CMSC614 FINAL PROJECT

CHASE HUTTON AND LEO S.P. VELLOSO

ABSTRACT. **The certificate ecosystem and PKI is an essential component to modern network security. In order to ensure secure communication between a client and server, the SSL/TLS protocol requires the client to properly validate the certificate chain offered by the server. Much of this includes checking the certificate chain for any possible revoked certificates. Some of the current revocation querying methods available (such as CRLs and OCSPs) are too impractical to see wide scale adoption. Although more secure and space efficient methods, like CRLite, have been proposed and implemented, the need for optimization persists. In this paper, we present a novel, space-efficient construction for storing certificate revocation data. We do so by taking advantage of an inherent structure in the certificate ecosystem, namely, through partitioning certificates by their corresponding issuers.**

1. INTRODUCTION

The foundation of network security relies on the SSL/TLS protocol, which is used to cryptographically establish communication between a client and server. During the TLS Handshake, the server sends its certificate chain to the client – one of the critical steps to properly ensure server authenticity. This works in tandem with the certificate ecosystem's hierarchical structure, in which trusted Certificate Authorities (CAs) issue X.509 certificates. These certificates connect a domain name to a corresponding cryptographic key that is used to sign certificates and establish communication. Recall that a certificate chain is a series of certificates, each of which is cryptographically signed by its predecessor in the chain. Moreover, the chain must contain a certificate issuer that is trusted by the client. If a certificate expires or becomes invalid through other means, it is necessary that the issuer 'revokes' the certificate – creating a record that the certificate is, in fact, invalid. If an invalid certificate is not revoked, then client software might still consider the certificate valid, and hence the chain valid. If attackers gain access to these certificates or their private keys – as seen in the large-scale vulnerability *Heartbleed* [7] - improper revocation could lead to Man in the Middle Attacks (MitM) and/or phishing attacks. Thus, it is crucial that certificate issuers properly revoke their certificates and store this information in order for clients to properly validate certificate chains.

Currently, the most popular methods used by client applications, such as browsers, for checking certificate revocations are Certificate Revocation Lists (CRLs) and Online Certificate Status Protocols (OCSPs). A CRL is simply a list corresponding to the revoked certificates of an issuer. When a certificate issuer signs a certificate, it includes a URL to the corresponding CRL, which can be checked should that certificate be revoked. OCSP providers, which are similarly linked to certificates, allow clients to query certificate issuers for the revocation status of a single certificate. Querying for certificate revocations, however, implies taking many factors into consideration, namely latency, bandwidth concerns, privacy risks due to third parties, and ambiguity. Moreover, most browsers currently implement *fail-open* systems for revocation checking, meaning that if they are unable to obtain certificate revocation information, they proceed with the connection as normal. Considering that most browsers do not push all revocations to their client, this paradigm can have serious security implications.

Ultimately, developing a new method for checking revocations in the certificate ecosystem is analogous to the more abstract idea of constructing an efficient, deterministic querying system that can be practically implemented with large data sets. Unfortunately, due to the large scale of the certificate ecosystem, naive data structures, such as dictionaries, require an impractical amount of space.

In this paper, we provide a construction of a space efficient and queryable data structure to store certificate revocation information. We achieve this by partitioning the certificate ecosystem according to one of its

underlying structures: certificate issuers. In Section 2, we describe the previous work on systems used to store certificate revocation data – focusing on the implementation and methods of CRLite. In Section 3 we introduce background on approximate membership data structures and filter cascades. In Section 4 we formally define the problem and discuss how partitioning certificates by common issuer leads to a more space efficient data structure. In Section 5, we describe the methodology behind our system to push certificate revocations to browsers daily. Finally, we state and analyze the results of our experiment in Section 6, and follow with (conclusion) and discussion on future work in Section 7.

