r_\gamma\}} D_j\right) \oplus D_{r_\gamma} = D_i.$$

To hide whether the client is in the "common case" or "rare case," the client sends dummy indices r_0, r_1 to the servers in the common case to mimic its behavior in the rare case.

Remark (Pipelined queries). When a client makes many PIR queries in sequence, it may want to issue a new query to the servers before receiving the servers' response to its previous query. Our scheme (Construction 1) allows the client to have any number of queries in flight at once, while still using only a single hint. The key observation is that the client can generate the replacement set $\mathsf{sk}_{\mathsf{new}}$ as soon as it issues a query. The client can thus issue a second query immediately after issuing the first, and a third query immediately after issuing the second—the client just has to receive the server's responses in the order in which it issued its queries.

Remark. The client's expected online query time in our construction is linear in the size of the database, since the client has to expand its set keys one by one in a random order, until it finds a key of a set that contains the index of interest i. As in prior offline/online PIR schemes [27], a client can use a data structure to reduce the query time at the cost of increasing its storage. Checklist uses a simple data structure that has size linear in the database size n but that supports constant-time queries. That is, the client stores a hash table mapping database indices $i \in [n]$ to "set pointers" $j \in [\lambda\sqrt{n}]$ such that $i \in \mathsf{Eval}(\mathsf{sk}_j)$. The client lazily populates this map whenever it evaluates set keys and invalidates entries whenever it discards set keys. As a compromise between storage and query time, the map contains at most one set pointer for each database index. Therefore, discarding a set may leave some database indices without valid set pointers, even though other sets in the client's hint may still contain those indices. At query time, if the client fails to find a set pointer for the desired database index in the map, it falls back to exhaustively searching through the hint. As it iterates through the hint, the client "opportunistically" adds set pointers to the map.

5 Offline/online PIR for dynamic dictionaries

PIR protocols typically treat the database as a static array of n records. To fetch a record, a PIR client must then specify the index $i \in [n]$ of the record. Our scheme of Section 4 follows this approach as well. In contrast, Checklist, like many other applications of PIR, needs to support dynamic databases and key-value-style lookups. Specifically, we would like to view the database as a list of key-value pairs $((K_1, V_1), \ldots, (K_n, V_n))$, where $K_i \in \{0,1\}^k$ are the keys, and $V_i \in \{0,1\}^\ell$ are their corresponding values. In Checklist, (i) a client should be able

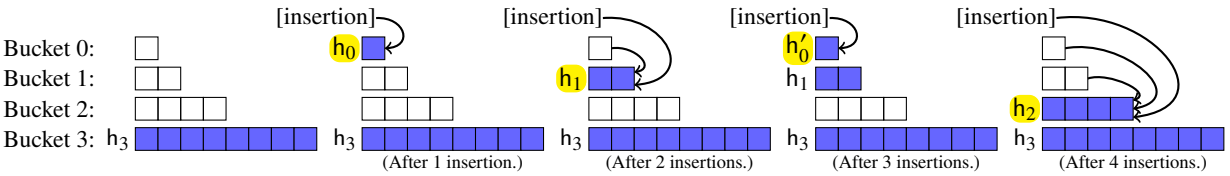


Figure 1: The database in our PIR scheme consists of many buckets, where the ith bucket can hold 2^i database rows. The client holds a hint (h_i) corresponding to each non-empty bucket i. The smaller buckets change frequently, but these hints are inexpensive to recompute. The larger buckets change infrequently, and these hints are expensive to recompute.

to look up a value V by its key K; and (ii) a server should be able to insert, modify, and delete key-value pairs.

5.1 Existing tool: PIR by keywords

Previous work has shown how to modify standard PIR schemes to support key-value-style databases. Specifically, Chor, Gilboa, and Naor [23] showed that it is possible to construct so-called "PIR-by-keywords" schemes from traditional PIR-by-index schemes in a black-box way. Modern PIR constructions [15] support PIR-by-keywords directly. The cost of such schemes, both in communication and server-side computation, matches the cost of standard PIR, up to low-order terms. The black-box PIR-by-keywords techniques [23] directly apply to offline/online PIR schemes as well. Specifically, our implementation of Checklist uses a simple PIR-by-keywords technique, which is tailored at the preexisting design of the Safe Browsing system. We describe this scheme in Section 6.3.

5.2 Handling changes with waterfall updates

Standard online-only PIR schemes do not need any special machinery to meet handle database updates, since their clients hold no state that depends on the database contents. The servers in online-only PIR schemes can thus simply process any changes to the database locally as they happen, and then answer each query using the latest version of the database. In contrast, clients in offline/online PIR schemes hold preprocessed "hints" about the database, and every change in the database invalidates these hints.

The simple solution works poorly. The simplest way to handle database updates is to have the servers compute a new hint relative to the latest database state after every update. The servers then send this updated hint to the client. The problem is that if the rate of updates is relatively high, the cost of regenerating these hints will be prohibitive.

Specifically, if the database changes at roughly the same rate as the client makes queries (e.g., once per hour), the client will have to download a new hint before making each query. In this case, the server-side costs of generating these hints will be so large as to negate the benefit of using an offline/online PIR scheme in the first place.

Our approach: Waterfall updates. Instead of paying the hint-generation cost for the full database on each change, we design a tiered update scheme, which is much more efficient. Specifically, if there is a single database update between each pair of client queries, the asymptotic online cost of our scheme is still $O(\sqrt{n})$—the same cost as if the database had not changed. As the frequency of updates increases, the performance of our scheme gracefully degrades. Our design builds on a classic idea for converting static data structures into dynamic structures [10]. Cryptographic constructions using this idea to handle data updates include oblivious RAMs [41], proofs of retrievability [20, 79], searchable encryption [83], and accumulators [70].

Our strategy is to have the servers store the database as an array of $B = \log n$ sub-databases, which we call "buckets." (Here, we assume for simplicity that the number of records n is a power of two.) The bth bucket will contain at most 2^b key-value pairs. In addition, the servers maintain a last-modified timestamp for each bucket. Initially, the servers store the entire database in the bottom (biggest) bucket, and all other buckets start out empty. As the database changes, the contents of the buckets change as well.

When a client joins the system, it fetches a hint for each bucket. Before making a query, the client updates its locally stored hints. To do this, the client sends to the first server the timestamp τ at which it received its last hint. The server then generates a fresh hint for each bucket that was modified after τ, and sends these new hints back to the client. To find the value associated with key K, the client then queries each of the B buckets in parallel for key K. If several buckets contain key K, the client uses the value V from the smallest bucket (i.e., the bucket that was updated most recently).

Since the underlying offline/online PIR-by-keywords scheme supports only static databases, each time a bucket changes, the server must regenerate from scratch a hint for this bucket for every client. The key to achieving our cost savings is that, as the database changes, the contents of the smallest buckets will change frequently, but it is relatively inexpensive for the servers to regenerate the hints for these buckets. The contents of the larger buckets—for which hint generation is expensive—will change relatively infrequently.

It remains to describe how the servers update the contents of the buckets upon database changes. Let us first consider database *insertions*. When the servers want to add a new pair (K, V) to the database, the servers insert that pair into the topmost (smallest) bucket. Such an update can cause a

bucket b to "fill up"—to contain more than 2^b entries. When this happens, the servers "flush" the contents of bucket b down to bucket $b+1$. If this flush causes bucket $b+1$ to fill up, the servers continue flushing buckets until all buckets are below their maximum capacity. If the bottommost bucket overflows, the servers create a new bucket, twice the size of the previous one. The two servers execute this process in lockstep to ensure that their views of the database state remain consistent throughout.

To *remap* an existing key K to a new value V', the servers add the updated record (K, V') to the topmost bucket. When, as a result of flushing, multiple pairs with the same key end up in the same bucket, the server keeps only the latest pair and discards any earlier pairs.

To *delete* an existing key K, the servers add a pair $(K, V_\perp)$ to the topmost bucket, where $V_\perp \in \{0,1\}^\ell$ is some special value. (If the set of possible values for a key is $\{0,1\}^\ell$ in its entirety, we can extend the bit-length of the value space by a single bit.) When, as a result of flushing, $(K, V_\perp)$ ends up in the same bucket as other pairs with the same key K, the servers only keep the latest pair and discard any earlier pairs. At the bottom-most bucket, the servers can discard all remaining $(K, V_\perp)$ pairs.

Analysis. The client needs a new hint for bucket b only each time all of the buckets $\{1, \ldots, b-1\}$ overflow. When this happens, the servers flush $1 + \sum_{i=1}^{b-1} 2^i = 2^b$ elements into bucket b. Intuitively, if the server generates a new hint after each update, then after u updates, the server has generated $u/2^b$ hints for bucket b, each of which takes time roughly $\lambda \ell 2^b$ to generate, on security parameter λ. (This is because our offline/online scheme has hint-generation time $\lambda \ell n$, on security parameter λ and a database of n ℓ-bit records.)

The total hint generation time with this waterfall scheme after u updates, on security parameter λ, with $B = \log n$ buckets, is then at most $\lambda u \ell B = \lambda u \ell \log n$. In contrast, if we generate a hint for the entire database on each change using the simple scheme, the total hint generation time is $\lambda u \ell n = \lambda u \ell 2^B$ (since $n = 2^B$). That is, the waterfall scheme gives an *exponential improvement* in server-side hint-generation time over the simple scheme.

