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 list of blocklisted 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 and (2) 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 eliminate 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×. Use of our new PIR protocol reduces server-side costs by 6.7×, compared to what would be possible with prior state-of-the-art two-server PIR.

1 Introduction

This paper proposes a new system for *private blocklist lookups*. In this setting, there is a client, who 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 [51, 56, 57].
- Users of Google's Password Checkup and the "Have I Been Pwned?" service check their passwords against a blocklist of breached credentials [48, 58, 61, 77, 80].
- Antivirus tools check the hashes of executed binaries against blocklists of malicious software [24, 54, 63].
- Browsers and mail clients check URLs against Google's Safe Browsing blocklist of phishing sites [8, 34, 40].

In each of these settings, the string that the client is checking against the blocklist is *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.

Today, there are two common approaches to solving this private blocklist-lookup problem. The first approach is to store the entire blocklist on the client side. 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. Because of these resource constraints, Chrome's client-side certificate-revocation blocklist [56] (version 6391, as of January 2020) covers under one thousand revoked certificates out of *millions* of revoked certificates on the web [49], and thus provides suboptimal protection against revoked certificates.

The second approach is to store the blocklist on the server side. The downside of this technique is that the client may leak bits of its private string when it queries the blocklist server. For example, when the Firefox web browser queries the server-side Safe Browsing blocklist, the browser reveals a 32-bit hash of its query to the server. This hash allows the server to make a good guess at which URL the browser is visiting [8, 34, 79].

Both existing techniques for private blocklist lookups are inadequate. Keeping the blocklist on the client is infeasible when the blocklist is large. Keeping the blocklist on the server 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 and (2) 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 [11, 25, 66]. These schemes use client-specific preprocessing in an offline phase to reduce the computation required at query (online) time. On a blocklist 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 [25] 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-line 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 [14] (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.

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 blacklist. (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 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*. There is a set $B \subseteq \{0,1\}^*$ of blocklisted strings, of which each server has a copy. Initially, the client may download some information about the blocklist from the servers. Later on, the client would like to issue *queries* to the blocklist: the client holds a query string $s \in \{0,1\}^*$ and, after interaction with the servers, the client should learn whether or not the query string s is on the servers' blocklist (i.e., whether $s \in B$). 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 9.

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.

We assume, without loss of generality, that the blocklisted strings are all of some common length (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.

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 [14, 22, 35, 55] minimizes communication but requires per-client server-side computation linear in the blocklist size. Checklist minimizes the server-side computation while keeping the total communication without the client having to download and store the entire blocklist.

2.2 Design overview

Checklist consists of three main layers: the first layer provides private database lookups, albeit to static array-like databases. The second layer further allows the server to continuously add entries to the database. The third layer enables key-value lookups. We now explain the design of each of the layers.

Private lookups. A straightforward way to implement private lookups is to use private information retrieval (PIR) [14,21,22]. 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 [25] 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.

Database updates. Offline/online PIR schemes allow the server to answer client queries at 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 idea is to divide the length-n blocklist into $O(\log n)$ buckets, where the ith bucket contains at most 2^i 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. (Specifically, the contents of bucket i change only after every 2^i blocklist updates.)

With this strategy, if a handful of database entries change between each client blocklist lookup, the amortized cost per blocklist update is $O(\log n)$ in the blocklist size. In contrast, a naïve application of offline/online PIR would lead to $\Omega(n)$ amortized cost per update.

Lookup by key. PIR protocols typically treat the database as an array of n rows. To fetch a row, a PIR client must then specify the index $i \in [n]$ of the row. In contrast, Checklist, as many other applications of PIR, needs to support key-value lookups, in which the database is a collection of key-value pairs $\{(k_i, v_i)\}_{i=1}^{n}$, the client holds a key k_i and it wants the corresponding value v_i (if one exists). It is possible to construct such PIR-by-keywords schemes from PIR-by-index schemes in a black-box way [21] or directly using modern PIR constructions [14]. The cost of such schemes, both in communication and server-side computation, matches the cost of standard PIR, up to low-order terms. In our implementation of Checklist, we use an alternative approach, that takes advantage of an existing feature in Safe Browsing, due to it being the main target application in this work. Specifically, the Safe Browsing protocol enables a client to rule out the appearance in the blocklist of a vast majority of URLs by shipping to the browser a list of partial hashes of the URLs in the blocklist. Checklist then uses the position of each partial hash in this list as a keyword-to-index mapping. We discuss this in more detail in Section 6.

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 $\{1, 2, \ldots, n\}$ and 1^n denotes the all-ones binary string of length 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 $\mathrm{poly}(\cdot)$ refers to a fixed polynomial and $\mathrm{negl}(\cdot)$ refers to a function whose inverse grows faster than any fixed polynomial. For a finite set S, the notation $x \xleftarrow{R} S$ refers to choosing x independently and uniformly at random from the set S. For $p \in [0, 1]$, the notation $b \xleftarrow{R} \mathrm{Bernoulli}(p)$ 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 $1 - b$.

3.1 Private information retrieval (PIR)

In a private information retrieval (PIR) system [22, 23], 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 [6].

Offline/online PIR. This linear-time cost on the servers is a performance bottleneck so, following recent work [11, 25, 27, 66], we construct "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. If the database contains n rows, generating each hint takes $\approx n \log n$ time, but the hint only has size $\approx \sqrt{n}$.
- **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 total communication required in this step is $\approx \log n$ and the servers can answer the client's query in $\approx \sqrt{n}$ time. The client can use a single hint to make arbitrarily many online queries.

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 only $\approx \sqrt{n}$, instead of $\approx n$. 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 amortized server-side computational cost per query is only $\approx \sqrt{n}$, compared with $\approx n$ for standard PIR schemes.

3.2 Puncturable pseudorandom set

To construct our PIR schemes, we will use *puncturable pseudorandom sets* [25, 71], for which there are simple constructions from any pseudorandom generator (e.g., AES-CTR).

Informally, a puncturable pseudorandom set gives a way to describe of 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 [12, 15, 47, 53, 69].