2. Related Work

In 2017, Lariach et al. [6] proposed CRLite – a system that fully checks for certificate revocations. Originally, CRLite was implemented as a browser plugin, but has since been adopted by Mozilla Firefox as their preferred method of querying for certificate revocations. CRLite uses a space-efficient cascade structure, comprised of bloom filters, to store certificate revocation data. It is implemented in two parts: a server-side that constructs the filter consisting of known certificate data, and a client side that downloads the filter. The client can, then, use the filter to check for revocations locally. Moreover, CRLite handles newly revoked certificates by reconstructing the filter server-side, and providing the client with 'delta-updates'. In order to produce a delta-update, the server compares the bitwise differences between the previous filter and the newly constructed filter. It then signs these updates, and makes them available to clients, which can be used to update filters locally. This allows for a more space-efficient approach that reduces latency, as clients only have to download the filter once, and update their filter according to the corresponding delta-updates – which are comparatively much smaller in size. By the end of their experiment, the dataset collected by CRLite consisted of ~ 12 million revoked certificates, which produced a filter size of roughly 10MB – with about 580KB per day for the delta-updates.

3. Preliminaries

3.1. **Retrieval Data Structures and Filters.** A *retrieval data structure* represents a function $f\colon R \to \{0,1\}^r$ for a set $R \subseteq \mathcal{U}$ of keys from a universe U and $r \in \mathbb{N}$. A query for $x \in R$ must return $f(x)$ whereas a query for $x \in U \setminus R$ may return any value from $\{0,1\}^r$. Using a retrieval data structure, we can implement an *approximate membership query data structure* (AMQ, also referred to as a filter) by associating to each key in R an r-bit fingerprint. This structure supports membership queries for R with *false positive rate* $\varphi = 2^{-r}$. A membership query for x simply compares the fingerprint of x with the result returned by the retrieval data structure for x, i.e. $f(x)$. These values are the same if $x \in R$ and otherwise are the same with probability 2^{-r}. The two primary examples of AMQ's we use in this paper are Ribbon Filters [5] and Bloom filters [2].

3.2. **Certificate Ecosystem.** In this paper we take U to be the set of known certificates and R to be the set of known revoked certificates. Each certificate is uniquely identified by concatenating the SHA256 hash of the SubjectPublicKeyInfo field from the issuer's certificate with the certificates issued serial number.

3.3. **Filter Cascades.** A *filter cascade* is a series of filters that in combination provide a data structure capable of answering membership queries for R while ensuring no $x \in U \setminus R$ is a false positive. The notion of a filter cascade was first introduced in [4] and used as the main data structure for determining revocations in CRlite. We use the filter cascade construction algorithm given by CRlite and for completeness give pseudocode below (see Algorithm 1). We note we make a slight change to algorithm given by CRlite in that we set the false positive rate for each filter construction after the first level of the cascade to be $1/2$. For a detailed explanation of this algorithm, see section 3 in [6]. The exact size of a filter cascade is determined by the type of filter (or base filter type) used in the construction. Using Ribbon and Bloom filters this size is upper bounded by an expression of the form

$$c_1|R| + c_2|R| \log_2\left(\frac{|U|}{|R|}\right)$$

where c_1 and c_2 are positive constants depending on the filter type.

Algorithm 1: Build Cascade(ℓ, C, R, S)

Input: The current level ℓ, a partial filter cascade C, a set R and its complement S
Output: The complete filter cascade C
if $|R| = 0$ **then**
  return C
if $|R| < |S|$ and $\ell = 0$ **then**
  $k \leftarrow -\log_2(\frac{|R|}{\sqrt{2}|S|})$
else
  $k \leftarrow 1$
$F_\ell \leftarrow$ build Filter for universe $R \cup S$ and subset R with $\varphi = 2^{-k}$
add F_ℓ at level ℓ of C
$R' \leftarrow$ subset of S that are false positives on F_ℓ
return Build Cascade($\ell + 1$, C, R', R)

4. Partitioning by Common Issuer

Formally we wish to solve the following problem: given a set $U = R^{\mathsf{c}} \cup R$ where U represent the universe of known certificates and R represents the subset of known revoked certificates, we wish to compute a queryable data structure for R. As discussed above, we can do this using a filter cascade. We refer to a filter cascade constructed using base filter F for the entire universe U, as an F-global filter cascade and denote it $F_G(U, R)$. A priori, doing better than an F-global filter cascade seems challenging. As mentioned in section 3, the space complexity for these cascades are bounded above by $c_1|R| + c_2|R|\log_2(|U|/|R|)$ where c_1 and c_2 are positive constants depending on F. Ignoring constants and lower order terms, these constructions essentially achieve the information theoretic lower bound for the number of bits required to communicate R. However, there is inherent structure in U that is not taken into account when constructing a global filter cascade. In this section, we describe how to take advantage of the fact that certificates in U can be naturally grouped by common issuer, to produce a data structure for R which in general is able to out perform global filter cascade constructions.

