Clubcards for the WebPKI: smaller certificate revocation tests in theory and practice

John Schanck
Mozilla
Email: jschanck@mozilla.com

***Abstract*—CRLite is a low-bandwidth, low-latency, privacy-preserving mechanism for distributing certificate revocation data. A CRLite aggregator periodically encodes the revocation status of a set of certificates into a compact static hash set, or *membership test*. These membership tests can be downloaded by clients and queried privately. We present a novel data-structure for membership tests, which we call *clubcard*, and we evaluate the encoding efficiency of clubcard using data from Mozilla's CRLite infrastructure.**

As of November 2024, the WebPKI contains over 900 million valid certificates and over 8 million revoked certificates. We describe an instantiation of CRLite that encodes the revocation status of these certificates in a 6.7 MB package. This is 54% smaller than the original instantiation of CRLite presented at the 2017 IEEE Symposium on Security and Privacy, and it is 21% smaller than the lower bound claimed in that work.

A sequence of clubcards can encode a dynamic dataset like the WebPKI. Using data from late 2024 again, we find that clubcards encoding 6 hour delta updates to the WebPKI can be compressed to 26.8 kB on average—a size that makes CRLite truly practical.

We have extended Mozilla's CRLite infrastructure so that it can generate clubcards, and we have added client-side support for this system to Firefox. We report on some performance aspects of our implementation, which is currently the default revocation checking mechanism in Firefox Nightly, and we propose strategies for further reducing the bandwidth requirements of CRLite.

1. Introduction

The WebPKI is a set of technologies, organizations, standards, and procedures that convey *trust* in TLS certificates.

The problem of distributing trust is largely solved. Web browsers and operating systems are packaged with sets of root certificates (*trust stores*) and with software for building and validating paths from TLS server certificates to those trust stores. Trust stores are small—just a few megabytes of data—and they are not modified frequently, so changes to trust stores can be incorporated into the regular software update cycle.

The problem of revoking trust is more subtle, as the number of server certificates far exceeds the number of roots. A full collection of certificate revocation lists (RFC 5280 CRLs [1]) for the WebPKI is on the order of a gigabyte. Worse yet, the WebPKI CRLs change rapidly: tens of thousands of certificates are added each day, and a similar number are removed due to expiry. It would be prohibitively expensive for a software vendor to ship a full set of CRLs even once in their update cycle—let alone to ship fresh CRLs regularly. It would also be prohibitively expensive for CAs if clients regularly downloaded CRLs from them directly. In 2014, Cloudflare estimated that one CA's monthly bandwidth bill increased by $400,000 USD [19] due to certificates revoked in response to Heartbleed.

Fortunately, clients do not need a full listing of all revoked certificates in order to check the revocation status of any particular certificate. The CRLite proposal by Larisch, Choffnes, Levin, Maggs, Mislove, and Wilson [12] encodes the full WebPKI revocation set into a compact hash set implemented as a *many-level cascade of Bloom filters*. In 2017, the authors of [12] estimated that CRLite could distribute revocation data to users with an initial 10 MB download followed by a 580 kB daily update. Mozilla implemented CRLite shortly after it was proposed, and the system has been available under an experimental feature flag since Firefox 69 [8], [11].

As the WebPKI has grown, so has the size of Bloom filter cascades and delta updates. By mid-2024, the average cascade published by Mozilla's CRLite service was approximately 18 MB; and daily delta updates were 800 kB. The service would publish a new cascade every 10 days, so Firefox users who enabled CRLite could expect to download 26 MB of revocation data over any 10 day period. This large bandwidth requirement is one reason that CRLite has never been enabled by default for all Firefox users.

At Real World Crypto 2022, Mike Hamburg presented *two-level cascades of frayed ribbon filters* as an alternative membership test for CRLite [10]. Hamburg's construction is nearly optimal, and he estimated that it would reduce the size of the initial download in CRLite by 40%.

In the present work, we describe another instantiation of CRLite using two-level cascades of ribbon filters. Using an unpublished idea of Hamburg (private communication [9]) we find that we are actually able to reduce the size of the initial download in CRLite by more than 50%. Hamburg's idea, specifically, is to partition the WebPKI into blocks–e.g. by issuer–and then to encode those blocks separately.

Our contributions are as follows:

1) We describe a novel data structure for membership tests that we call a *clubcard* (Section 3). Clubcards are two-level cascades enriched with metadata to describe a partition of the encoded set.
2) We describe an instantiation of clubcard for the WebPKI, and an instantiation of CRLite that uses clubcard-based membership tests (Section 4).
3) We describe the performance characteristics of our CRLite instantiation using real data from the WebPKI in late 2024 (Section 5). We consider two partitioning strategies: partitioning by issuer, and partitioning by issuer and hour-of-expiry. We generate issuer partitioned clubcards that are, on average, 54% smaller than Bloom filter cascades for the same data. We show that partitioning by issuer and hour-of-expiry could reduce size even further.
4) We evaluate our CRLite instantiation by adding client-side support for clubcards to Firefox Nightly. We find that we can deliver users a fresh view of the WebPKI revocation set with a 6.7 MB download every 22 days and a 26.8 kB download every 6 hours in-between. We find that our membership tests contain information about 93% of the certificates that Firefox Nightly users encounter. And we find that CRLite improves a key network performance metric by $\sim 15\%$, primarily by reducing the number of OCSP requests that Firefox performs.
5) We compare clubcard-based CRLite against several other revocation checking systems, and we outline some promising avenues for further improvements (Section 6).

While our focus is on the application of clubcards to CRLite, we provide a permissively-licensed (MPL 2.0) Rust library for constructing and querying arbitrary clubcards (https://github.com/mozilla/clubcard) which may have other applications. Our instantiation of clubcard for the WebPKI (https://github.com/mozilla/clubcard-crlite), as well as our improvements to Mozilla's CRLite implementation (https://github.com/mozilla/crlite), and Firefox (https://hg.mozilla.org/mozilla-unified) are also made freely available.

2. Preliminaries

2.1. Notation

We denote the set of r-element subsets of an n-element set U by $\binom{U}{r}$. The binary strings of length at most k are denoted $\{0,1\}^{\leq k}$. All logarithms are to base 2.

2.2. Asymptotics of binomial coefficients

The following lemma, which expresses $\log\binom{n}{r}$ in terms of the binary entropy function, $\mathrm{H}(p) := -p\log p - (1-p)\log(1-p)$, is useful for building intuition for the asymptotic behavior of binomial coefficients and the optimal size of membership tests.

Lemma 1.

$$\log\binom{n}{r} = n\,\mathrm{H}(r/n) + \mathrm{O}(\log n)$$

The error term can be reduced to $\mathrm{O}(\log r)$ in the case $r \ll n$.

Proof: Apply Stirling's approximation, $\log n! = n\log n - n\log e + \mathrm{O}(\log n)$, to each term in $\log\binom{n}{r} = \log n! - \log r! - \log(n-r)!$ and simplify. For the $r \ll n$ case refine the $\mathrm{O}(\log n)$ error term in Stirling's approximation to $\frac{1}{2}\log(2\pi n) + \mathrm{O}(1/n)$. □

The approximation given in [12] for the case of $r \ll n$ is

$$\log\binom{n}{r} \approx r\log n/r + 1.44r. \tag{1}$$

This can be derived from Lemma 1 by writing $n\,\mathrm{H}(r/n) = r\log(n/r) - (n-r)\log(1-r/n)$ and expanding $-\log(1-r/n) = (\log e)\, r/n + \mathrm{O}(r^2/n^2)$.