The query time of this scheme is $\sum_{b=1}^{B} O(\ell \cdot \sqrt{2^b}) = O(\ell\sqrt{n})$. So, we achieve an exponential improvement in hint-generation cost at a modest (less than fourfold) increase in online query time.

One subtlety is that in our base offline/online PIR scheme, the length of a hint for a bucket of size 2^b is roughly $\lambda \ell \sqrt{2^b}$. For buckets smaller than λ^2, using offline-online PIR would require more communication than just downloading the contents of the entire bucket. We thus use a traditional "online-only" PIR scheme for those small buckets.

6 Use case: Safe Browsing

Every major web browser today, including Chrome, Firefox, and Safari, uses Google's "Safe Browsing" service to warn users before they visit potentially "unsafe" URLs. In this context, unsafe URLs include those that Google suspects are hosting malware, phishing pages, or other social-engineering content. If the user of a Safe-Browsing-enabled browser tries to visit an unsafe URL, the browser displays a warning page and may even prevent the user from viewing the page.

6.1 How Safe Browsing works today

At the most basic level, the Safe Browsing service maintains a blocklist of unsafe URL prefixes. The browser checks each URL it visits against this blocklist before rendering the page to the client. Since the blocklist contains URL prefixes, Google can add an entire portion of a site to the blocklist by adding just the appropriate prefix. (In reality, there are multiple Safe Browsing blocklists, separated by the type of threat, but that detail is not important for our discussion.)

Two factors complicate the implementation:

- **The blocklist is too large for clients to download and store.** The Safe Browsing blocklist contains roughly three million URL prefixes. Even sending a 256-bit hash of each blocklisted URL prefix would increase a browser's download size and storage footprint by more than 90MB. This would more than *double* the download size of Firefox on Android [68].
- **The browser cannot make a network request for every blocklist lookup.** For every webpage load, the browser must check every page resource (image, JavaScript file, etc.) against the Safe Browsing blocklist. If the browser made a call to the Safe Browsing API over the network for every blocklist lookup, the latency of page loads, as well as the load on Google's servers, would be tremendous.

The current Safe Browsing system (API v4) [43] addresses both of these problems using a two-step blocklisting strategy.

Step 1: Check URLs against an approximate local blocklist. Google ships to each Safe Browsing client a data structure that represents an *approximate and compressed* version of the Safe Browsing blocklist, similar to a Bloom filter [11, 18]. Before the browser renders a web resource, it checks the corresponding URL against its local compressed blocklist. This local blocklist data structure has no false negatives (it will always correctly identify unsafe URLs) but it has false positives (sometimes it will flag safe URLs as unsafe). In other words, when given a URL, the local blocklist either replies "definitely safe" or "possibly unsafe." Thus, whenever the local blocklist identifies a URL as safe, the browser can immediately render the web resource without further checks.

In practice, this local data structure is a list of 32-bit hashes of each blocklisted URL prefix. Delta-encoding the set [42] further reduces its size to less than 5MB—

roughly 18× smaller than the list of all 256-bit hashes of the blocklisted URL prefixes. The browser checks a URL (e.g., `http://a.b.c/1/2.html?param=1`) by splitting it into substrings (`a.b.c/1/2.html?param=1`, `a.b.c/1/2.html`, `a.b.c./1`, `a.b.c/`, `b.c/`, etc.), hashing each of them, and checking each hash against the local blocklist.

Step 2: Eliminate false positives using an API call. Whenever the browser encounters a possibly unsafe URL, as determined by its local blocklist, the browser makes a call to the Safe Browsing API over the network to determine whether the possibly unsafe URL is truly unsafe or whether it was a false positive in the browser's local blocklist.

To execute this check, the browser identifies the 32-bit hash in its local blocklist that matches the hash of the URL. The browser then queries the Safe Browsing API for the full 256-bit hash corresponding to this 32-bit hash.

Finally, the browser hashes the URL in question down to 256 bits and checks whether this full hash matches the one that the Safe Browsing API returned. If the hashes match, then the browser flags the URL as unsafe. Otherwise, the browser renders the URL as safe.

This two-step blocklisting strategy is useful for two reasons. First, it requires much less client storage and bandwidth, compared to downloading and storing the full blocklist locally. Second, it adds no network traffic in the common case. The client only queries the Safe Browsing API when there is a false positive, which happens with probability roughly $n/2^{32} \approx 2^{-11}$. So, only one in every 2,000 or so blocklist lookups requires making an API call.

However, as we discuss next, the current Safe Browsing architecture ***leaks information about the user's browsing history to the Safe Browsing servers***.

6.2 Safe Browsing privacy failure

Prior work [9, 38, 45, 72] has observed that the Safe Browsing protocol leaks information about the user's browsing history to the servers that run the Safe Browsing API—that is, to Google. In particular, whenever the user visits a URL that is on the Safe Browsing blocklist, the user's browser sends a 32-bit hash of this URL to Google's Safe Browsing API endpoint. Since Google knows which unsafe URLs correspond to which 32-bit hashes, Google then can conclude with good probability which potentially unsafe URL a user was visiting. (To provide some masking for the client's query, Firefox mixes the client's true query with queries for four random 32-bit hashes. Still, the server can easily make an educated guess at which URL triggered the client's query.)

There is some chance (a roughly one in 2,000) that a user queries the Safe Browsing API due to a false positive—when the 32-bit hash of a safe URL collides with the 32-bit hash of an unsafe URL. Even in this case, Google can identify a small list of candidate safe URLs that the user could have been browsing to cause the false positive.

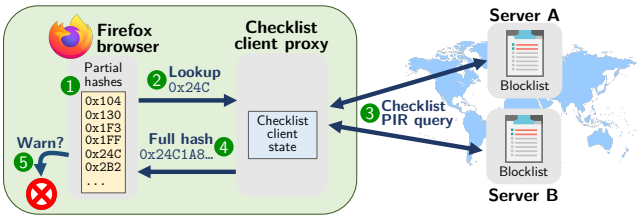


Figure 2: Using Checklist for Safe Browsing. ❶ The browser checks whether the URL's partial hash appears in its local blocklist. ❷ If so, the browser issues a Safe Browsing API query for the full hash corresponding to the matching partial hash. ❸ The Checklist client proxy issues a PIR query for the full hash to the two Checklist servers. ❹ The Checklist client proxy returns the full hash of the blocklisted URL to the browser. ❺ The browser warns the user if the URL hash matches the hash of the blocklisted URL.

6.3 Private Safe Browsing with Checklist

We design a private Safe-Browsing service based on Checklist, which uses our new PIR scheme of Section 4. Our scheme requires two non-colluding entities (e.g., CloudFlare and Google) to host copies of the blocklist, but it has the privacy benefit of not revealing the client's query to either server.

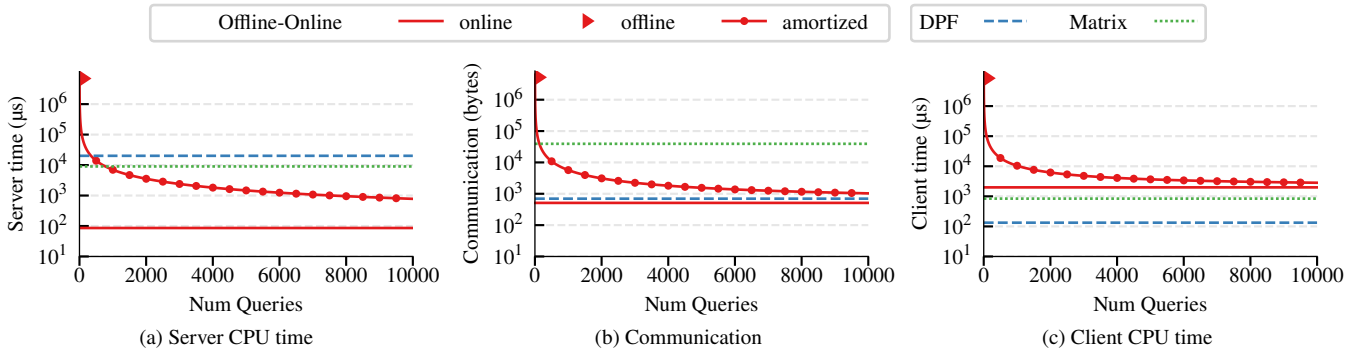
Our Checklist-based Safe Browsing client works largely the same as today's Safe Browsing client does (Figure 2). The only difference is that when the client makes an online Safe Browsing API call (to check whether a hit on the client's local compressed blocklist is a false positive), the client uses our PIR scheme to perform the lookup. In this way, the client can check URLs against the Safe Browsing blocklist without revealing any information about its URLs to the server (beyond the fact that the client is querying the server on some URL).