The full syntax and definitions appear in prior work [25], 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:

- $\mathsf{Gen}() \to \mathsf{sk}$. Generate a puncturable set key sk.
- $\mathsf{GenWith}(i) \to \mathsf{sk}$. Given an element $i \in [n]$, generate a puncturable set key sk such that the element $i \in \mathsf{Eval}(\mathsf{sk})$.
- $\mathsf{Eval}(\mathsf{sk}) \to S$. Given a key sk (or punctured set key sk_p), output a pseudorandom set $S \subseteq [n]$ of size $\sqrt{n}$.
- $\mathsf{Punc}(\mathsf{sk}, i) \to \mathsf{sk}_\mathsf{p}$. 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 [25] 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 have 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. Compared with the best prior two-server scheme [25], 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 database size n and security parameter λ, the online server in the existing PIR schemes have online communication $O(\lambda^2 \log n)$ and have online server time $O(\lambda\sqrt{n})$, measured in terms of main-memory reads and evaluations of a length-doubling PRG. We bring the online communication cost down to $O(\lambda \log n)$ bits and the online server-side computation time down to $O(\sqrt{n})$ operations (dominated by the cost of $O(\sqrt{n})$ AES operations and $O(\sqrt{n})$ random-access memory lookups). Concretely, these cost 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, our protocol uses $\lambda\sqrt{n}$ bits of communication and requires the server to do $O(\lambda n)$ work per client.

4.1 Definition

A two-server offline/online PIR scheme for an n-bit database consists of the following four algorithms, where we leave the security parameter implicit. We refer to the two PIR servers as the "left" and "right" servers.

- $\mathsf{Hint}(D) \to h$. The left database server uses the Hint algorithm to generate a preprocessed data structure h that a client can later use to privately query the database $D \in \{0,1\}^n$. The Hint algorithm is randomized, and the left 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.
- $\mathsf{Answer}(D, q) \to a$. The servers uses Answer, on database $D \in \{0,1\}^n$ 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 bit $D_i \in \{0,1\}$.

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 [25].

Correctness. If an honest client interacts with honest servers, the client recovers its desired database bit. We say that an offline/online PIR scheme is *correct* if, for all databases $D = (D_1, \ldots, D_n) \in \{0,1\}^n$ and all $i \in [n]$, the probability

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

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 bit the client is querying, even if the attacker deviates arbitrarily from the prescribed protocol. More formally, for a database $D \in \{0,1\}^n$, $\beta \in \{0,1\}$, and $i \in [n]$, define the probability distribution $\mathsf{View}_{D,\beta,i}$, capturing the "view" of server β as:

$$\mathsf{View}_{D,\beta,i} := \left\{ q_\beta : \begin{array}{rl} h & \leftarrow \mathsf{Hint}(D) \\ (\_, q_0, q_1) & \leftarrow \mathsf{Query}(h, i) \end{array} \right\}.$$

An offline/online PIR scheme is *secure* if, for all databases $D \in \{0,1\}^n$, $\beta \in \{0,1\}$, and $i, j \in [n]$: $\mathsf{View}_{D,\beta,i} \approx_c \mathsf{View}_{D,\beta,j}$.

4.2 Our scheme

Prior offline/online PIR schemes [25] 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 database bit 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 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 1. *Construction 2 is a computationally secure offline/online PIR scheme, assuming the security of the underlying puncturable pseudorandom set. On database size $n \in \mathbb{N}$ and security parameter $\lambda \in \mathbb{N}$ (used to instantiate the puncturable pseudorandom set), the scheme has:*

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

Remark. If each database record is ℓ bits long, the offline communication increases to $\lambda(\sqrt{n} \cdot \ell + 1)$ bits and the online communication increases to $2(\lambda \log n + 1)\log n + 4\ell$ bits.

We formally analyze the correctness and security of Construction 2 in Appendix B. Here, we describe the intuition behind how the construction works.

Offline phase. In the offline phase of the protocol, the left 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 bits indexed by the set $\mathsf{Eval}(\mathsf{sk}_t)$. If the database is $D = D_1 \cdots D_n \in \{0,1\}^n$, then the t-th parity bit is: $h_t = \sum_{j \in \mathsf{Eval}(\mathsf{sk}_t)} D_j \bmod 2$. The t keys $(\mathsf{sk}_1, \ldots, \mathsf{sk}_T)$ along with the T parity bits $(h_1, \ldots, h_T)$ 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 consists of $(\mathsf{seed}, h_1, \ldots, h_T)$ and has length $\lambda + \lambda\sqrt{n}$ bits.

The key property of this hint is that with overwhelming probability (at least $1 - 2^{-\lambda}$), each bit of the database will be included in at least one of the parity bits. 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 bit of the database, for $i \in [n]$. 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 bits $(h_1, \ldots, h_T)$.

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 sets 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 bit $i \in \mathsf{Eval}(\mathsf{sk}_t)$.

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 bit i. The client also

holds the parity h_t of the database bits indexed by S_t To recover the database bit D_i, the client needs only to learn the parity h' of all of the database bits indexed by the set $S_t \setminus \{i\}$, as it would allow the client to recover its database bit of interest as:

$$h + h' = \left(\sum_{j \in S_t} D_j\right) + \left(\sum_{j \in S_t \setminus \{i\}} D_j\right) = D_i \pmod 2.$$

The key challenge is thus for the client to fetch the bit h' from the servers. There are two cases here:

- *Easy case (most of the time).* Most of the time, the client can just send the set $S' \leftarrow S_t \setminus \{i\}$ to the right server. (To save communication, the client compresses this set using puncturable pseudorandom sets.) The server returns the parity h' of the database bits indexed by this set S'.

  However, the set S' *never* contains the index i of the client's desired database bit. So the client cannot always follow this strategy, otherwise the right server would learn which database bits the client is definitely *not* querying. So, with small probability, the client executes the following case.
- *Hard case (rarely).* With small probability (roughly $1/\sqrt{n}$), the client must send a set containing i to each server. To ensure that the client can still recover its desired database bit D_i in this case, we have the client send fresh correlated random sets to the servers.