2.3. Lower bounds on set encodings

Carter, Floyd, Gill, Markowsky, and Wegman initiated the study of membership tests in [2] with the question: "how much space is needed to represent a set". We recall their lower bound and state some simple corollaries of it.

Generic subset encoders. Suppose that $\mathrm{enc}_r : \binom{U}{r} \to \{0,1\}^{\leq k}$ is injective, i.e. suppose that enc_r unambiguously encodes the r-element subsets of U as binary strings of length at most k. Since there are only $2^{k+1}-1$ such binary strings, we have $k+1 \geq \log\left(\binom{n}{r}+1\right)$. In other words, there exist R for which $\mathrm{enc}_r(R)$ is $\Omega\left(\log\binom{n}{r}\right)$ bits in length. The same counting argument establishes that the average length of $\mathrm{enc}_r(R)$, over a uniform choice of R, must also be $\Omega\left(\log\binom{n}{r}\right)$.

Type-aware subset encoders. There is no non-trivial lower bound on the bit-length of $\mathrm{enc}_r(R)$ when R is fixed. But we can often derive a non-trivial lower bound when R is drawn from a specific, non-uniform, distribution.

Suppose that $\pi = \{U_i\}_i$ is a partition of U into non-empty blocks of size $n_i = |U_i|$. Let $r_i = |R \cap U_i|$ be the number of elements of R that lie in block i. We say that the ordered list $\vec{r} = (r_i)_i$ is the *type* of R with respect to the partition π. A *type-aware encoder* is some function $\mathrm{enc}^{\pi}_{\vec{r}} : \prod_i \binom{U_i}{r_i} \to \{0,1\}^{\leq k}$ that unambiguously encodes type $\vec{r}$ subsets of U as binary strings of length at most k. Using the same counting argument as before, we have $k = \Omega\left(\sum_i \log\binom{n_i}{r_i}\right)$.

From Lemma 1 we see that a type-aware encoder must use at least

$$\sum_i \left(n_i\,\mathrm{H}(r_i/n_i) + \mathrm{O}(\log n_i)\right) \tag{2}$$

bits for some set in its domain. The strict concavity of the entropy function suggests that this quantity may be smaller than $n\,\mathrm{H}(r/n)$ for some partitions.

2.4. Linear algebraic membership tests

A linear algebraic membership test is a set encoding that admits an efficient membership query that involves evaluating one or more linear functions. In this section, we recall some basic concepts that underlie state-of-the-art linear algebraic membership tests.

k-bit retrieval functions. Consider an n-element universe U and a function $b : U \to \mathbb{F}_2^k$ that associates k bits of information with each element of U. There is a well-known way to encode b using linear algebra [3], [14], [18], [20]. Let h be a hash function that, for some positive integer d, maps elements of U to subsets of $[0, d-1]$. We can view $h(u)$ (or, more properly, its indicator function) as a d-dimensional vector over $\mathbb{F}_2$. Let $\mathbf{H}$ denote the $n \times d$ matrix $\left(h(u)\right)_{u \in U}$ and let $\mathbf{B}$ denote the $n \times k$ matrix $\left(b(u)\right)_{u \in U}$. If some $d \times k$ matrix $\mathbf{X}$ satisfies $\mathbf{H} \cdot \mathbf{X} = \mathbf{B}$, then any party with knowledge of h and $\mathbf{X}$ can retrieve the k bit value $b(u)$ by computing $h(u) \cdot \mathbf{X}$. The pair $(h, \mathbf{X})$ is a *k-bit retrieval function*.

Exact membership tests. In the case where R is some subset of U and $b(u) = 0$ if and only if $u \in R$, a one-bit retrieval function $(h, \mathbf{X})$ that encodes b is an *exact membership test* for R. Note that one-bit retrieval functions are not necessarily compact: the vector $\mathbf{X}$ is at least $d \geq n$ bits, which may be much larger than the lower bound (Equation 1) when $|R|$ is small.

Approximate membership query filters. An *approximate membership query (AMQ) filter* for R perfectly encodes some set S that contains R. With $r = |R|$ and $s = |S|$, the *false positive rate* of an AMQ filter is $(s-r)/(n-r)$. Dietzfelbinger and Pagh observed that AMQ filters can be constructed from linear retrieval functions [3]. With the same setup as above, if $\mathbf{X} \in \mathbb{F}_2^{d \times k}$ solves the system

$$\left(h(u) \cdot \mathbf{X} = \mathbf{0}^{n \times k}\right)_{u \in R} \tag{3}$$

then $(h, \mathbf{X})$ is an AMQ filter for R. The false positive rate of $(h, \mathbf{X})$ is, heuristically, equal to 2^{-k}. The rationale being that if $\mathbf{X}$ were uniformly random, then for any fixed $u \in U \setminus R$ with nonzero $h(u)$ we would have $\Pr[h(u) \cdot \mathbf{X} = 0] = 2^{-k}$. This intuition can be made rigorous for some choices of d and h, in particular those satisfying Theorem 2 below.

Ribbon retrieval functions. A ribbon retrieval function is a k-bit retrieval function where h has a special form. Specifically, there is a constant w (the *ribbon width*) and a function $s : U \to [0, d-w-1]$ (the *offset* function) such that $h(u) \subseteq [s(u), s(u)+w-1]$. In other words, each $h(u)$ is clustered in a width-w subinterval of $[0, d-1]$.

The following theorem shows that ribbon retrieval functions can be used to efficiently construct compact membership tests and AMQ filters. The theorem first appears as Theorem 2 in [4], but our presentation is closer to Theorem 3.1 of [6].

Theorem 1 (Thm. 3.1 of [6]). For any constant $0 < \epsilon < \frac{1}{2}$, if $w = (\log n)/\epsilon$ and $n/d = (1-\epsilon)$, then with high probability the linear system $(h(u) \cdot \mathbf{X} = b(u))_{u \in U}$ is solvable for any $k \in \mathbb{N}$ and any $b : U \to \mathbb{F}_2^k$. Moreover, after sorting $(h(u))_{u \in U}$ by $s(u)$, Gaussian elimination can compute a solution $\mathbf{X}$ in expected time $\mathrm{O}\big(n/\epsilon^2\big)$.

An AMQ filter $(h, \mathbf{X})$ that is constructed in alignment with Theorem 1 with $b(u) = 0$ for all u is called a *homogeneous ribbon filter*. The following theorem establishes the additional conditions on ϵ and w that are necessary for a homogeneous ribbon filter to have a small false positive rate.

Theorem 2 (Thm 3. of [5]). For any $k \in \mathbb{N}$ and $0 < \epsilon < \frac{1}{2}$, then there is a $w \in \mathbb{N}$ with $\frac{w}{\max(k, \log w)} = \mathrm{O}\big(1/\epsilon\big)$ such that a homogeneous ribbon filter with ribbon width w has false positive rate $\approx 2^{-k}$.

Constructing ribbons. A series of works culminating in [6] describe an "on-the-fly" variant of Gaussian elimination that is compatible with Theorem 1. We recall the algorithm here, as we will use a variant of it in Section 3.