When the client visits a URL whose 32-bit hash appears in the client's local blocklist, the client needs to fetch the full 256-bit SHA256 hash of the blocked URL from the Safe Browsing servers. To do this, the client identifies the index $i \in [n]$ of the entry in its local blocklist that caused the hit. (Here n is the total number of entries in the local blocklist.) The client then executes the PIR protocol of Section 4 with the Safe Browsing servers to recover the ith 256-bit URL hash. If the full hash from the servers matches the full hash of the client's URL, the browser flags the webpage as suspect. If not, it is a false positive and the browser renders the page.

As the Safe Browsing blocklist changes, the client can fetch updates to its local blocklist using the method of Section 5.2.

When two or more full hashes in the blocklist have the same 32-bit prefix, the Checklist servers can lengthen the partial hashes for the colliding entries. This way, a partial hash on the client's local list always maps to a single full hash on the servers' blocklist. Safe Browsing already supports variable-length partial hashes.

Partial hashes as PIR-by-keywords. The client's local list of partial hashes essentially serves as a replacement for using a general PIR-by-keywords transformation [23]. The downside of this replacement is that it uses offline communication that



(a) Server CPU time (b) Communication (c) Client CPU time

Figure 3: For a static database of three million 32-byte records, we show the query cost in server time, client time, and communication. The figure also shows the offline cost of the new offline-online PIR scheme and its total cost (offline and online), amortized over a varying number of queries. The offline phase of the new scheme is expensive but its per-query server-side time is lower than in prior PIR schemes.

is linear in the number of records in the database. In Safe Browsing, the primary purpose of the local list is to reduce latency, bandwidth, and server computation, by allowing the browser to respond to most queries locally. Checklist takes advantage of the existence of this local list, additionally using it to map partial hashes to their positions in the blocklist. In principle, both for Safe Browsing and for other applications, Checklist could use other PIR-by-keywords techniques or a local blocklist of a different size, allowing different tradeoffs between storage, communication, and latency.

Remaining privacy leakage. Checklist prevents the Safe Browsing server from learning the partial hash of the URL that the client is visiting. However, the fact that the client makes a query to the server at all leaks some information to the server: the server learns that the client visited *some* URL whose partial hash appears on the blocklist. While this minimal leakage is inherent to the two-part design of the Safe Browsing API, it may be possible to ameliorate even this risk using structured noise queries [34].

7 Implementation and evaluation

We implement Checklist in 2,481 lines of Go and 497 lines of C. (Our code is available on GitHub [1].) We use C for the most performance-sensitive portions, including the puncturable pseudorandom set (Section 3.2). We discuss low-level optimizations in Appendix C.

7.1 Microbenchmarks for offline-online PIR

First, we evaluate the computational and communication costs of the new offline-online PIR protocol, compared to two previous PIR schemes. One is an information-theoretic protocol of Chor et al. [25] ("Matrix PIR"), which uses $\sqrt{n}$ bits of communication on an n-bit database. The second comparison protocol is that of Boyle, Gilboa, and Ishai [15], based on distributed point functions ("DPF"). This protocol requires only $O(\log n)$ communication and uses only symmetric-key cryptographic primitives. We use the optimized DPF code of Kales [56]. We run our benchmarks on a single-machine single-threaded setup, running on a e2-standard-4 Google Compute Engine machine (4 vCPUs, 16 GB memory).

Static database. We begin with evaluating performance on a static database. Figure 3 presents the servers' and client's per-query CPU time and communication costs on a database of three million 32-byte records. Since the Checklist PIR scheme has both offline and per-query costs, the figure also presents the amortized per-query cost as a function of the number of queries to the static database made by the same client following an initial offline phase. Figure 3 shows that the offline-online PIR scheme reduces the server's *online* computation time by 100× at the cost of an expensive seven-second *offline* phase, which the server runs once per client. Even with this high offline cost, for a sufficiently large number of queries, the Checklist PIR scheme provides *overall* computational savings for the server. For example, after 1,500 queries, the total computational work of a server using Checklist PIR is two to four times less than that of a server using the previous PIR schemes. The Checklist PIR scheme is relatively expensive in terms of client computation—up to 20× higher compared to the previous PIR schemes.

Database with periodic updates. We evaluate the performance of the waterfall updates mechanism (Section 5.2). This experiment starts with a database consisting of three million 32-byte records. We then apply a sequence of 200 updates to the database, where each update modifies 1% of the database records. After each update, we compute the cost for the server of generating an updated hint for the client. Figure 4 shows the cost of this sequence of updates. The majority of the updates require very little server computation, as they trigger an update of only the smallest bucket in the waterfall scheme. We also plot the average update cost (dashed line) in the waterfall scheme and the cost of naively regenerating the hint from scratch on each update (red square). The waterfall scheme reduces the average cost by more than 45×.

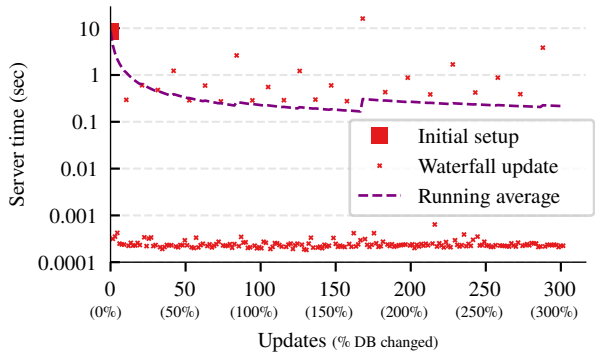


Figure 4: Server-side cost of client updates. At each time step, 1% of the three million records change. The waterfall update scheme reduces the average update cost by more than 45× relative to a naive solution of rerunning the offline phase on each change.

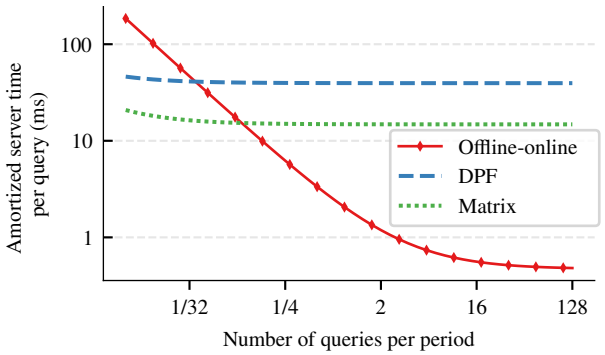


Figure 5: The amortized server compute costs of PIR queries on a database with updates. As the number of queries between each pair of subsequent database updates grows, the offline-online PIR scheme achieves lower compute costs compared to previous PIR schemes.

Next, we evaluate the impact of using the waterfall update scheme on the query costs. This experiment begins with a database of $n = 3 \times 10^6$ records, of size 32 bytes each, and runs through a sequence of periods. At the beginning of each period, we apply a batch of $B = 500$ updates to the database, after which the client fetches a hint update from the server, and then performs a sequence of queries. We measure the cost to the server of generating the update and responding to the queries. We amortize the cost of each update over the queries in each period, and we average the costs over n/B consecutive periods, thus essentially evaluating the long-term amortized costs of the scheme. Figure 5 presents the amortized server costs as a function of the number of queries made by a single client in each period. The new PIR scheme outperforms the previous schemes as long as the client makes a query at least every 10 periods (i.e., at least once every 5000 database changes). As queries become more frequent, the reduced online time of our scheme outweighs its costly hint updates.

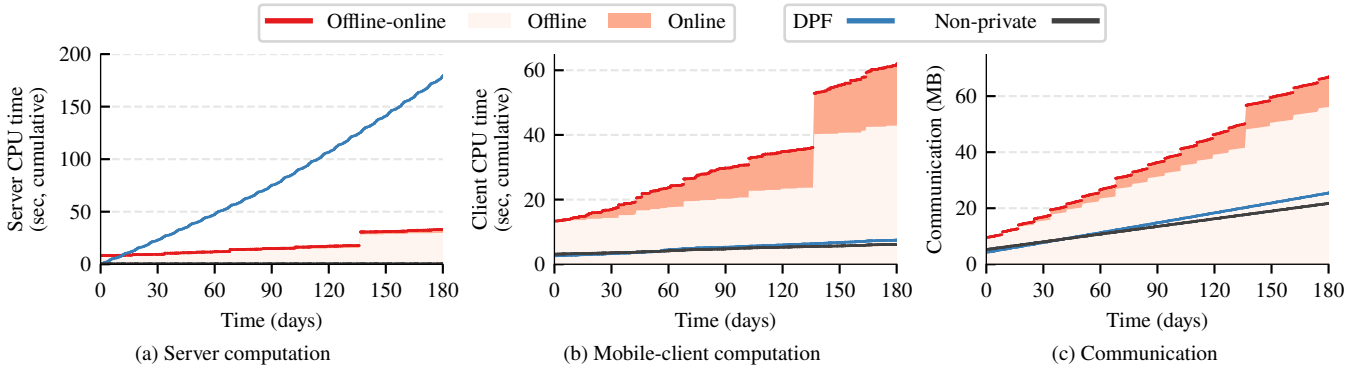
7.2 Safe Browsing with Checklist

To evaluate the feasibility of using Checklist for Safe Browsing, we integrate Checklist with Firefox's Safe Browsing mechanism. We avoid the need to change the core browser code by building a *proxy* that implements the standard Safe Browsing API. The proxy runs locally on the same machine as the browser, and we redirect all of the browser's Safe Browsing requests to the proxy by changing the Safe Browsing server URL in Firefox configuration. See Figure 2.