  Specifically, in this case, the client samples a random size-$(\sqrt{n} - 1)$ set S_{new} of values in $[n]$. The client chooses one of the two servers at random. The client then sends $Z \leftarrow S_{\mathsf{new}} \setminus \{i\}$ to this server (again, compressed using puncturable pseudorandom sets), along with the index of a random element $w \xleftarrow{R} S_{\mathsf{new}} \setminus \{i\}$. The server returns the parity h_Z of the database bits indexed by Z along with the value of the database bit D_w.

  To the other server, the client sends $Z' \leftarrow S_{\mathsf{new}} \setminus \{w\}$. The server replies with the parity $h_{Z'}$ of the database bits indexed by Z'. Now, the client can recover its database bit of interest as: $D_i = h_Z + h_{Z'} + D_w \bmod 2$, since this sum is equal to

  $$\left(\sum_{j \in S_{\mathsf{new}} \setminus \{i\}} D_j\right) + \left(\sum_{j \in S_{\mathsf{new}} \setminus \{w\}} D_j\right) + D_w = D_i \pmod 2.$$

  To hide which server plays which role, the client sends a dummy value w' to this second server.

To hide whether the client is in the "easy case" or "hard case," the client sends dummy queries to the servers in the easy case to mimic its behavior in the hard case.

5 Handling database changes with waterfall updates

In an offline/online PIR scheme, the client downloads a preprocessed "hint" about the database in the offline phase. To generate the hint, the servers must run a costly computation, which takes time linear in the database size. Given the hint, the client can make an essentially unlimited number of subsequent private database queries, which the servers can answer in time *sublinear* in the database size.

This approach works well as long as the database is static. However, a problem arises when the database changes, since any update to the database invalidates the client's hint.

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 strategy, is to have the servers maintain an array of $B = \log n$ sub-databases, which we call "buckets." (Here, we assume for simplicity that n is a power of two.) The b'th bucket will contain at most 2^b database rows. Before a client makes a query, it will hold a preprocessed hint for each of the B buckets. When a client makes a database query, it will make one PIR query to each of the B buckets.

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.

The use of a hierarchy of buckets is a classic idea for converting static data structures into dynamic data structures [9]. Cryptographic constructions using this idea to handle data updates include oblivious RAMs [37], proofs of retrievability [18,72], searchable encryption [76], and accumulators [65].

In more detail, if the database is an array of n bits $(D_1, \ldots, D_n)$, the contents of the b'th bucket in our update scheme is a list of 2^b pairs of the form (i, D_i), where $i \in [n]$ is the index of a database bit and $D_i \in \{0, 1\}$ is that bit's value.

Initially, the servers store the entire database in the bottom (biggest) bucket and all of the other bucket start out empty. That is, the last bucket holds the pairs $\{(i, D_i)\}_{i \in [n]}$.

Construction 2 (Our offline/online PIR scheme)**.** Parameters: database size $n \in \mathbb{N}$, security parameter $\lambda \in \mathbb{N}$, $T := \lambda\sqrt{n}$, puncturable pseudorandom set $(\mathsf{Gen}, \mathsf{GenWith}, \mathsf{Eval}, \mathsf{Punc})$ construction of Section 3.2 with universe size n and set size $\sqrt{n}$.

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

- 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 $h_t \in \{0,1\}$ of the database bits indexed by the set S_t.
    That is, let $h_t \leftarrow \sum_{j \in S_t} D_j \bmod 2$.
- Output the hint as: $h \leftarrow \big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (b_1, \ldots, b_T)\big)$.

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

- Parse the hint h as $\big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (b_1, \ldots, b_T)\big)$.
- Let $t \in [T]$ be a value such that $i \in \mathsf{Eval}(\mathsf{sk}_t)$.
  (If no such value t exists, abort.)
- Sample bits $\beta \xleftarrow{R} \mathsf{Bernoulli}(2(\sqrt{n}-1)/n)$ and $\gamma \xleftarrow{R} \{0,1\}$.
- If $\beta = 0$: $(\mathsf{sk}_{\mathsf{new}}, q_0, q_1) \leftarrow \mathsf{QueryEasy}(i, \mathsf{sk}_t)$.
- If $\beta = 1$: $(\mathsf{sk}_{\mathsf{new}}, q_0, q_1) \leftarrow \mathsf{QueryHard}(i, \gamma)$.
- Set $\mathsf{st} \leftarrow (h, t, \beta, \gamma, \mathsf{sk}_{\mathsf{new}})$.
- Return (st, q_0, q_1).

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

- Parse the query q as a pair $(\mathsf{sk}_{\mathsf{p}}, i)$, where sk_{p} is a punctured set key and $i \in [n]$.
- Compute $S_{\mathsf{p}} \leftarrow \mathsf{Eval}(\mathsf{sk}_{\mathsf{p}})$ and compute the parity $b_{\mathsf{p}} \in \{0,1\}$ of the database bits indexed by this set: $b_{\mathsf{p}} \leftarrow \sum_{j \in S_{\mathsf{p}}} D_i \bmod 2$.
- Return $a \leftarrow (b_{\mathsf{p}}, D_i) \in \{0,1\}^2$ to the client.

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

- Parse the state st as $(h, t, \beta, \gamma, \mathsf{sk}_{\mathsf{new}})$.
- Parse the hint h as $\big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (b_1, \ldots, b_T)\big)$.
- Parse the answers as bits (u_0, v_0) and (u_1, v_1).
  - If $\beta = 0$: *// Easy case*
    - Set $D_i \leftarrow b_t + u_1 \bmod 2$.

    // The client updates the t-th component of the hint.
    - Set $\mathsf{sk}_t \leftarrow \mathsf{sk}_{\mathsf{new}}$ and $b_t \leftarrow D_i + u_0 \bmod 2$.
    - Set $h' \leftarrow \big((\mathsf{sk}_1, \ldots, \mathsf{sk}_T), (b_1, \ldots, b_T)\big)$.
  - If $\beta = 1$: *// Hard case*
    - Set $D_i \leftarrow u_0 + u_1 + v_\gamma \bmod 2$.
    - Set $h' \leftarrow h$. *// The hint is unmodified.*
- Return (h', D_i).

$\underline{\mathsf{QueryEasy}(i, \mathsf{sk}_t) \to (\mathsf{sk}_{\mathsf{new}}, q_0, q_1).}$

// The client asks both servers for the parity of $\sqrt{n}-1$ bits
// and the value of one database bit.
// – None of the bits are the desired database bit D_i.
// – The client asks the right server for the parity of the
// database bits in $S_t \setminus \{i\}$. This parity is equal
// to $b_t + D_i \bmod 2$.