When h has the special form required for a ribbon retrieval function, we can describe $h(u)$ by some tuple $(s(u), c(u))$ where $s \in [0, d-w-1]$, $c \subseteq [0, w-1]$, and

$$h(u) = \{s(u) + i : i \in c(u)\}.$$

There may be more than one representations of $h(u)$ in this form, so it is useful to single out a canonical choice. To that end, we can define the *aligned representative* of a non-empty $h(u)$ to be the unique value $(s^*(u), c^*(u))$ compatible with $s^*(u) = \min h(u)$. (If one prefers to think of $h(u)$ as a row vector, then $s^*(u)$ is the index of its leftmost 1 and $c^*(u)$ is the block of w coefficients starting at index $s^*(u)$.) In the special case of $h(u) = \emptyset$ we define $s^*(u) = 0$.

A linear system $(h(u) \cdot \mathbf{X} = b(u))_{u \in U}$ can be represented by the set of triples $\{(s(u), c(u), b(u)) : u \in U\}$. If the corresponding $s^*(u)$ values are distinct, i.e. if the system is upper triangular, then these triples can be represented by an array, M, of length d where $M[s^*(u)] = (c^*(u), b(u))$. Such a system can be solved in linear time by back-substitution.

In practice, the $s^*(u)$ are unlikely to be distinct. However, we may be able to find an equivalent upper-triangular system of equations using Gaussian elimination. The ReducedSystem algorithm below does just that. It is essentially the Ribbon algorithm from [6], but it additionally keeps track of equations that fail the insertion step.

```
ReducedSystem$(U, h, b)$:
  1) Let $M$ be an array of length $d$ where each entry is initialized to $(\emptyset, 0)$.
  2) Let $F = \emptyset$.
  3) For $u \in U$:
  4)    Let $(s, c)$ be the aligned representative of $h(u)$.
  5)    Call InsertEquation$(M, s, c, b(u))$.
  6)    If the call to InsertEquation returned inconsistent, add $u$ to $F$.
  7) Return $M$, $F$.
```

```
InsertEquation$(M, s, c, b)$:
  1) If $(c, b) = (\emptyset, 0)$, return redundant.
  2) If $(c, b) = (\emptyset, 1)$, return inconsistent.
  3) Let $(c_s, b_s) = M[s]$.
  4) If $c_s = \emptyset$, set $M[s] = (c, b)$ and return inserted.
  5) Replace $(s, c)$ with the aligned representative of $(s, c \oplus c_s)$ and replace $b$ with $b \oplus b_s$.
  6) Call InsertEquation$(M, s, c, b)$.
```

A detailed analysis of InsertEquation can be found in [6] (where it is labeled Algorithm 1).

Two-level cascades. Hamburg's two-level cascade construction [10] builds an exact membership test from a pair of linear algebraic retrieval functions. The "levels" of the cascade are (1) an AMQ filter that encodes a set S that contains R, and (2) a one-bit retrieval function that encodes R as a subset of S. We summarize the construction as the TwoLevelCascade$_k$ algorithm below. We leave k as a free parameter, but one should think of k as being $\lfloor \log((n-r)/r) \rfloor$ so that $|S| = \mathrm{O}(|R|)$.

```
TwoLevelCascade$_k(U, R)$:
  1) Construct a AMQ filter $(h, \mathbf{X})$ for $R$ with false positive rate $2^{-k}$.
  2) Enumerate the set $S = \{u \in U : h(u) \cdot \mathbf{X} = 0\}$.
  3) Construct a one-bit retrieval function $(g, \mathbf{y})$ encoding $R \subseteq S$.
  4) Output $(h, g, \mathbf{X}, \mathbf{y})$.
```

To see that $(h, g, \mathbf{X}, \mathbf{y})$ is a membership test for R, note that any party with knowledge of this tuple can determine whether some u is in R by checking whether both $h(u) \cdot \mathbf{X}$ and $g(u) \cdot \mathbf{y}$ are equal to zero.

Since $(h, \mathbf{X})$ has a false positive rate of 2^{-k}, the set S is of expected size $r + (n-r)/2^k$. If we instantiate TwoLevelCascade$_k$ using ribbon filters with parameters w and ϵ that are compatible with Theorems 1 and 2, and we take $k = \lfloor \log((n-r)/r) \rfloor$, the pair $(\mathbf{X}, \mathbf{y})$ is of expected size

$$(1+\epsilon)(rk + r + (n-r)/2^k). \quad (4)$$

This compares favorably with Eq. 1. In fact, by numerical evaluation, we find that the ratio of Equation 4 and $\log \binom{n}{r}$ is no more than $1.11 \cdot (1+\epsilon)$ for $r < n/2$. In any membership test, we can choose to encode R or $U \setminus R$ by keeping track of one bit of additional metadata. As such, $1.11 \cdot (1+\epsilon)$ bounds the overhead for all r. Advanced constructions of ribbon retrieval functions, e.g. the Bumped ribbons of [5] or the Frayed ribbons of [10], can achieve $\epsilon < 0.01$. This suggests that TwoLevelCascade can produce membership tests that are within $\approx 12\%$ of the optimal size.

3. The clubcard datastructure

A *clubcard* is a membership test for a *partitioned universe* based on Hamburg's two-level cascade construction. Partitioning allows clubcards to be type-aware in the sense of Section 2.3, and this allows clubcards to be significantly smaller than generic set encodings.

Throughout this section, I is an arbitrary index set for the blocks of a partitioned universe $U \dashv \{U_i\}_{i \in I}$. The elements of I are the *block identifiers*. The encoded set is denoted R. The portion of the encoded set that lies in block i is denoted $R_i = U_i \cap R$.

Formally, a clubcard is a 6-tuple (InUniverse, BlockId, BlockMeta, U, $\mathbf{X}$, $\mathbf{y}$). The U term is a description of the universe U and the partition $\{U_i\}_{i \in I}$. The $\mathbf{X}$ and $\mathbf{y}$ terms are matrices over $\mathbb{F}_2$. The remaining components are algorithms with the properties below.

- InUniverse(U, u): takes a bitstring u and returns `true` if u describes[1] an element of U and `false` otherwise.
- BlockId(U, u): takes a bitstring u that describes an element of U and returns the index $i \in I$ of the block U_i that contains that element. (Optionally, this function can return $\perp$ to signal that block i is not encoded.)
- BlockMeta(i): takes an index $i \in I$ and returns the metadata necessary for evaluating R_i-membership. This includes a pair of hash functions $h_i : U_i \to \operatorname{dom} \mathbf{X}$ and $g_i : U_i \to \operatorname{dom} \mathbf{y}$ as well as a (small) set of exceptional elements $F_i \subset U_i \setminus R_i$.

To determine whether some bitstring u describes an element of R, one runs the following algorithm.

```
Query(U, $u$):
  1) If $\neg$ InUniverse(U, $u$), return not in universe.
  2) Let $i \leftarrow$ BlockId(U, $u$).
  3) If $i = \perp$, return no data.
  4) Let $(h_i, g_i, F_i) \leftarrow$ BlockMeta$(i)$.
  5) If $h_i(u) \cdot \mathbf{X} \neq 0$, return non-member.
  6) If $g_i(u) \cdot \mathbf{y} \neq 0$, return non-member.
  7) If $u \in F_i$, return non-member.
  8) Return member.
```

3.1. Constructing clubcards

We can generate the values h_i, g_i, $\mathbf{X}$, and $\mathbf{y}$ that appear in Query by "stitching" several ribbon filters together. We

1. It is often useful for u to contain more information than is strictly necessary to represent an element of U, so we do not require the bitstrings accepted by InUniverse to be one-to-one with elements of U.

will first describe a simplified version of this stitching procedure that uses the TwoLevelCascade algorithm from Section 2.4 as a black box. We then describe some optimizations that can be made by unpacking TwoLevelCascade.