We begin by measuring the rate of updates to the Safe Browsing database and the pattern of queries generated in the course of typical web browsing. To this end, our proxy logs all Safe-Browsing requests, forwards them to Google's server, and logs the responses. (For privacy, we do not log the actual URL hashes.) This trace allows us to directly compute the frequency of lookups. Moreover, the fact that the browser continuously downloads updates to the list of partial hashes allows us to compute the rate of updates to the database. We run the proxy on our personal laptops for a typical work week, using the instrumented browser for all browsing. The database size is roughly three million records, and it has grown by about 30,000 records over the course of the week. These data are consistent with the public statistics that Google used to publish on the Safe Browsing datasets [44]. In our trace, the client updates its local state every 94 minutes on average and performs an online lookup every 44 minutes on average.

We repeatedly replay our recorded one-week trace to simulate long-term usage of Checklist. On each update request in the trace, we first use the information from the response to update the size of the Checklist database, such that the database size evolves as in the recording. We measure the cost of fulfilling the same update request using Checklist, which includes updating the list of partial hashes and updating the client's PIR hint. For each lookup query in the trace, we issue a PIR query. Figure 6 shows the cumulative costs of using Checklist with two different PIR schemes. We measure the server costs on an e2-standard-4 Google Compute Engine machine with 16 GB of memory and the client costs on a Pixel 5 mobile phone. Offline/online PIR requires 5.5× less computation on the server and 9× more computation on the client than DPF-based PIR. In absolute terms, the amortized computation on the client when using Checklist with offline/online PIR is less than 0.4 CPU-seconds per day. Offline-online PIR uses more communication, mostly due to the cost of maintaining the hint: it doubles the communication cost of the initial setup, and requires 2.7× more communication than DPF-based PIR on a running basis. Checklist with DPF-based PIR uses only 20% more communication than non-private Safe Browsing.

We also measure the amount of local storage a Checklist client requires for its persistent state. With DPF-based PIR, or with non-private lookups, the client stores a 4-byte partial hash for each database record. Delta-encoding the list of hashes [42]) further reduces the storage to fewer than 1.5 bytes per record (for a list of 3 million partial hashes). With offline-online PIR, the Checklist client stores on average 6.8 bytes for each 32-byte database record, in order to store the list of partial hashes and the latest hint. To reduce the query



(a) Server computation (b) Mobile-client computation (c) Communication

Figure 6: We repeatedly replay the trace of Safe Browsing queries and updates, recorded on a seven-day user session. The server-side computational saving of offline-online PIR comes at a cost of more communication and client computation. The measurements are of the application-level costs of Checklist and do not include the computation and communication cost of the network stack. The client-side computation cost of Checklist is less than 0.4 CPU-seconds per day. Discontinuities happen when buckets in the waterfall scheme overflow and trigger a hint update for a larger bucket.

time, the client also stores an 18-bit "set pointer" from each database index to a set that contains it, as described at the end of Section 4. The total storage cost for a list of 3 million partial hashes is 25MB. As a point of reference, the download size of the the Firefox Android package is 70MB [68], and after installation, it uses 170MB of storage.

To measure the end-to-end throughput and latency of Checklist, we set up three virtual cloud instances: a Checklist server, a Checklist client, and a load generator. The load generator simulates concurrent virtual Checklist users, by producing a stream of requests to the server, each through a new TLS connection. The generator sets the relative frequency of update and query requests, as well as the size of the updates, based on the recorded trace. With the server under load, an additional client machine performs discrete Checklist lookups and measures their end-to-end latency. The measured latency includes the time it takes the client to generate the two queries, obtain the responses from the server, and process the responses. We compare between (i) Checklist running the new offline-online PIR protocol, (ii) Checklist running the DPF-based protocol, and (ii) Checklist doing non-private lookups. Figure 7 shows that the throughput of a single Checklist server providing private lookups using offline-online PIR is 9.4× smaller (at a similar latency) than that of a server providing non-private lookups. A Checklist server achieves 6.7× higher throughput and a 30ms lower latency when using offline-online PIR, compared to when running DPF-based PIR.

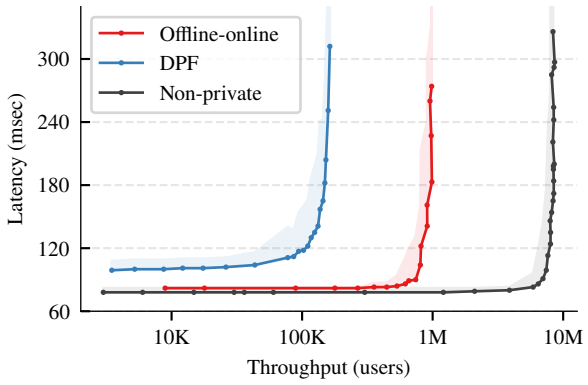


Figure 7: The performance of a Checklist server. Solid lines display the average latency, and shaded regions show the latency of the 95th-percentile of requests.

Table 8 summarizes our evaluation of Safe Browsing with Checklist. We estimate that a private Safe Browsing service using Checklist with offline-online PIR would require 9.4× more servers than a non-private service with similar latency. A DPF-based PIR protocol would require 6.7× more servers than our offline-online protocol and would increase the latency by 30ms, though it would use 10× less client computation and 2.7× less bandwidth on a running basis.

8 Discussion

8.1 Deployment considerations

When is Checklist cost effective for Safe Browsing? Table 8 shows three different ways to achieve full privacy for Safe Browsing queries: having the client maintain a full client-side blocklist ("Full list"), using Checklist with a standard PIR scheme ("DPF"), and using Checklist with our new offline/online PIR scheme ("Offline-Online"). Which of these three schemes will be best in practice depends on the relative costs of server-side computation, client-side computation, communication, and client storage.

Download full list. When communication and client storage are relatively inexpensive, as on a powerful workstation with a hard-wired network connection, the best Safe Browsing solution may be to have the client keep a local copy of the entire blocklist. Downloading the full list would require roughly

Table 8: Summary of costs of Safe Browsing with Checklist. For each column, we use green, yellow, and red, to indicate the least-, middle-, and most-expensive solution. The offline-online variant offers lower compute costs and latency, while a DPF-based system is more communication efficient. The second row presents the communication costs of a fully offline solution in which the client maintains a local copy of the blocklist.

Approach	**Server costs**	**Latency**	**Client computation**		**Communication**		**Client storage**	
			Initial	Running	Initial	Running	Initial	Running
	(servers per 1B users)	(ms)	(sec)	(sec/month)	(MB)	(MB/month)	(MB)	(MB/month)
Non-private	143	91	3.1	0.5	5.0	3.0	4.3	0.2
Full list	*Very small – not measured*				91.8	13.2	91.8	4.5
Checklist with offline-online PIR (§4)	1348	90	13.3	8.0	10.3	9.8	24.5	1.6
Checklist with DPF PIR [15]	9047	122	2.6	0.8	5.0	3.6	4.3	0.2

9× more communication initially and 3.7× more storage than Checklist with offline-online PIR, but the reduction in server-side computational cost would be significant.

Checklist with offline-online PIR. When trying to jointly minimize communication and server-side computation, Checklist with our new offline-online PIR scheme is the most appealing approach. This point in the trade-off space may be useful for general devices (laptops, etc.) in which it is reasonable to shift some work to the client for the benefit of decreased server cost. The total communication is lower than downloading the full blocklist and the server-side computation is roughly 7× less than would be required when using standard PIR.

Checklist with DPF PIR. Finally, when trying to minimize client computation and storage, Checklist with DPF-based PIR may be the best option. This configuration may be useful on mobile devices, where client resources are especially scarce. This approach requires the least storage (22× less than storing the full blocklist and 5.7× less than Checklist with offline-online PIR), at the cost of increased server-side computation.

As Table 8 shows, there is not yet one private-blocklisting scheme that dominates the others in all dimensions. Identifying the optimal point in this trade-off space requires measuring the relative costs of the various computational resources.

Denial-of-service attacks. The initial hint-generation phase of our scheme is relatively expensive—it requires 7.3 seconds of server-side computation per client. If a single client could ask the Safe Browsing servers to rerun the offline hint-generation phase as frequently as the client wanted, a single client could easily exhaust server resources, denying service to honest clients. We envision at least two approaches to preventing this type of denial-of-service attack: First, in some settings, clients have long-term identities, such as when Google Chrome users are logged into the browser with their Google accounts. In this case, the Safe Browsing server can limit the number of offline requests each client makes. (If the client exceeds this limit, the servers could force it back to making non-private queries.) Alternatively, the servers could use a proof-of-work puzzle [6, 33] to force the clients to do at least as much work as the servers do. This approach is wasteful, both in energy and in that it doubles the total time of the offline phase. Nevertheless, since an honest client only requests a new full hint very infrequently—whenever it installs the browser for the first time—requiring several seconds of client CPU time on initial hint generation seems feasible.

Synchronizing state. A Checklist deployment requires two non-colluding entities to run the two Checklist servers. For an Internet-scale deployment, we would implement each logical Checklist server on hundreds or thousands of physical replica servers, distributed around the world. As the blocklist database changes, the replicas will need to download the database updates from the main server.