- Sample $\mathsf{sk}_{\mathsf{new}} \leftarrow \mathsf{GenWith}(n, i)$.
- Compute:

$$\begin{aligned} S_{\mathsf{new}} &\leftarrow \mathsf{Eval}(\mathsf{sk}_{\mathsf{new}}) & S_t &\leftarrow \mathsf{Eval}(\mathsf{sk}_t) \\ w_0 &\xleftarrow{R} S_{\mathsf{new}} \setminus \{i\} & w_1 &\xleftarrow{R} S_t \setminus \{i\} \\ \mathsf{sk}_{\mathsf{p}0} &\leftarrow \mathsf{Punc}(\mathsf{sk}_{\mathsf{new}}, i) & \mathsf{sk}_{\mathsf{p}1} &\leftarrow \mathsf{Punc}(\mathsf{sk}_t, i) \\ q_0 &\leftarrow (\mathsf{sk}_{\mathsf{p}0}, w_0) & q_1 &\leftarrow (\mathsf{sk}_{\mathsf{p}1}, w_1). \end{aligned}$$

- Return $(\mathsf{sk}_{\mathsf{new}}, q_0, q_1)$.

$\underline{\mathsf{QueryHard}(i, \gamma) \to (\mathsf{sk}_{\mathsf{new}}, q_0, q_1).}$

// The client asks both servers for the parity of $\sqrt{n}-1$ bits
// and the value of one database bit.
// – The client asks one of the servers (determined by bit γ)
// for the parity of a random set of bits containing D_i.
// – The client asks the other server for the parity of the
// same set of bits, along with the database bit needed to
// reconstruct D_i.

- Sample $\mathsf{sk}_{\mathsf{new}} \leftarrow \mathsf{GenWith}(n, i)$.
- Compute:

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

- Return $(\mathsf{sk}_{\mathsf{new}}, q_0, q_1)$.

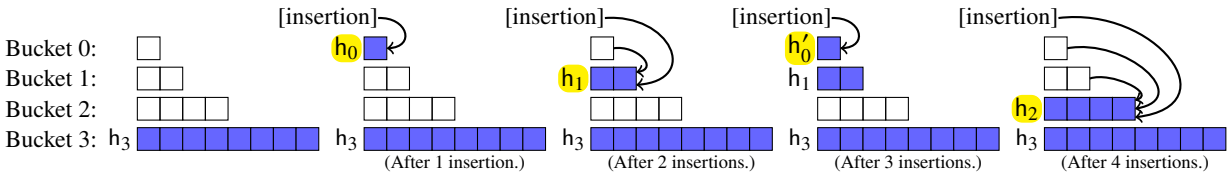


Figure 1: The database in our PIR scheme consists of many buckets, where the ith bucket can hold 2^i database rows. The client holds a hint (h_i) corresponding to each non-empty bucket i. The smaller buckets change frequently, but these hints are inexpensive to recompute. The larger buckets change infrequently,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

When the servers want to update the value of a bit D_i in the database to a new value D'_i, the servers insert the pair (i, D'_i) 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.)

The two servers execute this process in lockstep to ensure that their views of the database state remain consistent throughout. In addition, the servers maintain a last-modified timestamp for each bucket.

When the client joins the system, it fetches a hint for each bucket. In steady state, when the client wants to find the value of the ith database bit, it queries each of the $B = \log n$ buckets in parallel for a pair (i, D_i), using an offline/online PIR-by-keywords scheme on keyword $i \in [n]$. The client uses the value of D_i from the smallest bucket (i.e., the bucket that was updated most recently).

Before making a query, the client updates its locally stored hints. To do this, the client sends to the servers the timestamp τ at which it received its last hint. The servers sends back to the client the hint for each bucket that was modified after τ.

Analysis. Note that the server 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 $\lambda 2^b$ to generate, on security parameter λ. (This is because our offline/online scheme has hint-generation time λn, on security parameter λ and database size n.)

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 $\lambda u B = \lambda u \log n$. In contrast, if we generate a hint for the entire database on each change using the simple scheme, we get total hint generation time of $\lambda u n = \lambda u 2^B$ (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 waterfall scheme is $\sum_{b=1}^{B} O(\sqrt{2^b}) = O(\sqrt{n})$. So, we achieve an exponential improvement in hint-generation cost at a modest (less than fourfold) increase in online query time.

Subtleties. There are a few corner cases still to consider. For example, in our base offline/online PIR scheme, the length of a hint for a layer of size 2^b is roughly $\lambda\sqrt{2^b}$. For layers smaller than λ^2, we would naively get a hint that is larger than the layer itself, and so using offline-online PIR would be worse than just downloading the contents of the entire bucket. To make the communication cost more affordable, we truncate the smallest buckets—the smallest bucket in our scheme is of size at least $5\lambda^2$. For that smallest bucket, we use a traditional PIR scheme, that does not require the client to download a hint on each change. As a result, the client only needs to download an updated hint once every $5\lambda^2 \approx 80{,}000$ updates to the database.

Modifications and deletions. To modify an existing database record (i, D_i), the server adds the updated record (i, D'_i) to the topmost bucket. When the client reads index i from the database, if it gets different values from different buckets, it uses the value from the smallest (most recently updated) bucket. In the bottom-most bucket, the servers can garbage collect old updates by discarding all-but-the-latest $(i, \cdot)$ records for every database index i. Similarly, to delete a record with index i from the database, the server adds a key-value pair $(i, \perp)$ for some special value $\perp$ to the topmost bucket.

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 blocklist an entire portion of a site by just blocklisting

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 memory footprint by 96MB. This would more than *double* the binary size of Chrome on Android [38].
- **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) [40] 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 [10, 16]. 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. 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. This local blocklist is roughly 8× smaller than the list of all 256-bit hashes of the blocklisted URL prefixes.

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.