To start, we need a consistent way to label certificate issuers that is more space efficient then using the SHA256 hash of their public key. We do this using a minimal perfect hash function (MPHF). Recall that a MPHF is a function that maps a set of k objects to $[k]$. We can construct a MPHF for the set of unique issuer public key hashes and store it using a linear number of bits in the number of unique issuers. Specifically, the number of bits required to store such a function is at most 3 times the number of unique issuers (See any standard paper on MPHF, e.g. [3]). Going forward, we denote this function h and note that, when we make reference to the ith issuer, we mean the issuer whose public key hash is mapped to i by h. Furthermore, we define the partition of U by common issuer as $P = \{U_1, \cdots, U_k\}$ where k is the number of unique certificate issuers and $U_i = R_i^{\mathsf{c}} \cup R_i$ is the set certificates issued by the ith issuer. Let $n_i = |U_i|$ and $r_i = |R_i|$. We denote the filter cascade constructed for the pair (U_i, R_i) by $F_L(U_i, R_i)$ and denote its space complexity by $S(r_i, n_i) = c_1 r_i + c_2 r_i \log_2(n_i/r_i)$. Then our proposed membership data structure is simply the collection of filter cascades $\{F_L(U_i, R_i)\}$ together with h. We refer to this construction as an F-local filter cascade. Pseudocode for the construction and querying algorithms is given below.

Next we analyze the space complexity of our new construction. Particularly, we show that the space complexity of an F-local filter cascade is (for all practical purposes) smaller than the space complexity of an F-global filter cascade. Note that the space complexity for the F-local filter cascade is simply $3k + \sum_i S(n_i, r_i)$. Suppose now that $n = \sum_i n_i$ and $r = \sum_i r_i$ where $r_i < n_i$. Define the following ratios:

Algorithm 2: Construct Local Filter Cascade(P)

Input: P is a set of certificates partitioned by issuer
Output: Collection of issuer-specific filters
for $U_i \in P$ **do**
  Construct Filter $F_L(U_i, R_i)$
Return $\{F_L(U_i, R_i)\}$

Algorithm 3: Query Local Filter Cascade(c)

Input: a certificate c with issuer public key hash x and serial s
Output: true if c is a known revoked certificate and false otherwise
$i \leftarrow h(x)$
Query $F_L(U_i, R_i)$ with s and return the result

$p = r/n$ and $p_i = r_i/n_i$. Then

$$p = \frac{\sum_i r_i}{\sum_i n_i} = \sum_i \frac{n_i}{n} p_i \tag{1}$$

so p is a convex combination of values p_i with weights $\alpha_i = n_i/n$. Next note that the function $f\colon (0,1) \to \mathbb{R}$ defined by $f(x) = x \log_2(1/x)$ is strictly concave. It now follows by Jensen's inequality and (1) that

$$p \log(1/p) = f\left(\sum_i \alpha_i p_i\right) \geq \sum_i \alpha_i f(p_i) = \sum_i \frac{n_i}{n} p_i \log_2(1/p_i). \tag{2}$$

Multiplying through (2) by n and substituting for p and p_i gives

$$r \log_2(n/r) \geq \sum_i r_i \log_2(n_i/r_i). \tag{3}$$

Note that equality holds in (3) exactly when $n_1/r_1 = n_2/r_2 = \cdots = n/r$. It now follows from (3) that

$$S(n,r) - \sum_i S(n_i, r_i) = c_2 \left(r \log_2(n/r) - \sum_i r_i \log_2(n_i/r_i) \right) \geq 0. \tag{4}$$

It is certainly the case that the ratio of total issued certificate to revoked certificates, i.e. n_i/r_i, is not the same for every certificate issuer. Therefore the inequality in (4) can taken to be strict. Further, as $k \ll n$, the added cost of storing h is negligible. Therefore, when storing any reasonable number of revocations, we conclude that the F-local filter cascade has strictly smaller space complexity than the F-global filter cascade.

5. System Design

In this section we discuss how to use F-local filter cascades to design a simple system for pushing all certificate revocations to browsers on a regular basis. We note that one can smoothly modify the current CRlite implementation to use local bloom filter cascades for each unique certificate issuer rather than the currently used global construction.