When the Checklist client fetches its hint in the offline phase, the server includes a timestamp τ (as in Section 5.2) indicating the database version that this hint is for. When the client makes an online query later on, it sends this timestamp τ along with its query. If the server's database is newer than τ, the client and the server run the update process described in Section 5.2. If the server's database is older than τ, then the server is out of date and the client must retry its query at another replica.

Clients only update their Safe Browsing data a few times an hour (at most), so the main Safe Browsing server needs to push updates to the replica servers only a few times per hour as well. Since each update involves exchanging at most a few megabytes of data with each replica, we expect it to be relatively easy to keep a distributed fleet of replicas up to date.

8.2 Extensions

Privacy for the server. We focus on protecting the privacy of the client's blocklist query but we do not attempt to hide the full blocklist from the client. In many applications, such as password-breach notification services [52, 62, 65, 84, 86], hiding the blocklist from the client is important. That is, at each interaction with the server the client should only learn whether its string appears on the blocklist.

Freedman, Ishai, Pinkas, and Reingold [36] show that it is possible to lift a PIR scheme like ours, with privacy for the client only, into a PIR scheme with privacy for the client *and* servers using oblivious pseudorandom functions.

Their transformation is elegant and concretely efficient. It makes black-box use of the underlying PIR and just requires minimal extra server-side work and no additional rounds of communication between the client and the server. While we have not yet implemented this extension, since server-side privacy is not crucial for us, we expect it to be a simple and useful extension for other applications of Checklist.

Batching. In some applications of Checklist, a client may want to query the blocklist on many strings at once. In this case, the client and servers can use batch PIR schemes to improve performance [4, 49, 54]. These schemes can reduce the problem of making $t \gg 1$ PIR queries to a database of size n to the problem of making roughly t queries (ignoring log factors) to a database of size n/t. When applied to our offline/online PIR schemes with online time $\sqrt{n}$, the online time is $t\sqrt{n/t} = \sqrt{tn}$, instead of the $t\sqrt{n}$ cost of t-fold repetition. Since the Safe Browsing client only rarely makes multiple PIR queries at once, we have not implemented this extension.

8.3 Future work

Single-server setting. Checklist requires two servers to maintain replicas of the blocklist, and client privacy holds against adversaries that control at most one server. In practice, it can be difficult to deploy multi-party protocols at scale, since it requires coordination between multiple (possibly competing) companies or organizations. An important direction for future work would be to extend our offline/online PIR scheme to work in the single-server setting [59], taking advantage of recent advances in lattice-based PIR schemes [2, 3, 4, 5].

Prior work [27] shows that it is possible in theory to construct single-server offline/online PIR schemes with sublinear online server time. Those schemes have two limitations that would make them unsuitable in practice: they make extensive use of expensive homomorphic-encryption schemes and they do not allow the client to reuse its hint over multiple queries. The latter property means that the total amortized server cost per query is at least n on a database of size n, whereas the amortized server-side cost of our scheme is roughly $\sqrt{n}$. An important task for future work would be to design single-server offline/online PIR schemes with modest concrete costs that allow a client to reuse a single hint for multiple online queries.

Weakening the trust requirements. We present a two-server offline/online PIR scheme that protects client privacy against a single malicious server. It would be much better if, for any $k > 1$, we could construct a k-server offline/online PIR scheme with sublinear online time that protects client privacy against a coalition of $k - 1$ malicious servers.

While no such PIR scheme exists, to our knowledge, we sketch one possible approach to constructing one here. Prior work [27] constructs a *single-server* offline/online PIR scheme with sublinear online time. In the offline phase of that scheme, the client sends an encryption of a vector to the server, using an additively homomorphic encryption scheme, and the server applies a linear operation to the client's query. We can execute the same protocol in the k-server setting, by replacing the additively homomorphic encryption with a k-out-of-k linear secret-sharing scheme. That is, the client would split its query into k pieces, send one share to each server, each server would apply the same linear function to the client's query, and the client would reconstruct the response. The rest of the protocol proceeds as in the scheme of prior work.

This gives a k-server protocol with offline communication $n^{2/3}$ bits per server and online time $n^{2/3}$, with security against adversarial coalitions of up to $k - 1$ servers. Unfortunately, in this scheme, the client must rerun the offline phase after each online query. An intriguing open question is whether we can construct more efficient offline/online PIR schemes in the k-server model and whether we can extend such schemes to allow the client to reuse its hint over multiple queries.

9 Related work

Checklist follows recent work on improving the efficiency and privacy of blocklisting systems. CRLite [61], used in the Firefox browser today, gives a sophisticated technique for compressing a certificate-revocation blocklist using a hierarchy of Bloom filters [11]. A browser can download and store this compressed blocklist, and can thus make fast and private local blocklist queries to it. CRLite relies on the fact that the servers can enumerate over the set of valid certificates by inspecting Certificate Transparency logs. Unfortunately, CRLite's optimizations do not apply to our setting—in which the set of all possible URLs is far too large to enumerate. In addition, CRLite inherently requires total communication linear in the size of the blocklist, whereas Checklist can have total communication sublinear in the blocklist size. (In the application of Checklist to Safe Browsing, the total communication is linear in the blocklist size, since the client must download a list of partial hashes, as in Section 6.1.)

Other work has proposed ambitious, if more challenging to deploy, approaches to certificate revocation. Revcast proposes broadcasting certificate-revocation information over FM radio [77]. Let's Revoke [81] proposes modifying the public-key infrastructure to facilitate revocation. Solis and Tsudik [82] identify privacy issues with OCSP certificate revocation checks and propose heuristic privacy protections.

A number of tech companies today maintain blocklists of passwords that have appeared in data breaches. Users can check their passwords against these blocklists to learn whether they should change passwords. Recent work [52, 62, 65, 84, 86] develops protocols with which users can check their passwords against these blocklists while (1) hiding their password from the server and (2) without the server revealing the entire blocklist to the client. Some of these breach-notification services [84] leak a partial hash of the user's password to the server [65]. Schemes using private-set-intersection protocols [21, 22, 73, 75] avoid this leakage, but

require the server to do online work that is linear in the database size. Using Checklist in this setting would eliminate leakage of the hashed password to the server and would reduce the server-side computational cost, since our amortized lookup cost is sublinear in the blocklist size. The downside of Checklist is that it requires two non-colluding servers to hold replicas of the database, whereas these existing schemes do not.

Our focus application of Checklist is to the Safe Browsing API. Prior work has demonstrated the privacy weaknesses of the Safe Browsing API [9, 38], arising from the fact that the client leaks 32-bit hashes of the URLs it visits to the server. Apple recently started to proxy Safe Browsing requests on iOS via Apple servers to hide the requestors' IP addresses from Safe Browsing service providers such Google or Tencent [17].

The private Safe Browsing system of Cui et al. [28] provides privacy to both the client and the server by having the client store a local encrypted copy of the blocklist. The client decrypts individual entries by running an oblivious-pseudorandom-function evaluation protocol with the server. We can view their approach as applying the transformation of Freedman et al. [36], which we mention in Section 8.2, to the simplest possible PIR scheme—storing the full blocklist at the client. In contrast, one of the design goals of Checklist is to avoid the cost of storing the full blocklist.

Piotrowska et al. describe a private notification service called AnNotify [74] and discuss its application to blocklist lookups. Unlike Checklist, AnNotify tolerates some amount of leakage about the queries. To mitigate the remaining leakage, AnNotify runs on top of an anonymity network such as Tor.

The core of Checklist is a new two-server offline/online private-information-retrieval (PIR) scheme in which the servers run in sublinear online time. While one offline/online PIR scheme with sublinear online time appears in prior work [27], ours reduces the online time by a factor of $\lambda \approx 128$ and gives the first implementation of such a scheme.

Our PIR scheme builds on a long and beautiful body of work on privacy-protecting database lookups. The literature on PIR is vast and we will only be able to scratch the surface here. Chor et al. [24,25] initiated the study of PIR in which the client communicates with multiple non-colluding servers. Our PIR scheme works in this multi-server model. Gasarch [37] gives an excellent survey on the state of multi-server PIR as of 2004. Recent work improves the *communication* cost of two-server PIR using sophisticated coding ideas [32, 35, 87]. Under mild assumptions, there exist two-server PIR schemes with almost optimal communication cost [14, 15, 39, 48]. An orthogonal goal is to protect against PIR server misbehavior [29, 40].

Given that modern multi-server PIR schemes have very low communication costs, the remaining task is to reduce the server-side *computational* cost of multi-server PIR. On a database of n rows, the above PIR schemes have server-side cost $\Omega(n)$. Beimel et al. [8] show that if the servers preprocess the database, they can respond to client queries in $o(n)$ time. Unfortunately, the schemes of Beimel et al. [8] are relatively expensive in terms of communication cost and require very large amounts of server storage. Alternatively, "batch PIR" [49, 54] allows the client to fetch many records at roughly the server-side cost of fetching a single record. Lueks and Goldberg extend this approach to allow the servers to answer queries from many mutually distrusting clients at less than the cost of answering each client's request independently [66]. Other work relaxes the privacy guarantees of PIR to improve performance [85]. Our work builds most directly on offline/online PIR protocols [27, 71], in which the client fetches some information about the database in an offline phase to improve online performance.