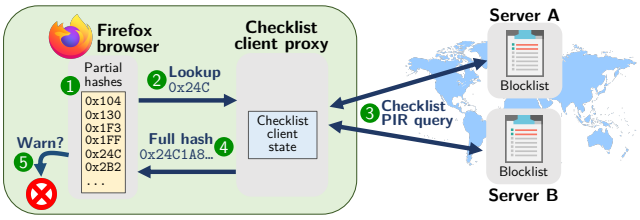


Figure 2: Using Checklist for Safe Browsing. ❶ The browser checks whether the URL's partial hash appears in its local blocklist. ❷ If so, the browser issues a Safe Browsing API query for the full hash corresponding to the matching partial hash. ❸ The Checklist client proxy issues a PIR query for the 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.

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.

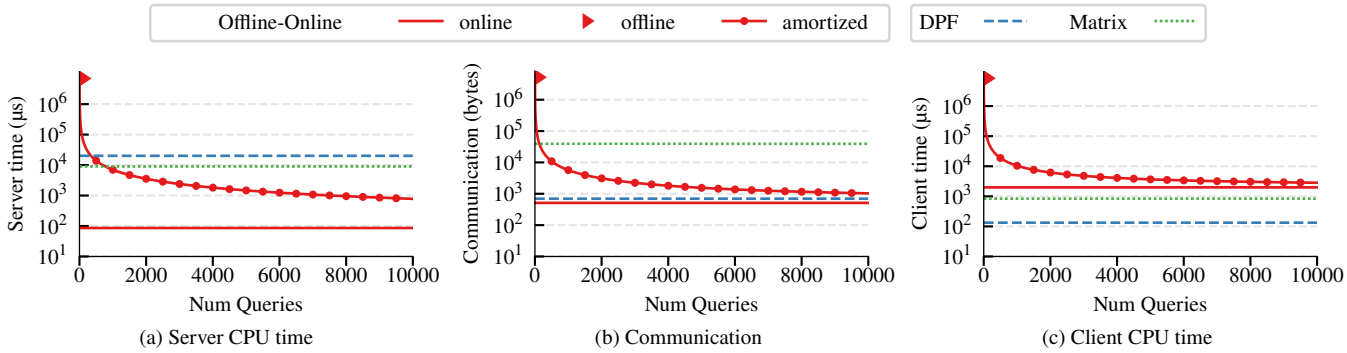
However, as we discuss next, the current Safe Browsing architecture ***leaks information about the user's browsing history to the Safe Browsing servers***.

6.2 Safe Browsing privacy failure

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. Prior work has observed this fact [8, 34, 79], though given the ubiquity of the Safe Browsing API—and especially given its inclusion in privacy-sensitive browsers such as Firefox and Safari—it is worth emphasizing.

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.



(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. For the new offline-online PIR scheme, which includes a relatively expensive offline cost, we also show the total cost, amortized over the number of queries that the client makes to the database. The new scheme requires an expensive setup phase but has lower per-query server-side time than prior PIR schemes.

6.3 Private Safe Browsing with Checklist

We design a private Safe-Browsing service based on Checklist, which uses our new two-server PIR scheme of Section 4. So, our scheme has the cost of requiring 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 (apart from 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.

Remaining privacy leakage. Using Checklist for the online Safe Browsing queries 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 servers learn 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 [30].

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. 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. [23] ("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 [14], 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 [52]. 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. Specifically, we consider a database of three million 32-byte records. Figure 3 presents the servers' and client's per-query CPU time and communication costs. 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 eleven-second *offline* phase, run once per client. Even with this

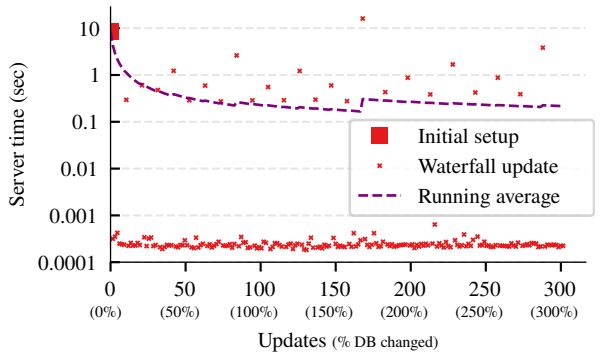


Figure 4: Server-side cost of client updates. At each time step, 1% of the three million records change. The waterfall update scheme reduces the average update cost by more than 45× relative to a naive solution of rerunning the offline phase on each change.

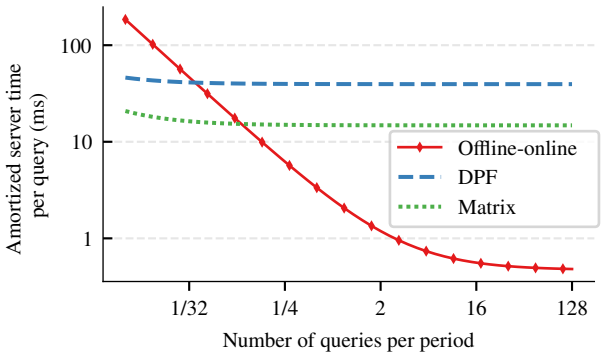


Figure 5: The amortized server compute costs of PIR queries on a database with updates. As the number of queries between each pair of subsequent database updates grows, the offline-online PIR scheme achieves lower compute costs compared to previous PIR schemes.

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. Next we evaluate the performance of the waterfall updates mechanism (Section 5). 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×.

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. In the beginning of each period, we apply a batch of $B = 500$ updates to the database, after which the client fetches an 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 per-update costs across queries in each period, and we average the costs across 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 forwards all the Safe Browsing requests it intercepts to Google's server. Since the browser maintains an updated list of partial hashes, we can measure the rate of updates to the database by observing the updates to the list of partial hashes that the browser downloads from the server. Furthermore, we can directly measure the query frequency. We run this tool on our personal laptop for a typical work week, using the instrumented browser for all browsing. Figure 6 shows the pattern of updates and

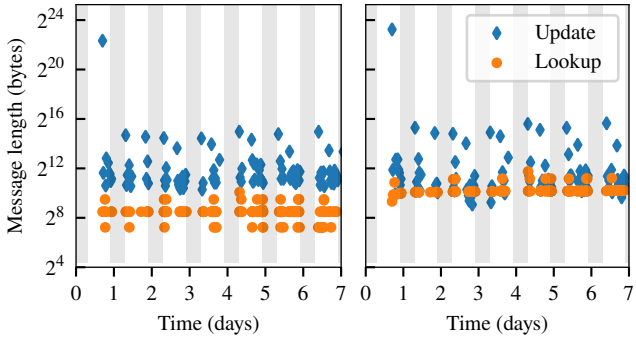


Figure 6: Communication on recorded trace (left) and on the same trace replayed with Checklist (right). Shaded regions are 10pm-7am each day.