The first step in designing a system for disseminating revocation information, is to create a program that collects raw certificates and processes them into some consistent representation. For simplicity, we use the current CRlite aggregator. For details about its design see [6]. Using this aggregator, we obtain the following daily information: for each certificate issuer we have the SHA256 hash of its public key, a list of DER-encoded serial numbers for its known valid certificates, and a list of DER-encoded serial numbers for its known revoked certificates. Using the list of SHA256 hashes, we construct the MPHF h described

in section 4. Furthermore, observe that for the ith issuer, the list of known revoked certificates corresponds exactly to the set R_i and the list of known valid certificates corresponds exactly to the set R_i^{c}. Thus for the ith issuer we obtain $U_i = R_i^{\mathsf{c}} \cup R_i$ and thus obtain the desired partition $P = \{U_1, \ldots, U_k\}$. To differentiate between days, we label the partition computed for day i as $P_i = \{U_{i,1}, \ldots, U_{i,k}\}$.

With this partition, we can build an F-local filter cascade for each day i. The next step is to determine how best to distribute this data structure to clients. Ideally, we could build the data structure once, push it the client, and maintain it through significantly smaller daily updates. Unfortunately, filter data structures are typically static. CRlite solves this problem by constructing new filter cascades each day, and computing the bitwise XOR of each level to determine which bits have changed in each filter. These deltas are complied and sent daily to the client. We could adopt this approach for our data structure, however, if we wish to use more space efficient filters, e.g. ribbon filters, it is less clear how large delta updates could be. Instead we devise a more naive strategy.

We propose the following system: on day 1, we build an F-local filter cascade for P_1. We refer to this as the primary lcoal filter cascade. Then, on day i, we compute P_i and for each unique certificate issuer j, we compute $U'_{i,j} = U_{i,j} \setminus U_{1,j}$ and $R'_{i,j} = R_{i,j} \setminus R_{1,j}$. In words, we determine for each certificate issuer, the set of newly issued known certificates and the set of newly known revoked certificates since day 1. Finally, we build an F-local filter cascade for the partition $P'_i = \{U'_{i,1}, \ldots, U'_{i,k}\}$. We note that we make a simplifying assumption that the number of certificate issuers is static. This assumption can be easily removed by simply computing a new MPHF h' for each day i. Moving forward, we refer to the structure computed for P'_i as the secondary local filter cascade computed on day i. What we've described so far will constitute the server side of our system.

The client side is just as simple and works as follows: on day 1 the client downloads the primary local filter cascade. Now on day i the client downloads the secondary local filter cascade constructed on day i. Thus, on day i the client has a primary and a secondary local filter cascade. To query a certificate, the client first checks the 'not before date' on the certificate to determine if it was issued before day 1. If it was, the client first queries the primary local filter cascade. If the primary local filter cascade identifies the certificate as revoked, then the client can be assured that the certificate is still revoked on day i. Otherwise, its possible that this certificate has been revoked since day 1, and hence the client must also query the secondary local filter cascade. In the case that the certificate was not issued before day 1, the client simply queries the secondary local filter cascade.

We define the size of this system on day i to be the size of the primary local filter cascade plus the size of the secondary local filter cascade constructed on day i. Eventually, our system will no longer be cost effective on some day i as the secondary cascade size will grow too large. When this happens, we simply reset the system and construct a new primary local filter cascade that is pushed to the client. The process then proceeds as before.

6. Implementation and Experiments

We implement algorithm 1 using C++ templates to abstract the functionality of a filter. Since our data sets are quite large ($\sim$ 900 million) we use Parlaylib [1], which is a toolkit for programming parallel algorithms on shared-memory multicore machines, to parallelize the construction. For our base filter types, we use bloom filters and ribbon filters [5]. We use the implementation of ribbon filters provided in [5] and publicly available here. For bloom filters, we use an open source implementation available here. We slightly modify both filter implementations to take as input a user provided hash function and a desired false positive rate. The full implementation is available on GitHub.

We did not implement the full system in section 5. Namely, we did not create an actual server to host our filter constructions nor did implement construction an MPHF for certificate issuers. Instead, we used AWS cloud computing to simulate the system's space requirement on the client side over a span of 20 days on large data sets.