Warm-up: block diagonal stitching. Define the *rank* of block i to be

$$k_i := \begin{cases} 0, & \text{if } r_i = 0 \text{ or } r_i \geq n_i/2 \\ \lfloor \log((n_i - r_i)/r_i) \rfloor, & \text{otherwise,} \end{cases} \tag{5}$$

and let $k = \max_i k_i$. Without loss of generality, we can assume that the block identifiers are presented in descending order by rank, so $k = k_1$. This will simplify our indexing and provide some space-saving opportunities later.

For each $i \in I$ let $(h'_i, g'_i, \mathbf{X}^{(i)}, \mathbf{y}^{(i)}) = \mathsf{TwoLevelCascade}_{k_i}(U_i, R_i)$. For the purpose of this warm-up, we will assume that all of the implied ribbon filters are constructed without error, so we will have $F_i = \emptyset$ for all i. We would like to define a set of hash functions $\{h_i\}_i$ and a matrix $\mathbf{X}$ such that $h_i(u) \cdot \mathbf{X} = h'_i(u) \cdot \mathbf{X}^{(i)}$ for all i and $u \in U_i$. Toward that end, define $\mathbf{X}$ to be the vertical concatenation of the $\mathbf{X}^{(i)}$ (right-padding $\mathbf{X}^{(i)}$ to k columns with zeros if necessary). The matrix $\mathbf{X}^{(i)}$ has d_i rows and is embedded at row δ_i in $\mathbf{X}$ where $\delta_i = \sum_{j<i} d_j$. We obtain a satisfactory h_i by simply "shifting" the entries of $h'_i(u)$ by δ_i positions:

$$h_i(u) = \{\delta_i + j : j \in h'_i(u)\}.$$

To see that $h_i(u) \cdot \mathbf{X} = h'_i(u) \cdot \mathbf{X}^{(i)}$, let $\mathbf{x}$ to be any column vector of $\mathbf{X}$, and let $\mathbf{x}^{(i)}$ to be the corresponding column of $\mathbf{X}^{(i)}$. We have

$$\begin{aligned} h_i(u) \cdot \mathbf{x} = \sum_{j \in h_i(u)} \mathbf{x}_j &= \sum_{j \in h'_i(u)} \mathbf{x}_{\delta_i + j} \\ &= \sum_{j \in h'_i(u)} \mathbf{x}^{(i)}_j = h'_i(u) \cdot \mathbf{x}^{(i)}. \end{aligned} \tag{6}$$

The construction of the hash functions g_i and the vector $\mathbf{y}$ is identical, as is the proof that

$$g_i(u) \cdot \mathbf{y} = g'_i(u) \cdot \mathbf{y}^{(i)}. \tag{7}$$

From Equations 6 and 7 and the correctness of TwoLevelCascade we have that the following are equivalent: (1) $u \in R$, (2) $h'_i(u) \cdot \mathbf{X}_i = 0 \ \wedge \ g'_i(u) \cdot \mathbf{y}_i = 0$, (3) $h_i(u) \cdot \mathbf{X} = 0 \ \wedge \ g_i(u) \cdot \mathbf{y} = 0$.

Stitching ribbons. In the above construction, the matrix $\mathbf{X}^{(i)}$ is obtained by solving some upper triangular system $M^{(i)}$ which is output by ReducedSystem(R_i, h'_i, b). Rather than define $\mathbf{X}$ as the concatenation of the $\mathbf{X}^{(i)}$, we could define M as the concatenation of the $M^{(i)}$ and define $\mathbf{X}$ as a solution to the system described by M. In doing so, we can see that the offsets, $\delta_i = \sum_{j<i} d_j$, used above are slightly suboptimal: there may be empty rows (entries of the form $(\emptyset, 0)$) at the end of $M^{(i)}$ which could have been removed without impacting the solubility of the system. Every row of M adds at least one bit to the size of $\mathbf{X}$, so removing these empty rows would reduce the size of our membership tests.

The process of removing empty rows from $M^{(i)}$ and concatenating $M^{(i)}$ and $M^{(i+1)}$ is an example of *stitching* $M^{(i)}$ and $M^{(i+1)}$. In full generality, a tail of t equations can be removed from $M^{(i)}$ and stitched into $M^{(i+1)}$ by calling InsertEquation($M^{(i+1)}, j, M^{(i)}[d_i - t + j - 1]$) sequentially with j ranging from 0 to t. If any insertion step returns `inconsistent`, the changes to $M^{(i)}$ and $M^{(i+1)}$ can either be rolled back, or the corresponding set element can be added to F_i. This process can be attempted for several values of t to maximize the number of stitched equations. Equations that are successfully moved from $M^{(i)}$ to $M^{(i+1)}$ can be deleted from $M^{(i)}$.

The hash functions h_i are defined similarly to the block-diagonal stitching case above. The only difference is that the δ_i offsets record the position of $M^{(i)}$ in M rather than the position of $\mathbf{X}^{(i)}$ in $\mathbf{X}$.

Handling insertion errors. As observed in [6], the calls to InsertEquation(M, s, c, b) made in the construction of a homogeneous ribbon filter never fail, since each call starts with $b = 0$. Likewise, when constructing $\mathbf{y}$, we can insert elements of R (which have $b = 0$) before inserting elements of $S \backslash R$. This guarantees that any failure that occurs involves an element of $S \setminus R$ and not an element of R. The set of insertion failures, F_i, that occurred while constructing $y^{(i)}$, or as part of stitching, is stored in BlockMeta(i) and checked in Query.

Omitting zeros. Because we have assumed that the block identifiers are sorted by descending value of rank, the matrix $\mathbf{X}$ will have a staircase-shaped region of zero coefficients in its bottom right corner. These zero coefficients do not need to be encoded explicitly.

Rank zero blocks. If block i has rank zero ($k_i = 0$) then $h_i(u) \cdot \mathbf{X} = 0$ for all u in U_i. This means that $|S_i| = |U_i|$. In this case, it may be more efficient to encode $U_i \setminus R_i$ than to encode R_i. Our implementation handles this case by conditionally exchanging the `non-member` and `member` returns in Query based on an `inverted` flag stored in BlockMeta(i). The `inverted` flag also enables a compact encoding of empty blocks. During construction, if $|R_i| = 0$ we set $k_i = 0$ (Eq. 5) and we set the `inverted` flag in BlockMeta(i). The result is that block i occupies zero rows of $\mathbf{X}$ and $\mathbf{y}$, and yet Query(`U`, u) still correctly returns `non-member` for all $u \in U_i$.

Putting it all together. The full clubcard construction is as follows.

1) Select hash functions h'_i.
2) Decide whether to encode R_i or $U_i \setminus R_i$ and record the `inverted` flag in BlockMeta(i).
3) Construct the upper triangular systems $M^{(i)}$.
4) Stitch the $M^{(i)}$ together to form the upper triangular system M, and record the offsets necessary to define h_i in BlockMeta(i).

5) Solve M by back-substitution to obtain $\mathbf{X}$.
6) Enumerate the sets $S_i = \{u \in R_i : h_i(u) \cdot \mathbf{X} = 0\}$.
7) Select hash functions g'_i.
8) Construct the upper triangular systems $N^{(i)}$ by inserting all elements of R_i *before* inserting any elements of $S_i \setminus R_i$. Record any failure to insert an element of $S_i \setminus R_i$ in the F_i component of $\mathsf{BlockMeta}(i)$.
9) Stitch the $N^{(i)}$ to form the upper triangular system N, and record the offsets necessary to define g_i in $\mathsf{BlockMeta}(i)$.
10) Solve N by back substitution to obtain $\mathbf{y}$.