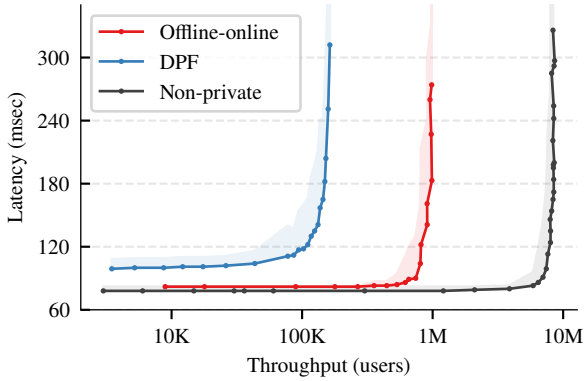


Figure 7: The performance of a Checklist server. Solid lines display the average latency, and shaded regions show the latency of the 95th-percentile of requests.

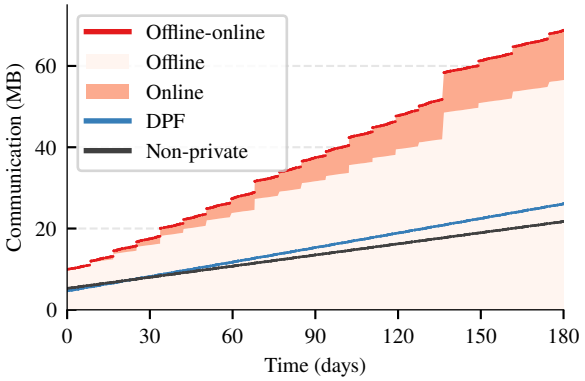


Figure 8: We repeatedly replay the trace of Safe Browsing queries and updates, recorded on a seven-day user session. The computational benefits of offline-online PIR come at a cost of more communication.

queries in the recorded trace and plots the combined request and response size for both updates and queries. The database size is roughly three million records, which, along with the rate of updates, is consistent with Google's public statistics on the Safe Browsing datasets [41]. 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 then measure the throughput and latency of the Checklist system. We setup three virtual machines on Google Cloud: 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. The generator sets the relative frequency of update and query requests, as well as the size of the updates, based on their values in the recorded trace. With the Checklist server under load, an additional client machine performs discrete Checklist lookups, and measures their end-to-end latency. The measured latency includes the time to generate the two queries, obtain the responses from the server, and process the responses on the client side. The load generator establishes a new TLS connection for each request to simulate different users more faithfully. 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.

To measure the communication costs for the client, we repeatedly replay our recorded one-week trace to simulate long-term usage of Checklist. Specifically, for each update request in the trace, we first use the information from the response to update the size of the database on the Checklist server, such that the database size evolves as in the recording. We then measure the cost of fulfilling the same update request using Checklist, which includes updating the list of partial hashes (as in the original implementation) and updating the client's PIR hint. For each lookup query in the trace, we issue a random query to the Checklist server and measure the associated costs. Figure 6 plots the size of the messages when using Checklist compared to the existing non-private implementation. Figure 8 shows the cumulative costs of using Checklist with two different PIR schemes, as well as with non-private lookups. The DPF-based PIR is communication efficient, using only 20% more communication than non-private lookups. Offline-online PIR has a more significant communication cost, 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.

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. The client can compress the list of partial hashes as an unordered set (e.g., using Rice-Golomb encoding [39]) to further reduce 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 (ordered) list of partial hashes and the latest hint.

We summarize our evaluation of Safe Browsing with Checklist in Table 9. We estimate that deploying Safe Browsing with Checklist using offline-online PIR would require 9.4× more servers than the non-private service with the same latency. A DPF-based PIR protocol would require 63× more servers than the non-private service and would increase the latency by 30ms, though it would use 2.7× less bandwidth on a running basis.

8 Related work

Certificate-revocation lists are an important type of blocklist used on the Internet today. CRLite [57], used in the Firefox browser today, gives a way to compress a blocklist of revoked

Table 9: Summary of costs of Safe Browsing with Checklist. The offline-online variant offers lower compute costs and latency, while a DPF based system is more communication efficient.

PIR type	Server costs	Latency	Client bandwidth	
			Initial	Running
	(servers per 1B users)	(ms)	(MB/user)	(MB/month)
Non-private	143	91	5.0	3.0
Offline-Online	1348	90	10.3	9.8
DPF	9047	122	5.0	3.6

certificates to ship to the client. CRLite's client-side storage grows linearly with the size of the blocklist, unlike Checklist. Revcast proposes broadcasting certificate-revocation information over FM radio [70]. Let's Revoke [74] proposes modifying the public-key infrastructure to facilitate revocation. Solis and Tsudik [75] 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 [48,58,61,77,80] 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. These systems use private-set-intersection protocols [19,20,67,68] to provide privacy for the client and server. Some of these breach-notification services [77] leak a partial hash of the user's password to the server [61]. Using Checklist for these applications would eliminate this leakage and would reduce the server-side computational cost (since our amortized lookup cost is sublinear in the blocklist size), at the price of requiring two non-colluding blocklist servers.

Our focus application of Checklist is to the Safe Browsing API. Prior work has demonstrated the privacy weaknesses of the Safe Browsing API [8,34], arising from the fact that the client leaks 32-bit hashes of the URLs it visits to the server.

The core of Checklist is a new offline/online private-information-retrieval (PIR) scheme. The body of work on PIR is vast and we will only be able to scratch the surface here. Chor et al. [22,23] initiated the study of PIR in which the client communicates with multiple non-colluding servers. Gasarch [33] 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 [29,31,81]. Under mild assumptions, we can now construct two-server PIR schemes with almost optimal communication cost [13,14,35,44]. An orthogonal goal is to protect against PIR server misbehavior [26,36].