Under appropriate "public-key assumptions" [31], it is possible [19, 59] to construct PIR schemes in which the client communicates with only a single database server. Sion and Carbunar [80] ask whether single-server PIR schemes can ever be more efficient (in terms of total time) than the naïve PIR scheme in which the client downloads the entire database. Olumofin and Goldberg [69] argue that modern lattice-based protocols can indeed outperform the trivial PIR protocols. Recent work has refined single-server lattice-based schemes using batch-PIR techniques to get relatively efficient single-server PIR schemes [2, 3, 4, 5]. The reliance on public-key primitives makes these schemes concretely more expensive than the multi-server schemes we construct, but they are invaluable in settings in which multiple servers are unavailable.

Finally, prior work has applied PIR to private media consumption [47], eCommerce [50], and private messaging [5].

10 Conclusion

With Checklist, a client can check a string against a server-side blocklist, without revealing its string to the server. Checklist uses significantly less communication and storage than a baseline scheme in which the client downloads and maintains a local copy of the entire blocklist. Our new offline/online private-information-retrieval scheme reduces the server-side cost of Checklist compared to previous private-information-retrieval schemes. We hope that Checklist leads to further improvements in practical private-information-retrieval systems and that it encourages large-scale deployment of privacy-preserving blocklist systems in major web browsers.

Acknowledgements. We gratefully acknowledge Dan Boneh for his advice and support throughout this project. Eric Rescorla first brought these privacy concerns with Safe Browsing to our attention and asked whether PIR schemes could ever be fast enough to address them. We thank Kostis Kaffes for very helpful conversations on our experimental evaluation. Krzysztof Pietrzak suggested a technique to improve the efficiency of our earlier PIR scheme [27], which was helpful as we developed the results of Section 4. Elaine Shi kindly pointed us to related work on dynamic data structures. A team at Google, including Alex Wozniak, Emily Stark, Rui Wang, Nathan Parker, and Varun Khaneja answered a number of our questions about the internals of the Safe Browsing service. Finally, we thank the USENIX Security reviewers and our shepherd, Ian Goldberg, for

extensive feedback and suggestions on how to improve the paper. This work was funded by NSF, DARPA, a grant from ONR, the Simons Foundation, a Facebook research award, a Google research award, and a Google Cloud Platform research-credits award. Opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

References

[1] Source code for Checklist. Available at: `https://github.com/dimakogan/checklist`.

[2] Carlos Aguilar-Melchor, Joris Barrier, Laurent Fousse, and Marc-Olivier Killijian. XPIR: Private information retrieval for everyone. *PoPETs*, 2016(2):155–174, 2016.

[3] Asra Ali, Tancrède Lepoint, Sarvar Patel, Mariana Raykova, Phillipp Schoppmann, Karn Seth, and Kevin Yeo. Communication-computation trade-offs in PIR. Cryptology ePrint Archive, Report 2019/1483, 2019.

[4] Sebastian Angel, Hao Chen, Kim Laine, and Srinath T. V. Setty. PIR with compressed queries and amortized query processing. In *IEEE Symposium on Security and Privacy*, 2018.

[5] Sebastian Angel and Srinath Setty. Unobservable communication over fully untrusted infrastructure. In *SOSP*, 2016.

[6] Adam Back. Hashcash – a denial of service counter-measure. August 2002.

[7] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *CRYPTO*, 2000.

[8] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers' computation in private information retrieval: PIR with preprocessing. *J. Cryptol.*, 17(2):125–151, 2004.

[9] Simon Bell and Peter Komisarczuk. An analysis of phishing blacklists: Google Safe Browsing, OpenPhish, and PhishTank. In *Proceedings of the Australasian Computer Science Week, ACSW*, 2020.

[10] Jon Louis Bentley and James B. Saxe. Decomposable searching problems I: static-to-dynamic transformation. *J. Algorithms*, 1(4):301–358, 1980.

[11] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[12] Dan Boneh, Sam Kim, and Hart William Montgomery. Private puncturable PRFs from standard lattice assumptions. In *EUROCRYPT*, 2017.

[13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In *ASIACRYPT*, 2013.

[14] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*, 2015.

[15] Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing: Improvements and extensions. In *CCS*, 2016.

[16] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In *PKC*, 2014.

[17] Taha Broach. `https://the8-bit.com/apple-proxies-google-safe-browsing-privacy/`, 2021.

[18] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[19] Christian Cachin, Silvio Micali, and Markus Stadler. Computationally private information retrieval with polylogarithmic communication. In *EUROCRYPT*, 1999.

[20] Nishanth Chandran, Bhavana Kanukurthi, and Rafail Ostrovsky. Locally updatable and locally decodable codes. In *TCC*, 2014.

[21] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. Labeled PSI from fully homomorphic encryption with malicious security. In *CCS*, 2018.

[22] Hao Chen, Kim Laine, and Peter Rindal. Fast private set intersection from homomorphic encryption. In *CCS*, 2017.

[23] Benny Chor, Niv Gilboa, and Moni Naor. Private information retrieval by keywords. Cryptology ePrint Archive, Report 1998/003, 1998.

[24] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. In *FOCS*, 1995.

[25] Benny Chor, Oded Goldreich, Eyal Kushilevitz, and Madhu Sudan. Private information retrieval. *J. ACM*, 45(6):965–982, 1998.

[26] ClamAV. ClamAV Documentation: File hash signatures. `https://www.clamav.net/documents/file-hash-signatures`.

[27] Henry Corrigan-Gibbs and Dmitry Kogan. Private information retrieval with sublinear online time. In *EUROCRYPT*, 2020.

[28] Helei Cui, Yajin Zhou, Cong Wang, Xinyu Wang, Yuefeng Du, and Qian Wang. PPSB: An open and flexible platform for privacy-preserving Safe Browsing. *IEEE Transactions on Dependable and Secure Computing*, 2019.

[29] Casey Devet, Ian Goldberg, and Nadia Heninger. Optimally robust private information retrieval. In *USENIX Security*, 2012.

[30] Giovanni Di Crescenzo, Yuval Ishai, and Rafail Ostrovsky. Universal service-providers for private information retrieval. *J. Cryptol.*, 14(1):37–74, 2001.

[31] Giovanni Di Crescenzo, Tal Malkin, and Rafail Ostrovsky. Single database private information retrieval implies oblivious transfer. In *EUROCRYPT*, 2000.

[32] Zeev Dvir and Sivakanth Gopi. 2-server PIR with subpolynomial communication. *J. ACM*, 63(4):39:1–39:15, 2016.

[33] Cynthia Dwork and Moni Naor. Pricing via processing or combatting junk mail. In *CRYPTO*, 1992.

[34] Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Foundations and Trends in Theoretical Computer Science*, 9(3-4):211–407, 2014.

[35] Klim Efremenko. 3-query locally decodable codes of subexponential length. *SIAM J. Comput.*, 41(6):1694–1703, 2012.

[36] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, 2005.

[37] William Gasarch. A survey on private information retrieval. *Bulletin of the EATCS*, 82(72-107):113, 2004.

[38] Thomas Gerbet, Amrit Kumar, and Cédric Lauradoux. A privacy analysis of Google and Yandex Safe Browsing. In *IEEE/IFIP International Conference on Dependable Systems and Networks, DSN*, 2016.

[39] Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, 2014.

[40] Ian Goldberg. Improving the robustness of private information retrieval. In *IEEE Symposium on Security and Privacy*, 2007.

[41] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.

[42] Google. Compression in Safe Browsing APIs (v4)). https://developers.google.com/safe-browsing/v4/compression.

[43] Google. Safe Browsing APIs (v4). https://developers.google.com/safe-browsing/v4.

[44] Google. Safe Browsing transparency report. https://transparencyreport.google.com/safe-browsing/overview. 5 December 2020. Retreived from http://archive.today/w9vao.

[45] Matthew Green. How safe is Apple's Safe Browsing? https://blog.cryptographyengineering.com/2019/10/13/dear-apple-safe-browsing-might-not-be-that-safe/, 2019.

[46] Chun Guo, Jonathan Katz, Xiao Wang, and Yu Yu. Efficient and secure multiparty computation from fixed-key block ciphers. In *IEEE Symposium on Security and Privacy*, 2020.

[47] Trinabh Gupta, Natacha Crooks, Whitney Mulhern, Srinath Setty, Lorenzo Alvisi, and Michael Walfish. Scalable and private media consumption with Popcorn. In *NSDI*, 2016.

[48] Syed Mahbub Hafiz and Ryan Henry. A bit more than a bit is more than a bit better: Faster (essentially) optimal-rate many-server pir. 2019.

[49] Ryan Henry. Polynomial batch codes for efficient IT-PIR. *PoPETs*, 2016(4):202–218, 2016.

[50] Ryan Henry, Femi Olumofin, and Ian Goldberg. Practical PIR for electronic commerce. In *CCS*, 2011.

[51] Susan Hohenberger, Venkata Koppula, and Brent Waters. Adaptively secure puncturable pseudorandom functions in the standard model. In *ASIACRYPT*, 2015.