6.1. **Data Set.** As mentioned in section 5, we can use the CRlite aggregator to obtain processed certificate data. All of the data collected and processed by the aggregator is publicly hosted in Google Cloud Storage in batches of 30 days. We downloaded certificate data for dates between 11-08-2024 and 11-28-2024 using Google's gsutil tool. Using standard C++ file processing libraries, we computed some basic statistics about this data set and summarize the results in Table 1 in the appendix.

6.2. **Setup.** In addition to simulating the system described in section 5, we also compare the relative performance of our local filter cascades using different base filter types to CRlite in order to determine cost effectiveness as defined in the previous section. In our experiments, we took 11-08-2024 to be day 1 and use the implementation for algorithm 2 to construct primary local filter cascades using both bloom and ribbon filters. Then on each day after, we construct secondary local filter cascades (again using both ribbon and bloom filters) for the new known valid and revoked sets. Finally, we download the corresponding CRlite filters for each simulated day. These are hosted in the same Google Cloud Storage location as the CRlite aggregator data.

The simulation runs until 11-28-2024, after which the results are written to an output text file to be processed. Finally, to compare F-local vs F-global filter cascades we use the data set for 11-11-2024 and our implementations of algorithms 1 and 2 to construct local and global filter cascades using both bloom and ribbon filters. We also compare these results to the size of the CRlite filter on 11-11-2024.

6.3. **Results.** See the appendix 8.

6.4. **Discussion.** We frist discuss the results of our size comparison test for the data set on 11-11-2024 as seen in Table 2. The CRLite Filter had a total size of 13.7MB which roughly equals our implementation of a global bloom filter cascade. As expected, using ribbon filters instead of bloom filters for the global construction offers significant improvements with a $\sim$ 3.2MB difference in total size. Next, we see that our local bloom filter cascade implementation had a total size of 10.3MB, with a difference of $\sim$ 3.4MB from both CRLite and our global bloom filter cascade. We note that this result implies that CRLite could be significantly improved upon by using a local bloom filter cascade construction. Interestingly, we also see that the local bloom filter cascade outperformed the global ribbon filter cascade being $\sim$ .186MB smaller. Finally, we see that the most significant space improvements are achieved by our local ribbon filter cascade constitution, which is $\sim$ 5.8MB smaller than the CRlite filter.

The results of simulating the cost of our system using the data outlined in Table 1 can be seen in Figure 2. As seen in the graph, over the 20 day experiment, the total size of our system remained smaller than that of CRLite's. However, as seen in Figure 1, the size of the CRLite delta-updates remains under 100KB. Considering that the size of the secondary local filter cascade grows linearly with the number of newly revoked certificates (see Figure 1), it is likely that it would not remain cost-efficient over a longer period of time, and, generally, may not be the most cost-efficient system to disseminate revocation information. Perhaps, our system would significantly benefit from a method analogous to CRLite's delta-updates. This would be interesting avenue for future work.

7. Conclusion

In this paper we have provided a novel construction for a space-efficient data structure to represent revoked certificates. We hope that our initial findings can serve as a starting point for other ways in which one can partition the certificate environment in order to obtain more space efficient data structures.

REFERENCES

[1] Guy E. Blelloch, Daniel Anderson, and Laxman Dhulipala. Parlaylib - a toolkit for parallel algorithms on shared-memory multicore machines. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 507–509, 2020.

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

[3] Fabiano Botelho, David Menotti, and Nivio Ziviani. A new algorithm for constructing minimal perfect hash functions. 01 2008.

[4] Bernard Chazelle, Joe Kilian, Ronitt Rubinfeld, and Ayellet Tal. The bloomier filter: an efficient data structure for static support lookup tables. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '04, page 30–39, USA, 2004. Society for Industrial and Applied Mathematics.

[5] Peter C. Dillinger and Stefan Walzer. Ribbon filter: practically smaller than bloom and xor, 2021.

[6] 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 *2017 IEEE Symposium on Security and Privacy (SP)*, pages 539–556, 2017.

[7] Liang Zhang, David Choffnes, Tudor Dumitraş, Dave Levin, Alan Mislove, Aaron Schulman, and Christo Wilson. Analysis of ssl certificate reissues and revocations in the wake of heartbleed. *Commun. ACM*, 61(3):109–116, February 2018.