A parallel line of work aims 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. [7] show that if the servers preprocess the database, they can respond to client queries in $o(n)$ time. Alternatively, "batch PIR" [45,50] 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 [62]. Other work relaxes the privacy guarantees of PIR to improve performance [78]. Our work builds on offline/online PIR protocols [25,66], in which the client fetches some information about the database in an offline phase to improve online performance.

Under appropriate "public-key assumptions" [28], it is possible [17,55] to construct PIR schemes in which the client communicates with only a single database server. Sion and Carbunar [73] 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 [64] 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 setting in which multiple servers are unavailable.

Finally, prior work has applied PIR to private media consumption [43], eCommerce [46], and private messaging [5].

9 Discussion and conclusion

Extension: 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 to password-breach notification services [48,58,61,77,80], hiding the blocklist from the client is important. That is, the *only* information that a client should learn about the blocklist after a single interaction with the server is whether its string appears on the blocklist.

Freedman, Ishai, Pinkas, and Reingold [32] 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.

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 servers. In practice, it can be difficult to deploy multi-party protocols at scale, since it require 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 [55], taking advantage of recent advances in lattice-based PIR schemes [2,3,4,5]. While prior work [25] shows that it is possible in theory to construct single-server offline/online PIR schemes with sublinear online server time, these schemes are orders of magnitude less efficient than our two-server schemes.

Outlook. With Checklist, a client can check a string against a server-side list of blocklisted strings, without revealing its string to the server and without having to download and maintain a local copy of the 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 to see major browsers integrate Checklist into their Safe Browsing clients, password-breach alerting systems, and certificate-revocation blocklists to better protect their users' privacy.

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 private-information-retrieval 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 [25], 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. This work was funded by NSF, DARPA, a grant from ONR, the Simons Foundation, a Facebook 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. *Proceedings on Privacy Enhancing Technologies*, 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] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: PIR with preprocessing. In *CRYPTO*, 2000.

[7] 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.

[8] 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[24] ClamAV. ClamAV Documentation: File hash signatures. `https://www.clamav.net/documents/file-hash-signatures`. Accessed 28 January 2021.

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

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

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

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

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

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

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

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

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

[34] 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.

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

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

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

[38] Google. Chrome speed: Binary size metrics. `https://chromium.googlesource.com/chromium/src/+/master/docs/speed/binary_size/metrics.md`. Accessed 19 January 2021.

[39] Google. Compression in Safe Browsing APIs (v4)). `https://developers.google.com/safe-browsing/v4/compression`. Accessed 19 January 2021.

[40] Google. Safe Browsing APIs (v4). `https://developers.google.com/safe-browsing/v4`. Accessed 19 January 2021.

[41] Google. Safe Browsing transparency report. `https://transparencyreport.google.com/safe-browsing/overview`. Accessed 28 January 2021.

[42] 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.

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

[44] 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. *Proceedings on Privacy Enhancing Technologies*, 2019(4):112–131, 2019.

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

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

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

[48] Troy Hunt. Have I been pwned. https://haveibeenpwned.com/FAQs. Accessed 26 January 2021.

[49] Internet Storm Center. SSL CRL activity. https://isc.sans.edu/crls.html. Accessed 28 January 2021.

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

[51] J. C. Jones. Design of the CRLite infrastructure. https://blog.mozilla.org/security/2020/12/01/crlite-part-4-infrastructure-design/, December 2020. Accessed 28 January 2021.

[52] Daniel Kales. Go DPF library. https://github.com/dkales/dpf-go, 2019. Accessed 26 January 2021.

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

[54] Scott Knight. syspolicyd internals. https://knight.sc/reverse%20engineering/2019/02/20/syspolicyd-internals.html, February 2019. Accessed 26 January 2021.

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

[56] Adam Langley. CRL set tools. https://github.com/agl/crlset-tools. Accessed 28 January 2021.

[57] 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.

[58] 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. Accessed 26 January 2021.

[59] Daniel Lemire. A fast alternative to the modulo reduction. https://lemire.me/blog/2016/06/27/a-fast-alternative-to-the-modulo-reduction/, 2016. Accessed 26 January 2021.

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

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

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

[63] 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. Accessed 26 January 2021.

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

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

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

[67] 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. Accessed 26 January 2021.

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

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

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

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

[72] 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.

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

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

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

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

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

[78] 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.

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

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

[81] 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 [25, 71].

Correctness. We say that a puncturable pseudorandom set $(\mathsf{Gen}, \mathsf{GenWith}, \mathsf{Eval}, \mathsf{Punc})$ is *correct* if for all $\mathsf{sk} \leftarrow \mathsf{Gen}()$ 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 $i \in [n]$, $\mathsf{sk} \leftarrow \mathsf{GenWith}(i)$, 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 all efficiency adversaries $\mathcal{A}$ the following quantity

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

is less than some fixed negligible function in the implicit security parameter $\lambda \in \mathbb{N}$, for large enough λ. The same must hold if we sample sk by choosing $i \xleftarrow{R} [n]$ and setting $\mathsf{sk} \leftarrow \mathsf{GenWith}(i)$.

B Security analysis

We begin by analyzing a single query of Construction 2.