[52] Troy Hunt. Have I been pwned. https://haveibeenpwned.com/FAQs.

[53] Internet Storm Center. SSL CRL activity. https://isc.sans.edu/crls.html.

[54] Yuval Ishai, Eyal Kushilevitz, Rafail Ostrovsky, and Amit Sahai. Batch codes and their applications. In *STOC*, 2004.

[55] J. C. Jones. Design of the CRLite infrastructure. https://blog.mozilla.org/security/2020/12/01/crlite-part-4-infrastructure-design/, December 2020.

[56] Daniel Kales. Go DPF library. https://github.com/dkales/dpf-go, 2019.

[57] Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In *CCS*, 2013.

[58] Scott Knight. syspolicyd internals. https://knight.sc/reverse%20engineering/2019/02/20/syspolicyd-internals.html, February 2019.

[59] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *FOCS*, 1997.

[60] Adam Langley. CRL set tools. https://github.com/agl/crlset-tools.

[61] James Larisch, David Choffnes, Dave Levin, Bruce M Maggs, Alan Mislove, and Christo Wilson. CRLite: A scalable system for pushing all TLS revocations to all browsers. In *IEEE Symposium on Security and Privacy*, 2017.

[62] Kristin Lauter, Sreekanth Kannepalli, Kim Laine, and Radames Cruz Moreno. Password Monitor: Safeguarding passwords in Microsoft Edge. https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/, January 2021.

[63] Daniel Lemire. A fast alternative to the modulo reduction. https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/, 2016.

[64] Daniel Lemire. Fast random integer generation in an interval. *ACM Trans. Model. Comput. Simul.*, 29(1):3:1–3:12, 2019.

[65] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. Protocols for checking compromised credentials. In *CCS*, 2019.

[66] Wouter Lueks and Ian Goldberg. Sublinear scaling for multi-client private information retrieval. In *Financial Cryptography*, 2015.

[67] Andrés Cecilia Luque. Apple is sending a request to their servers for every piece of software you run on your Mac. https://medium.com/@acecilia/apple-is-sending-a-request-to-their-servers-for-every-piece-of-software-you-run-on-your-mac-b0bb509eee65, May 2020.

[68] Mozilla. Firefox for android—releases. https://github.com/mozilla-mobile/fenix/releases.

[69] Femi Olumofin and Ian Goldberg. Revisiting the computational practicality of private information retrieval. In *Financial Cryptography*, 2011.

[70] Charalampos Papamanthou, Roberto Tamassia, and Nikos Triandopoulos. Authenticated hash tables based on cryptographic accumulators. *Algorithmica*, 74(2):664–712, 2016.

[71] Sarvar Patel, Giuseppe Persiano, and Kevin Yeo. Private stateful information retrieval. In *CCS*, 2018.

[72] Ben Perez. How safe browsing fails to protect user privacy. https://blog.trailofbits.com/2019/10/30/how-safe-browsing-fails-to-protect-user-privacy/, 2019.

[73] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. Spot-light: Lightweight private set intersection from sparse OT extension. In *CRYPTO*, 2019.

[74] Ania M. Piotrowska, Jamie Hayes, Nethanel Gelernter, George Danezis, and Amir Herzberg. Annotify: A private notification service. In , *WPES*, 2017.

[75] Peter Rindal and Mike Rosulek. Malicious-secure private set intersection via dual execution. In *CCS*, 2017.

[76] Amit Sahai and Brent Waters. How to use indistinguishability obfuscation: deniable encryption, and more. In *STOC*, 2014.

[77] Aaron Schulman, Dave Levin, and Neil Spring. RevCast: Fast, private certificate revocation over fm radio. In *CCS*, 2014.

[78] Elaine Shi, Waqar Aqeel, Balakrishnan Chandrasekaran, and Bruce Maggs. Puncturable pseudorandom sets and private information retrieval with polylogarithmic bandwidth and sublinear time. Cryptology ePrint Archive, Report 2020/1592, 2020.

[79] Elaine Shi, Emil Stefanov, and Charalampos Papamanthou. Practical dynamic proofs of retrievability. In *CCS*, 2013.

[80] Radu Sion and Bogdan Carbunar. On the practicality of private information retrieval. In *NDSS*, 2007.

[81] Trevor Smith, Luke Dickinson, and Kent Seamons. Let's revoke: Scalable global certificate revocation. In *NDSS*, 2020.

[82] John Solis and Gene Tsudik. Simple and flexible revocation checking with privacy. In *International Workshop on Privacy Enhancing Technologies*, 2006.

[83] Emil Stefanov, Charalampos Papamanthou, and Elaine Shi. Practical dynamic searchable encryption with small leakage. In *NDSS*, 2014.

[84] Kurt Thomas, Jennifer Pullman, Kevin Yeo, Ananth Raghunathan, Patrick Gage Kelley, Luca Invernizzi, Borbala Benko, Tadek Pietraszek, Sarvar Patel, Dan Boneh, and Elie Bursztein. Protecting accounts from credential stuffing with password breach alerting. In *USENIX Security*, 2019.

[85] Raphael R. Toledo, George Danezis, and Ian Goldberg. Lower-cost ϵ-private information retrieval. *PoPETs*, 2016(4):184–201, 2016.

[86] Ke Coby Wang and Michael K. Reiter. Detecting stuffing of a user's credentials at her own accounts. In *USENIX Security*, 2020.

[87] Sergey Yekhanin. Towards 3-query locally decodable codes of subexponential length. *J. ACM*, 55(1):1:1–1:16, 2008.

A Definitions for puncturable pseudorandom sets

This definition comes directly from prior work on puncturable pseudorandom sets [27, 78].

Correctness. We say that a puncturable pseudorandom set $(\mathsf{Gen}, \mathsf{GenWith}, \mathsf{Eval}, \mathsf{Punc})$ is *correct* if for all $n \in \mathbb{N}$, $\mathsf{sk} \leftarrow \mathsf{Gen}(n)$, and $S \leftarrow \mathsf{Eval}(\mathsf{sk})$:

(a) S is a size-$\sqrt{n}$ subset of $[n]$, and

(b) for all $i \in S$, $\mathsf{Eval}(\mathsf{Punc}(\mathsf{sk}, i)) = S \setminus \{i\}$.

In addition, for all $i \in [n]$, $\mathsf{sk} \leftarrow \mathsf{GenWith}(n, i)$, and $S \leftarrow \mathsf{Eval}(\mathsf{sk})$, Properties (a) and (b) must hold, and additionally it must hold that $i \in S$.

Security. We say that a puncturable pseudorandom set $(\mathsf{Gen}, \mathsf{GenWith}, \mathsf{Eval}, \mathsf{Punc})$ is *secure* if for every polynomially bounded $n = n(\lambda)$ and every efficient adversary $\mathcal{A}$, there exists a negligible function $\mathrm{negl}(\lambda)$ such that for every $\lambda \in \mathbb{N}$, it holds that

$$\left| \Pr\left[i = i' : \begin{array}{c} \mathsf{sk} \leftarrow \mathsf{Gen}(n) \\ S \leftarrow \mathsf{Eval}(\mathsf{sk}) \\ i \xleftarrow{R} S \\ \mathsf{sk}_\mathsf{p} \leftarrow \mathsf{Punc}(\mathsf{sk}, i) \\ i' \leftarrow \mathcal{A}(\mathsf{sk}_\mathsf{p}) \end{array} \right] - \frac{1}{n - \sqrt{n} + 1} \right| \leq \mathrm{negl}(\lambda) .$$

The same must hold if we sample sk by choosing $i \xleftarrow{R} [n]$ and setting $\mathsf{sk} \leftarrow \mathsf{GenWith}(n, i)$.

Prior work [27, Appendix B.1., Proposition 34] shows that the above security property implies that the output of Eval on a random set key is a size-$\sqrt{n}$ pseudorandom subset of $[n]$.

B Security analysis

We begin by analyzing a single query of Construction 1.