Lines 5 and 10 take linear time in $|R|$ and $|S|$ respectively. All of the other steps can be performed in parallel over the blocks of the partition. In particular, the Gaussian elimination steps, which take time $O(r_i/\epsilon^2)$ and $O(s_i/\epsilon^2)$, can be performed in parallel.

4. Clubcards for the WebPKI

We can now describe our instantiation of clubcards for the WebPKI revocation set. Specifically, we describe the choice of the hash functions h'_i and g'_i as well as the description of the universe and a partition of that universe. We begin by recalling a few details about the CRLite aggregator role in order to motivate our choices.

4.1. CRLite aggregator

Our CRLite implementation largely follows [12]. The *aggregator* monitors a dynamic collection of RFC 6962 [13] certificate transparency logs (CT logs) to discover known, non-expired, certificates. Each entry in a CT log has (1) a timestamp and (2) a certificate chain. The entries are ordered—each entry has a well-defined position, or index, in the log—but they are not necessarily ordered by timestamp. An entry's timestamp is only a promise that the entry will be publicly visible in the log before the end of the next "merge window." The length of a merge window is called the *maximum merge delay* (MMD) of the log, and is typically 24 hours.

We denote the end-entity certificate at index j in log i by $\text{cert}_{i,j}$ and the corresponding issuing certificate by $\text{issuer}_{i,j}$. Upon reading index j in log i, our CRLite aggregator stores the following data in persistent storage:

- `Issuer SPKI Hash`: the SHA-256 hash of the DER encoded RFC 5280 Subject Public Key Info [1] of $\text{issuer}_{i,j}$.
- `Serial Number`: the DER encoded RFC 5280 Certificate Serial Number [1] of $\text{cert}_{i,j}$ with the tag and length octets removed.
- `Not After Hour`: the expiry date of $\text{cert}_{i,j}$ truncated to the hour.

Our aggregator also tracks of the following data about each CT log that it monitors:

- `Min Index`, `Max Index`: The smallest and largest indices read from the log.
- `Min Timestamp`, `Max Timestamp`: The smallest and largest timestamps observed.
- `MMD`: The maximum merge delay of the log.

This is all of the data that our aggregator stores about known certificates and CT logs. Our aggregator obtains a list of revoked certificates by periodically fetching CRLs that have been disclosed to the Common CA Database in accordance with Mozilla Root Store Policy.[2]

4.2. Clubcard universe and partition metadata

When constructing a clubcard, our aggregator takes the universe U to be the set of tuples (`IssuerSPKIHash`, `SerialNumber`) corresponding to known, non-expired, certificates that it observed in CT logs. It describes U by a bitstring `U` that encodes

1) A list of timestamp intervals described by tuples of the form
$$(i, \texttt{MinTimestamp}_i, \texttt{MaxTimestamp}_i)$$
where i is the RFC 6962 LogID of a CT log.
2) A list of distinct `Issuer SPKI Hash` values.

We say that `U` *covers* a timestamp t from log i if the timestamp interval for log i in `U` satisfies

$$\texttt{MinTimestamp}_i + \texttt{MMD}_i \le t \le \texttt{MaxTimestamp}_i - \texttt{MMD}_i.$$

The $\texttt{MMD}_i$ offsets are necessary because, as mentioned above, timestamps may not increase monotonically with the log index.[3]

The inputs to Query are `U` and a bitstring u that describes a certificate. The exact encoding of u is immaterial, but it must include (1) `Issuer SPKI Hash`, (2) `Serial Number`, and (3) one or more log id and timestamp tuples. The implementations of $\mathsf{InUniverse}$ and $\mathsf{BlockId}$ are as follows.

$\underline{\mathsf{InUniverse}(\texttt{U}, u)}$:

1) Parse `IssuerSPKIHash`, `SerialNumber`, and `Timestamps` from u.
2) Return `true` if `U` covers a timestamp in `Timestamps` and `false` otherwise.

2. This is a change from [12], which pre-dated the disclosure requirement in Section 4.1 of Mozilla's root store policy [17]. Mozilla's earlier instantiation CRLite discovered CRL distribution points through certificate transparency logs. This earlier CRLite instantiation was severly limited by the fact that some CAs, including Let's Encrypt, were only publishing revocations through OCSP.

3. We would like to define coverage in terms of $\texttt{MinIndex}_i$ and $\texttt{MaxIndex}_i$, but the signed certificate timestamps (RFC 6962 SCTs [13]) that clients use to check coverage do not include the index of the certificate. The Static Certificate Transparency API [22], [23] will change this.

```
BlockId(U, $u$):
  1) Parse IssuerSPKIHash and SerialNumber from $u$.
  2) Return $\perp$ if Issuer SPKI Hash does not appear in U.
  3) Return Issuer SPKI Hash.
```

Note that we could reject unrecognized issuer SPKI hashes in InUniverse, but performing the test in BlockId gives us a return code that differentiates this case from a coverage failure. This is useful if one needs to selectively exclude an issuer from a clubcard.

4.3. The hash functions

The hash function h_i is encoded in BlockMeta(i) as an offset s and a modulus m for the CRLiteRibbonHash algorithm below. The hash function g_i is encoded similarly.

```
CRLiteRibbonHash($u, s, m$):
  1) Parse IssuerSPKIHash, SerialNumber, and Timestamps from $u$.
  2) Let $a$ be SHA256(IssuerSPKIHash || SerialNumber).
  3) Let $t$ be the high 64 bits of $a$ as an unsigned little-endian integer.
  4) return $(s + (t \bmod m), a)$
```

While h_i and g_i will use different values of s and m, the a component of the output of CRLiteRibbonHash(u, s, m) depends only on u. As such, the SHA256 evaluation can be cached while evaluating $h_i(u)$ and re-used while evaluating $g_i(u)$. The a value can also be re-used in queries to several distinct WebPKI clubcards.

In the language of ribbon retrieval functions, CRLiteRibbonHash(u, s, m) has a ribbon width of $w = 256$ and an offset in $[s, s + m + 256 - 1]$.

Theorems 1 and 2 are expressed in asymptotic terms and are not sufficient to determine a concrete value of ϵ compatible with $w = 256$. In practice, we have had good results using a fixed value of $\epsilon = 0.02$, which is approximately $(\log w)/w$. We take $m = (1 + \epsilon) |R_i|$ for h_i and $m = (1+\epsilon) |S_i|$ for g_i. We use $s = 0$ while constructing the linear systems for each block (i.e. for the hash functions h'_i and g'_i), and we update s after stitching.

4.4. Delta updates

Given two snapshots of the WebPKI $(U^{(1)}, R^{(1)})$ and $(U^{(2)}, R^{(2)})$, our aggregator encodes the change in the revocation set between the two snapshots as a clubcard for $R^{(2)} \setminus R^{(1)}$ as a subset of $U^{(2)}$.

Some care must be taken when combining the results from querying multiple clubcards. Given a set of clubcards encoding a full snapshot and any number of delta updates, one can determine the revocation status of a certificate by running Query on each clubcard and returning the result with the highest precedence. The precedence order on return codes is `member` > `non-member` > `no data` > `not in universe`.

5. Evaluation

We have added support for clubcard-based CRLite to Mozilla's CRLite backend and to Firefox. Clubcard-based CRLite is the default revocation checking mechanism for Firefox Nightly users as of version 134. In this section we report on various characteristics of the system that we have observed through telemetry and experiments with these users. All of information reported here was collected in accordance with Mozilla's Data Privacy Principles [15] and the Mozilla Privacy Policy [16].