Lemma 3 (Correctness)**.** *For every $\lambda, n \in \mathbb{N}$, every database $D \in \{0,1\}^n$, and every $i \in [n]$, the client succeeds in retrieving the ith bit of the database with all but negligible probability.*

Proof. Suppose first that the Query algorithm does not abort. Let D'_i be the bit output of Reconstruct. We consider the following three cases:

- Easy case ($\beta = 0$): it holds that $D'_i = b_t + u_1 \bmod 2$ where b_t is the tth hint bit and u_1 is the parity bit in the answer of the right server. Since we are in the easy case, it holds that $\mathsf{sk}_{\mathsf{p}1} = \mathsf{Punc}(\mathsf{sk}_t, i)$ and therefore $S_{\mathsf{p}1} = S_t \setminus \{i\}$ and $u_1 = \sum_{j \in S_{\mathsf{p}1}} D_j = \sum_{j \in S_t \setminus \{i\}} D_j = b_t + D_i \pmod 2$. Therefore $D'_i = b_t + u_1 \bmod 2 = D_i$.
- Hard case ($\beta = 1$): it holds that $D'_i = u_0 + u_1 + v_\gamma \bmod 2$ where u_0 and u_1 are the parity bits in the answers of the two servers and $v_\gamma = D_{w_\gamma}$ is the extra bit in the answer of the server γ. We have the following:

$$S_{\mathsf{p}\gamma} = S_{\mathsf{new}} \setminus \{i\}$$
$$S_{\mathsf{p}\bar{\gamma}} = S_{\mathsf{new}} \setminus \{w_\gamma\}$$

  and therefore $D'_i = u_\gamma + u_{\bar{\gamma}} + v_\gamma \bmod 2 = \sum_{j \in S_{\mathsf{new}} \setminus \{i\}} D_j + \sum_{j \in S_{\mathsf{new}} \setminus \{w_\gamma\}} D_j + D_{w_\gamma} \bmod 2 = D_i$.

Finally, observe that the Query algorithm only fails if none of $T = \lambda n$ pseudorandom set of size $\sqrt{n}$ contain element i, which happens with negligible probability [25, Appendix C.2]. □

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

Lemma 4 (Security)**.** *Suppose that the underlying puncturable pseudorandom set is secure. Then for every $\lambda, n \in \mathbb{N}$, every database $D \in \{0,1\}^n$, every server $s \in \{0,1\}$, and every $i, i' \in [n]$, $\mathsf{View}_{D,i,s} \approx_c \mathsf{View}_{D,i',s}$.*

Proof. Consider the following sequence of distributions.

$\underline{\mathsf{Hyb}_0} = \mathsf{View}_{D,i,s} = (\mathsf{sk}_{\mathsf{p}s}, w_s)$ is the view of server s when the client is reading index i.

$\underline{\mathsf{Hyb}_1} = (\mathsf{sk}_{\mathsf{p}s}, \tilde{w})$ where $\tilde{w} \xleftarrow{R} \mathsf{Eval}(\mathsf{sk}_{\mathsf{p}1})$.

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

$\underline{\mathsf{Hyb}_2}$: we modify the Query algorithm such that after checking that there exists a set $t \in [T]$ such that $i \in S_t$, it always sets $\mathsf{sk}_t \leftarrow \mathsf{GenWith}(n, i)$, rather than taking the key from the hint.

If $s = 0$, the two distributions are identically distributed since $\mathsf{sk}_{\mathsf{p}0}$ is always chosen independently of S_t. If $s = 1$, the distributions Hyb_1 and Hyb_2 are identical since choosing a set S_t from a collection of random puncturable sets, such that $i \in S_t$— conditioned on the collection containing at least one such set—is identical to sampling a new set with $\mathsf{GenWith}(n, i)$. If the collection does not contain such a set, we abort in both cases.

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

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

$\underline{\mathsf{Hyb}_4}$: we modify the Query algorithm such that if $\beta = 1$ and $\gamma = s$, it calls QueryEasy instead of QueryHard.

The distributions Hyb_3 and Hyb_4 are identical, since if $\beta = 1$ and $\gamma = s$, the procedure QueryHard sets $\mathsf{sk}_{\mathsf{p}s} \leftarrow \mathsf{Punc}(\mathsf{sk}_{\mathsf{new}}, i)$, which is identical to how QueryEasy sets $\mathsf{sk}_{\mathsf{p}s}$.

$\underline{\mathsf{Hyb}_5}$: we modify the Query algorithm to so that it samples $\beta \xleftarrow{R} \mathsf{Bernoulli}((\sqrt{n}-1)/n)$ (notice that the probability is one half of the original probability). If $\beta = 0$ it calls QueryEasy, and if $\beta = 1$, it calls QueryHard with $\gamma = \bar{s}$.

The distributions Hyb_4 and Hyb_5 are identical, since in eahc of them QueryHard is called with probability exactly $(\sqrt{n}-1)/n$, and always with with $\gamma = \bar{s}$.

We can now write Hyb_5 explicitly as follows:

$$
\begin{aligned}
&\mathsf{sk}_{\mathsf{new}} \xleftarrow{R} \mathsf{GenWith}(n, i)\\
&S_{\mathsf{new}} \xleftarrow{R} \mathsf{Eval}(\mathsf{sk}_{\mathsf{new}})\\
&\beta \xleftarrow{R} \mathsf{Bernoulli}((\sqrt{n}-1)/n)\\
&\text{if } \beta = 0\text{: } i_{\mathsf{punc}} \leftarrow i\\
&\text{else: } i_{\mathsf{punc}} \xleftarrow{R} S_{\mathsf{new}} \setminus \{i\}\\
&\mathsf{sk}_{\mathsf{p}1} \leftarrow \mathsf{Punc}(\mathsf{sk}_{\mathsf{new}}, i_{\mathsf{punc}})\\
&\tilde{w} \xleftarrow{R} S_{\mathsf{new}} \setminus \{i_{\mathsf{punc}}\}\\
&\mathrm{Output}(\mathsf{sk}_{\mathsf{p}1}, \tilde{w})
\end{aligned}
$$

Now, by Lemma 36 in [25], it holds that Hyb_5 when reading index i is computationally indistinguishable from Hyb'_5 when reading index i'. Therefore $\mathsf{Hyb}_0 = \mathsf{View}_{D,i,s}$ is computationally indistinguishable from $\mathsf{Hyb}'_0 = \mathsf{View}_{D,i',s}$, which completes the proof of the lemma. □

The extension to the multi-query case is identical to the proof of Lemma 45 in previous work[25].

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 the parent node. The simple length-doubling pseudorandom generator we use is based on fixed-key AES [42] to avoid 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. A 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 results in a 7× faster running time, compared to a depth-first implementation, which essentially encrypts a much larger number of "individual" blocks one by one.

Another implementation choice that reduces the puncturable-set evaluation time is to use "Lemire's trick" [59, 60] for mapping random 32-bit values to random integers in a specified range without using arithmetic modulo operations.

Finally, we use SIMD compiler intrinsics whenever we compute the XOR of a large number of database records.