Lemma B.1 (Correctness)**.** *For every $n = n(\lambda)$ and $\ell = \ell(\lambda)$ that are polynomially bounded in the security parameter $\lambda \in \mathbb{N}$, every database $\mathcal{D} = (D_1, \ldots, D_n)$ with n records of length ℓ bits each, and every $i \in [n]$, the client succeeds in retrieving the ith record of the database with all but a negligible probability.*

Proof. Suppose first that the Query algorithm does not abort. Let D'_i be the output of Reconstruct. Let $S_{\mathsf{p}0}$ and $S_{\mathsf{p}1}$ be the punctured sets evaluated as part of the Answer algorithm by servers 0 and 1 respectively. We consider the following two cases:

- When $\beta = 0$, the client takes a key sk_t of a set S_t in the hint, and derives its query to the second server as $\mathsf{sk}_{\mathsf{p}1} \leftarrow \mathsf{Punc}(\mathsf{sk}_t, i)$. Therefore, $S_{\mathsf{p}1} = S_t \setminus \{i\}$, and the second server's response contains the parity word
$$W_1 \leftarrow \bigoplus_{j \in S_{\mathsf{p}1}} D_j = \bigoplus_{j \in S_t \setminus \{i\}} D_j = P_t \oplus D_i ,$$
where P_t is the tth parity word in the client's hint. The client then correctly reconstructs $D'_i \leftarrow P_t \oplus W_1 = D_i$.
- When $\beta = 1$, it holds that $D'_i = W_0 \oplus W_1 \oplus V_\gamma$ where W_0 and W_1 are the parity words in the answers of the two servers and $V_\gamma = D_{r_\gamma}$ is the extra record in the answer of server γ. From
$$S_{\mathsf{p}\gamma} = S_{\mathsf{new}} \setminus \{i\} \text{ and}$$
$$S_{\mathsf{p}\bar{\gamma}} = S_{\mathsf{new}} \setminus \{r_\gamma\} ,$$
it follows that
$$\begin{aligned} D'_i &= W_\gamma \oplus W_{\bar{\gamma}} \oplus V_\gamma \\ &= \left(\bigoplus_{j \in S_{\mathsf{new}} \setminus \{i\}} D_j \right) \oplus \left(\bigoplus_{j \in S_{\mathsf{new}} \setminus \{r_\gamma\}} D_j \right) \oplus D_{r_\gamma} \\ &= D_i . \end{aligned}$$

Finally, observe that the Query algorithm only fails if none of $T = \lambda n$ pseudorandom sets of size $\sqrt{n}$ contain element i, which happens with negligible probability [27, Appendix C.2]. □

We now turn to the security proof of a single query of our scheme.

Lemma B.2 (Security)**.** *Suppose that the underlying puncturable pseudorandom set is secure. Then for every $n = n(\lambda)$ and $\ell = \ell(\lambda)$ that are polynomially bounded in the security parameter $\lambda \in \mathbb{N}$, every database D with n records of length ℓ bits each, every server $s \in \{0, 1\}$, and every $i, i' \in [n]$, the distributions $\mathsf{View}_{D,s,i}$ and $\mathsf{View}_{D,s,i'}$ are computationally indistinguishable.*

Proof. Consider the following sequence of distributions.

$\underline{\mathsf{Hyb}_0} = \mathsf{View}_{D,s,i}$ is the view of server s when the client is reading index i. The view consists of $(\mathsf{sk}_{\mathsf{p}s}, w_s)$. When $s = 0$, the view also includes the hint h. Without loss of generality, to allow a uniform treatment of the two cases ($s = 0$ and $s = 1$), we artificially extend the view for the case of $s = 1$ to include a random hint h', independent of $\mathsf{sk}_{\mathsf{p}1}$.

$\underline{\mathsf{Hyb}_1}$: We replace the extra index w_s in the view with a random index $\tilde{w} \xleftarrow{R} S_{\mathsf{p}s}$ from the set $S_{\mathsf{p}s} \leftarrow \mathsf{Eval}(\mathsf{sk}_{\mathsf{p}s})$.

The distributions Hyb_0 and Hyb_1 are identical, since in both QueryCommon and QueryRare, the extra index w_s is already a uniformly random element in the punctured set $S_{\mathsf{p}s}$.

$\underline{\mathsf{Hyb}_2}$: We modify the QueryCommon algorithm such that it always sets $\mathsf{sk}_{\mathsf{p}1} \leftarrow \mathsf{sk}_{\mathsf{p}0}$ rather than having it be the result of puncturing a key sk_t from the hint.

We consider two cases.

- When $s = 0$, the distributions Hyb_1 and Hyb_2 are identical since our change only affects $\mathsf{sk}_{\mathsf{p}1}$ and not $\mathsf{sk}_{\mathsf{p}0}$.
- When $s = 1$, algorithm QueryCommon generates the punctured key $\mathsf{sk}_{\mathsf{p}1}$ differently in each of the two hybrids. In Hyb_1, the

QueryCommon obtains $\mathsf{sk}_{\mathsf{p}1}$ by finding in the hint a key sk_t such that $i \in \mathsf{Eval}(\mathsf{sk}_t)$ and puncturing it. (If there is no such set in the hint, QueryCommon aborts in both hybrids.) In contrast, in Hyb_2, algorithm QueryCommon takes $\mathsf{sk}_{\mathsf{p}1}$ to be $\mathsf{sk}_{\mathsf{p}0}$, which is the result of running $\mathsf{sk}_{\mathsf{new}} \leftarrow \mathsf{GenWith}(n, i)$ and puncturing $\mathsf{sk}_{\mathsf{new}}$ at index i. By the properties of puncturable pseudorandom sets (Appendix A), these two methods for generating the set key $\mathsf{sk}_{\mathsf{p}1}$ result in identically distributed punctured keys.

Moreover, the hint h', which we have artificially added to the view in case $s = 1$, is independent of $\mathsf{sk}_{\mathsf{p}1}$ in both hybrids. Overall, the distributions Hyb_1 and Hyb_2 are identical.

$\underline{\mathsf{Hyb}_3}$: We remove from QueryCommon the check that there exists a set $t \in [T]$ such that $i \in S_t$.

The distributions Hyb_2 and Hyb_3 are statistically indistinguishable, since (according to the correctness proof above) the check that we have removed fails with only negligible probability.

$\underline{\mathsf{Hyb}_4}$: We modify the distribution from which the Query algorithm samples the bit β. Instead of sampling it as $\beta \xleftarrow{R} \mathsf{Bernoulli}(2(\sqrt{n} - 1)/n)$, we sample it as $\beta \xleftarrow{R} \mathsf{Bernoulli}((\sqrt{n} - 1)/n)$. We also modify the QueryRare algorithm such that it always sets $\gamma = \bar{s}$.

We can view this latest change as using QueryCommon in place of QueryRare when $\gamma = s$. Given our previous changes, QueryCommon sets $\mathsf{sk}_{\mathsf{p},s} \leftarrow \mathsf{Punc}(\mathsf{sk}_{\mathsf{new}}, i)$, which is identical to what QueryRare does when $\gamma = s$. Our change is therefore purely syntactical, and the distributions Hyb_3 and Hyb_4 are identical.

We can now write Hyb_4 explicitly as follows:

$\mathsf{sk}_{\mathsf{new}} \xleftarrow{R} \mathsf{GenWith}(n, i)$
$S_{\mathsf{new}} \leftarrow \mathsf{Eval}(\mathsf{sk}_{\mathsf{new}})$
$\beta \xleftarrow{R} \mathsf{Bernoulli}((\sqrt{n} - 1)/n)$
if $\beta = 0$: $i_{\mathsf{punc}} \leftarrow i$
else: $i_{\mathsf{punc}} \xleftarrow{R} S_{\mathsf{new}} \setminus \{i\}$
$\mathsf{sk}_{\mathsf{p}} \leftarrow \mathsf{Punc}(\mathsf{sk}_{\mathsf{new}}, i_{\mathsf{punc}})$
$h \leftarrow (\mathsf{sk}_1, \ldots, \mathsf{sk}_T)$ where $\mathsf{sk}_j \leftarrow \mathsf{Gen}(n)\ \forall j \in [T]$
$\tilde{w} \xleftarrow{R} S_{\mathsf{new}} \setminus \{i_{\mathsf{punc}}\}$
Output$(h, \mathsf{sk}_{\mathsf{p}}, \tilde{w})$

Lemma 36 in [27] shows that sk_{p} when reading index i is computationally indistinguishable from sk_{p} when reading index i'. The remaining elements in the view—$\tilde{w}$ and h—are also independent of i. Therefore, Hyb_4 when the client is reading index i is computationally indistinguishable from Hyb_4 when the client is reading index i'. Together with the indistinguishability of the sequence of hybrids, this implies that $\mathsf{Hyb}_0 = \mathsf{View}_{D,s,i}$ is computationally indistinguishable from $\mathsf{Hyb}'_0 = \mathsf{View}_{D,s,i'}$, which completes the proof of the lemma. □

The extension to the multi-query case is identical to the proof of Lemma 45 in prior work[27].

C Additional optimizations

The puncturable pseudorandom set of Section 3.2 builds on a tree-based construction of a pseudorandom function. Each node in the binary tree has an associated pseudorandom label, where the two labels of each inner node's two children are recursively generated by evaluating a pseudorandom generator on the label of the parent node. The simple length-doubling pseudorandom generator we use is based on fixed-key AES [46]. Using a fixed key avoids the additional cost of key scheduling that is incurred with more traditional constructions of stream ciphers from block ciphers (e.g., counter mode). We further reduce the time to evaluate the pseudorandom generator on every node in a binary tree by using a breadth-first traversal. Our bread-first evaluation computes the labels for an entire layer of the binary tree using a single tight loop that essentially encrypts a sequence of blocks using AES with a single key. Evaluating the entire tree requires calling said loop once per layer. This results in a 7× faster running time, compared to a depth-first implementation, which essentially encrypts a much larger number of blocks one by one.

Another implementation detail that reduces the puncturable-set evaluation time is the usage of "Lemire's trick" [63, 64] for mapping random 32-bit values to random integers in a specified range without using arithmetic modulo operations.

Finally, to compute the XOR of a large number of database records, we use single-instruction multiple-data (SIMD) extensions—Intel SSE2 at the server and Arm Neon at the mobile client.