Coverage. We have found that approximately 93% of all revocation checks performed by Firefox Nightly users are performed with CRLite. Of the remaining revocation checks, 5% are for certificates that are not covered by the clubcards that the user has downloaded, and 2% are performed by users that have not downloaded any clubcards.[4] We do not have a detailed breakdown of the 5% of uncovered certificates, but these are likely a combination of new certificates and certificates from private CAs.

Firefox has several revocation checking mechanisms that it will attempt to use if CRLite is not enabled. In priority order, Firefox will use a stapled OCSP response (28% of revocation checks), a cached OCSP response (55% of revocation checks), or it will perform a synchronous OCSP request (17% of revocation checks). With CRLite enabled, the percentage of revocation checks serviced by each of these mechanisms drops to 2%, 3%, and 2% respectively.

Performance. Our CRLite aggregator is able to generate a clubcard, and verify the revocation status of every certificate in the WebPKI, in approximately 300 seconds on a Ryzen 3975WX with 64GB of RAM.

Querying a collection of clubcards involves a single SHA256 evaluation and a small number of memory accesses and arithmetic operations that grows linearly with the number of clubcards in the collection. The process takes approximately 10 microseconds per clubcard on a wide range of CPUs.

In an experiment in Firefox 130, we found that enabling CRLite decreased average TLS handshake time from 172.4 ms to 143.4 ms (-16.8%). The improvement primarily comes from avoiding synchronous OCSP requests.

Membership test and delta update size. The sizes of membership tests and delta updates published by Mozilla's CRLite infrastructure in October and November 2024 can be found in Figures 1 and 2 respectively.

Taking data from October 31 as an example, Mozilla's CRLite backend observed 903 million live certificates and 8.7 million revoked certificates. It generated a 7.0 MB clubcard for this set. It also generated delta updates every 6 hours in the week that followed. The average uncompressed

4. Firefox distribution binaries do not currently include a clubcard. Some users also disable the Remote Settings mechanism that is used to distribute clubcards.

size of one of these delta updates was 95.5 kB; the minimum size was 89.7 kB; and the maximum size was 109.5 kB.

The metadata in a clubcard is somewhat compressible. The average gzip-compressed size of one of the 28 delta updates was 54.3 kB. This corresponds well with the amount of data that was actually downloaded by Firefox clients, as the CDN that serves these files is configured to use gzip compression.

The metadata in a delta update clubcard also has significant overlap with the metadata in the clubcards that preceded it, and this can be leveraged for even better compression factors. Using zstandard compression (at the -19 compression level) with the October 31 full snapshot clubcard as a dictionary, we were able to compress the delta updates to an average size of 26.0 kB. Firefox does not yet support the use of zstd compression for clubcards, but support is planned.

Partitioning and optimality. While there is no lower bound on the size of an encoding of the WebPKI revocation set, we can consider a clubcard with a fixed partitioning strategy to be a type-aware encoder in the sense of Section 2. Figures 1 and 2 report lower bounds for the size of 1) generic encodings, 2) type-aware encodings using partitioning by issuer, and 3) type-aware encodings using partitioning by issuer and hour of expiry.

The sizes of the issuer partitioned clubcards reported in Figure 1 are consistently within 12% of the lower bound for partitioning by issuer. This is consistent with Equation 4, which as a function of r is bounded above by $1.13n\,\mathrm{H}\big(r/n\big)$ when $\epsilon = 0.02$.

Taking data from October 31 as an example, Mozilla's CRLite backend observed 903 million live certificates and 8.7 million revoked certificates. With I ranging over issuers, we found that $\sum_{i\in I} n_i\,\mathrm{H}\big(r_i/n_i\big)$ was 6.31 MB. For comparison, our aggregator generated a 7.0 MB issuer partitioned clubcard for this data set, which is 10.9% larger than the optimum. With I ranging over both issuers and expiry hours, we found that $\sum_{i\in I} n_i\,\mathrm{H}\big(r_i/n_i\big)$ was 5.17 MB. This suggests that we could reduce clubcard size by as much as 17% by switching to issuer-expiry partitioning.

We also computed $\sum_{i\in I} n_i\,\mathrm{H}\big(r'_i/n_i\big)$ for various index sets I with r'_i being the number of new revocations in the six hours after the aforementioned October 31 snapshot was generated. The sum came to 16.8 kB when partitioning by issuer and 10.2 kB when partitioning by issuer and hour-of-expiry. Our zstd-compressed issuer partitioned clubcard for this data set was 22.1 kB, which is 31.5% larger than the lower bound for issuer partitioning and 116% larger than the lower bound for issuer-expiry partitioning.

The larger relative improvement for delta updates can be explained by the fact that certificates are likely to be revoked shortly after issuance.[5]

Effect of stitching. Our implementation only performs the simple form of stitching wherein trivial equations are omitted. Nevertheless, this saves an average of 575 bits for each partition block of non-zero rank. The total savings is approximately 89 kB for issuer partitioned clubcards.

5. https://github.com/davidben/merkle-tree-certs/issues/41#issuecomment-2245218105

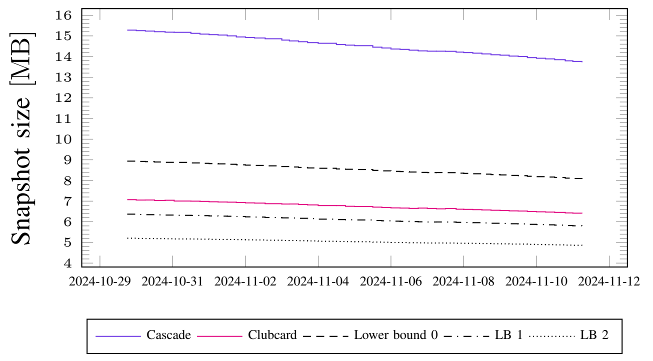

Figure 1. The size of Bloom filter cascades and clubcards produced by Mozilla's CRLite infrastructure between 2024-10-29 and 2024-11-11. Lower bound 0 is for unpartitioned data. Lower bounds 1 and 2 correspond to issuer partitioning and issuer-expiry partitioning respectively. The clubcard data points use issuer partitioning. The data for this plot is embedded in this pdf.

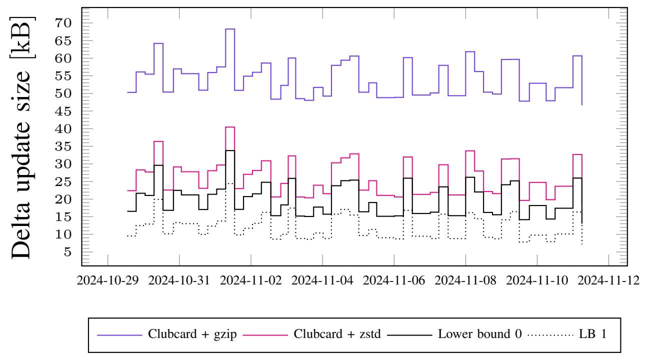

Figure 2. The size of compressed clubcard delta updates produced between 2024-10-29 and 2024-11-11. The clubcards use issuer partitioning. The zstd line is compressed using a full snapshot from 2024-10-09 as a dictionary. Lower bounds 0 and 1 corresponds to issuer partitioning and issuer-expiry partitioning respectively. The data for this plot is embedded in this pdf.

Recompression interval and latency. There are two additional parameters to the system that we have not discussed above:

1) the *recompression interval*: the number of days between full snapshots, and
2) *latency*: the number of hours between delta updates.

These parameters determine the maximum number of clubcards that a client may need to query to determine the revocation status of a certificate. The computational cost of a query grows linearly with the number of delta updates, so the cost remains constant if one doubles the recompression interval and the latency simultaneously.

Mozilla's current deployment has a recompression interval of 20 days and a latency of 6 hours. Table 5 shows how the bandwidth requirements of the system change as the recompression interval and latency are scaled.

TABLE 1. AVERAGE DAILY DOWNLOAD SIZE ASSUMING THAT FULL SNAPSHOTS ARE 7.72 MB, THAT 6-HOUR DELTA UPDATES ARE 26.8 KB, AND THAT DELTA UPDATE SIZE SCALES LINEARLY WITH LATENCY.

Recompression interval (days)	20	45	90	180
Latency (hours)	6	12	24	48
Bandwidth (kB / user / day)	493	279	193	150

6. Discussion

6.1. Comparison with other systems

CRLSets. CRLSets are Google Chrome's primary revocation checking mechanism. A CRLSet is a subset of the issuer–serial number pairs found in CRLs disclosed to CCADB and/or discovered through certificate transparency. The exact process by which CRLSets are generated is not public.[6] However, an email[7] from Adrian Taylor on July 19, 2024, to the chromium-dev mailing list notes that in Q2 of 2024 Chrome "added support for certificate revocations due to key compromise to CRLSet". The email also says that certificates "revoked with the key compromise reason code should now be blocked by Chrome clients within 24-48 hours." The CRLSet published on October 16, 2024 is 540 kB. It contains the issuers and serial numbers of 28669 certificates.

Table 5 already shows that CRLite can deliver *all revocations* at 6 hour latency for less than 500 kB per user per day. CRLite can also be tuned to deliver key compromise revocations at the same latency as CRLSets. Our own survey of the WebPKI on October 16 found 28575 certificates revoked with reason code key compromise (about 0.3% of all revocations). A compressed clubcard encoding these 28575 revocations is 85 kB. On the following day, October 17, we found 28637 key compromise revocations. Of these, 400 were new since the October 16 data set. A clubcard encoding the 400 new revocations was 3.9 kB after zstandard compression using the previous day's clubcard as a dictionary.

With a latency of 24 hours we can use a recompression interval of 90 days while maintaining performance similar to the current deployment of CRLite. Assuming an 85 kB snapshot and 3.9 kB delta updates, CRLite could deliver key compromise revocations with a bandwidth of 4.8 kB per user per day. This is a 99% improvement over CRLSets.

Let's Revoke. The Let's Revoke proposal of Smith, Dickenson, and Seamons [21] would have CAs encode a unique *revocation identifier* in every certificate that they issue. These identifiers would contain (1) the certificate's issuer, (2) the certificate's expiry date, and (3) a sequentially assigned non-negative *revocation number*. Using this new identifier, WebPKI revocation data could be distributed as a set of (compressed) bit vectors, one for each issuer and expiry date, in which the i-th bit of a vector encodes the revocation status of a certificate with revocation number i.

Put differently, Let's Revoke defines a new certificate extension that facilitates a type-aware encoding of the WebPKI revocation set partitioned by issuer and expiry date. Clubcard, on the other hand, requires no changes to the WebPKI and can be used to construct type-aware encodings using arbitrary partitions. Consequently, if Let's Revoke outperforms Clubcard it is by a small constant factor; whereas Clubcard could outperform Let's Revoke by an arbitrary factor by using a different partitioning strategy.

We can get some sense of the relative performance of the systems by looking at unpartitioned data. To this end, we have replicated the simulation of Section IV.A of [21]. We generated 100 random bit vectors of length $n = 1,000,000$ with $r = 10,000$ ones, and we compressed them using several methods. The optimal encoding is $\log \binom{n}{r}$ bits or 10107 bytes. Using xz compression directly on the bit vectors, as in [21], the average size was 12790 bytes ($+27\%$ over the optimum); using clubcards to encode the corresponding sets, the average size was 11122 bytes ($+10\%$); and using Westerbaan's ncrlite [24] to encode the sets, the average size was 10245 bytes ($+1.4\%$).

The $\sim 10\%$ overhead that we observed using clubcards in this simulation is similar to what we observed in Figure 1 using clubcards for real-world data without revocation IDs. Adding revocation IDs to certificates and instantiating Let's Revoke with Westerbaan's ncrlite encoding method would lead to slightly smaller encodings.

We leave it to future work to compare the two systems on partitioned data sets, to make a full accounting of metadata costs, and to compare the client-side performance of the systems.

6.2. Mass-revocation events

The binary entropy function $\mathrm{H}(p)$ takes a maximum value of 1 at $p = 1/2$. As we have seen in the previous sections, the size of a clubcard is a small constant factor (< 1.13) larger than $\sum_i n_i \mathrm{H}(r_i/n_i)$. In the worst-case, a mass-revocation event would result in there being a large number of blocks i for which the revocation rate r_i/n_i is close to $1/2$. As such, the performance of clubcards in a mass revocation event will depend strongly on how well the partitioning strategy "isolates" the revoked certificates.

Consider a single issuer with $n = 180$ M certificates.[8] Suppose that these certificates have with 90 day validity periods, and that this issuer produces 2 M certificates per day. Suppose further that the revocation rate is 0.1% on all days preceding a mass-revocation event.

Prior to the mass-revocation event, our hypothetical issuer would contribute approximately $1.13 \cdot n \cdot \mathrm{H}(0.001) \approx 0.29$ MB to an issuer partitioned clubcard (or an issuer-expiry partitioned clubcard). In the worst-case scenario, the CA would suddenly revoke half of its certificates and this

6. https://www.chromium.org/Home/chromium-security/crlsets/

7. https://groups.google.com/a/chromium.org/g/chromium-dev/c/cMM7RIc1kZI/m/G_MrtVXZAwAJ

8. This hypothetical CA has a number of certificates that is similar to the Let's Encrypt R10 intermediate.

contribution would grow to $1.13 \cdot n \cdot \mathrm{H}(0.5) \approx 25.4$ MB. The result is the same whether we partition by issuer or issuer-expiry as we are assuming a constant revocation rate.

It is unlikely that a CA would spontaneously revoke half of their certificates. In a more realistic scenario, a CA would revoke a large fraction of certificates issued during some temporally-bounded incident. A week-long incident at our hypothetical CA would affect 14 M certificates. For an issuer partitioned clubcard, the worst-case is if all 14 M certificates are revoked. This would increase the revocation rate of the block to 8.5%, and the issuer's contribution to the size of a clubcard would grow to $1.13 \cdot n \cdot \mathrm{H}(0.085) \approx 10.6$ MB. Clubcards partitioned by issuer and expiry would fare better, as the revoked certificates would cluster by expiry date. The worst case is if half of the affected certificates are revoked. With $n_1 = 14$ M certificates with a revocation rate of 0.5 and $n_2 = 166$ M certificates with a revocation rate of 0.001, we get $1.13 \cdot \left(n_1 \cdot \mathrm{H}(0.5) + n_2 \cdot \mathrm{H}(0.001)\right) = 2.24$ MB. Of this, 1.98 MB comes from the mass-revoked certificates.

It is worth noting that the situation improves markedly if all 14 M affected certificates are revoked. The contribution of our hypothetical CA to an issuer-expiry partitioned clubcard would then *drop* to $1.13 \cdot \left(n_1 \cdot \mathrm{H}(1) + n_2 \cdot \mathrm{H}(0.001)\right) = 0.27$ MB.

7. Conclusion

We have demonstrated that CRLite is more practical than previously believed, and that nearly size-optimal encodings of the WebPKI revocation set can be produced without adding new information to certificates. We have also refuted a widely-cited "lower bound" on the size of membership tests for the WebPKI, and we have identified a plausible path towards producing membership tests that are more compact than the ones we reported in Figure 1. We close with some other prospective improvements to CRLite.

7.1. Future work

Dynamic partitioning. Figures 1 and 2 suggest that partitioning by issuer and hour-of-expiry could reduce the size of clubcards. However, the benefit of this partitioning strategy is offset by the amount of metadata that it requires (the data included in U and the number of entries in BlockMeta). We are currently investigating techniques for dynamically selecting a partition that maximizes savings while minimizing metadata costs.

Using the leaf index extension in SCTs. The definition of U given in Section 4.2 has two undesirable properties. First, the MMD offsets on timestamp intervals mean that the clubcard cannot claim to cover all of the certificates that were known to the aggregator. Second, the MMD offsets can introduce false-positives if a CT log violates its MMD promise. Both of these problems would be fixed by the leaf index extension for SCTs that has been proposed as part of the Sunlight / Static Certificate Transparency API [22], [23]. Our CRLite aggregator already tracks the leaf indices that it has seen. We plan on encoding these indices into clubcards so that users can switch to an index-based definition of coverage as the Static Certificate Transparency API gains traction.

Avoiding full snapshots. WebPKI certificates have bounded ($\leq$ 398 day) validity periods [7]. As such, there is no intrinsic reason that we need to publish clubcards that encode full snapshots of the WebPKI. A collection of delta updates spanning a 398 day period can cover all of the certificates with validity in that period. The only reason to publish full snapshots is to reduce the computational cost of querying clubcards—querying hundreds or thousands of clubcards may be too computationally expensive for some clients.

One path to improving this situation is to reduce the maximum validity period for WebPKI certificates. We are currently investigating performance improvements that would make it practical to avoid full snapshots if the maximum validity period were between 45 and 90 days. Assuming 26.8 kB delta updates, as in Table 5, the bandwidth requirements of CRLite with 6 hour latency could drop from 493 kB per user per day to 107 kB per user per day.

Improved ribbon filters. The average overhead for a clubcard is $\sim 12\%$. Much of this is due to the intrinsic overhead for two-level cascades, which comes from integrality constraints, but there is also the ϵ parameter for ribbon filters. While our implementation uses $\epsilon = 0.02$, state of the art implementations such as [5], [10] can achieve sub-1% overhead.

Acknowledgements.. The author would like to thank Brandon Wells and the Mozilla SRE Purple Team for supporting the CRLite backend infrastructure; Dana Keeler for reviewing the code; as well as Mike Hamburg, Dennis Jackson, Dana Keeler, Christoph Kerschbaumer, James Larisch, Dave Levin, Zhisu "Gavin" Wang, Bas Westerbaan, Ben Wilson, and Kathleen Wilson for discussions related to CRLite and comments on early versions of this work.

References

[1] Boeyen, S., Santesson, S., Polk, T., Housley, R., Farrell, S., Cooper, D.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (May 2008), https://www.rfc-editor.org/info/rfc5280

[2] Carter, L., Floyd, R.W., Gill, J., Markowsky, G., Wegman, M.N.: Exact and approximate membership testers. Proceedings of the tenth annual ACM symposium on Theory of computing (1978), https://dl.acm.org/doi/pdf/10.1145/800133.804332

[3] Dietzfelbinger, M., Pagh, R.: Succinct data structures for retrieval and approximate membership (2008), http://arxiv.org/abs/0803.3693

[4] Dietzfelbinger, M., Walzer, S.: Efficient gauss elimination for near-quadratic matrices with one short random block per row, with applications (2019), http://arxiv.org/abs/1907.04750

[5] Dillinger, P.C., Hübschle-Schneider, L., Sanders, P., Walzer, S.: Fast succinct retrieval and approximate membership using ribbon (2021), https://arxiv.org/abs/2109.01892

[6] Dillinger, P.C., Walzer, S.: Ribbon filter: practically smaller than bloom and xor (2021), https://arxiv.org/abs/2103.02515

[7] Forum, C.: Baseline requirements for TLS server certificates. https://cabforum.org/working-groups/server/baseline-requirements/documents/CA-Browser-Forum-TLS-BR-2.0.9.pdf (2024)

[8] Goodwin, M.: Bug 1488865 – import CRLite enrollment state into cert_storage. https://bugzilla.mozilla.org/show_bug.cgi?id=1488865 (2019)

[9] Hamburg, M.: personal communication

[10] Hamburg, M.: Improved CRL compression with structured linear functions (2022), https://rwc.iacr.org/2022/program.php#abstract-talk-39, real World Cryptography

[11] Jones, J.C.: Introducing CRLite: All of the Web PKI's revocations, compressed. https://blog.mozilla.org/security/2020/01/09/crlite-part-1-all-web-pki-revocations-compressed/ (2020)

[12] Larisch, J., Choffnes, D.R., Levin, D., Maggs, B.M., Mislove, A., Wilson, C.: CRLite: A scalable system for pushing all TLS revocations to all browsers. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017. pp. 539–556. IEEE Computer Society (2017), https://jameslarisch.com/pdf/crlite.pdf

[13] Laurie, B., Langley, A., Kasper, E.: Certificate Transparency. RFC 6962 (Jun 2013), https://www.rfc-editor.org/info/rfc6962

[14] Majewski, B.S., Wormald, N.C., Havas, G., Czech, Z.J.: A Family of Perfect Hashing Methods. The Computer Journal **39**(6), 547–554 (01 1996), https://academic.oup.com/comjnl/article-pdf/39/6/547/1103380/390547.pdf

[15] Mozilla: Data privacy principles. https://www.mozilla.org/en-US/privacy/principles (2020)

[16] Mozilla: Privacy policy. https://www.mozilla.org/en-US/privacy/ (2020)

[17] Mozilla: Mozilla root store policy version 2.9. https://www.mozilla.org/en-US/about/governance/policies/security-group/certs/policy/ (2023)

[18] Porat, E.: An optimal bloom filter replacement based on matrix solving (2008), http://arxiv.org/abs/0804.1845

[19] Prince, M.: The hidden costs of heartbleed. https://blog.cloudflare.com/the-hard-costs-of-heartbleed/ (2014)

[20] Seiden, S.S., Hirschberg, D.S.: Finding succinct ordered minimal perfect hash functions. Inf. Process. Lett. **51**(6), 283–288 (1994), https://doi.org/10.1016/0020-0190(94)00108-1

[21] Smith, T., Dickenson, L., Seamons, K.E.: Let's revoke: Scalable global certificate revocation. In: 27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020. The Internet Society (2020), https://www.ndss-symposium.org/ndss-paper/lets-revoke-scalable-global-certificate-revocation/

[22] The Community Cryptography Specification Project: The static certificate transparency API. https://github.com/C2SP/C2SP/blob/main/static-ct-api.md (2024)

[23] Valsorda, F.: The Sunlight CT log. https://filippo.io/a-different-CT-log (2023)

[24] Westerbaan, B.E.: go-ncrlite. https://github.com/bwesterb/go-ncrlite (2024)