8. APPENDIX

Date	Total Certificates	Revoked Certificates	Valid Certificates	New Revoked	New Valid
11-08-2024	914,255,887	8,061,804	906,194,083	-	-
11-09-2024	915,983,933	7,964,291	908,019,642	37,241	8,073,590
11-10-2024	916,837,610	7,870,888	908,966,722	66,707	15,494,722
11-11-2024	918,024,801	7,759,402	910,265,399	90,776	23,137,122
11-12-2024	919,805,287	7,656,123	912,149,164	120,195	31,344,997
11-13-2024	921,165,701	7,565,760	913,599,941	157,016	39,248,051
11-14-2024	922,515,701	7,487,398	915,028,303	188,253	47,307,479
11-15-2024	923,849,095	7,414,799	916,434,296	223,086	55,416,562
11-16-2024	925,653,879	7,341,918	918,311,961	254,079	63,326,119
11-17-2024	927,478,623	7,258,016	920,220,607	277,460	71,301,072
11-18-2024	928,719,580	7,190,521	921,529,059	299,659	79,053,883
11-19-2024	930,490,180	7,123,352	923,366,828	329,302	87,687,409
11-20-2024	931,962,104	7,045,162	924,916,942	356,826	96,497,890
11-21-2024	933,495,950	6,959,868	926,536,082	382,999	105,282,485
11-22-2024	935,703,706	6,884,980	928,818,726	413,789	114,033,378
11-23-2024	936,679,756	6,809,042	929,870,714	441,102	121,976,675
11-24-2024	935,694,359	6,726,251	928,968,108	462,763	128,561,736
11-25-2024	935,026,270	6,647,508	928,378,762	483,747	135,338,148
11-26-2024	935,051,013	6,574,478	928,075,535	511,400	142,273,880
11-27-2024	934,069,205	6,492,317	928,576,888	539,373	150,137,112
11-28-2024	934,560,309	6,407,573	928,802,736	568,275	157,869,551

TABLE 1. Certificate Statistics by Date

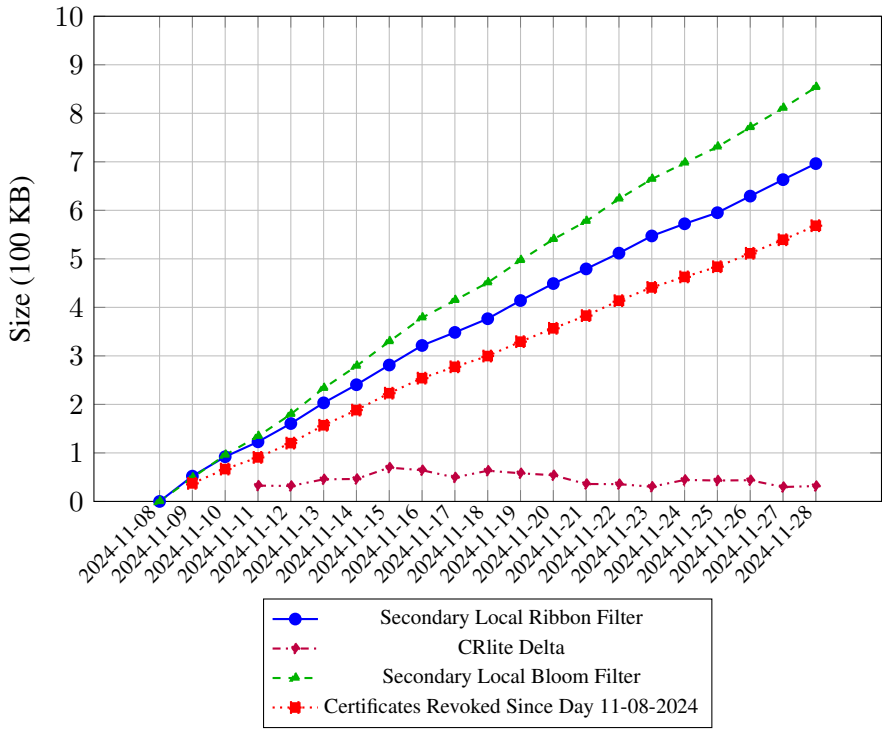

FIGURE 1.

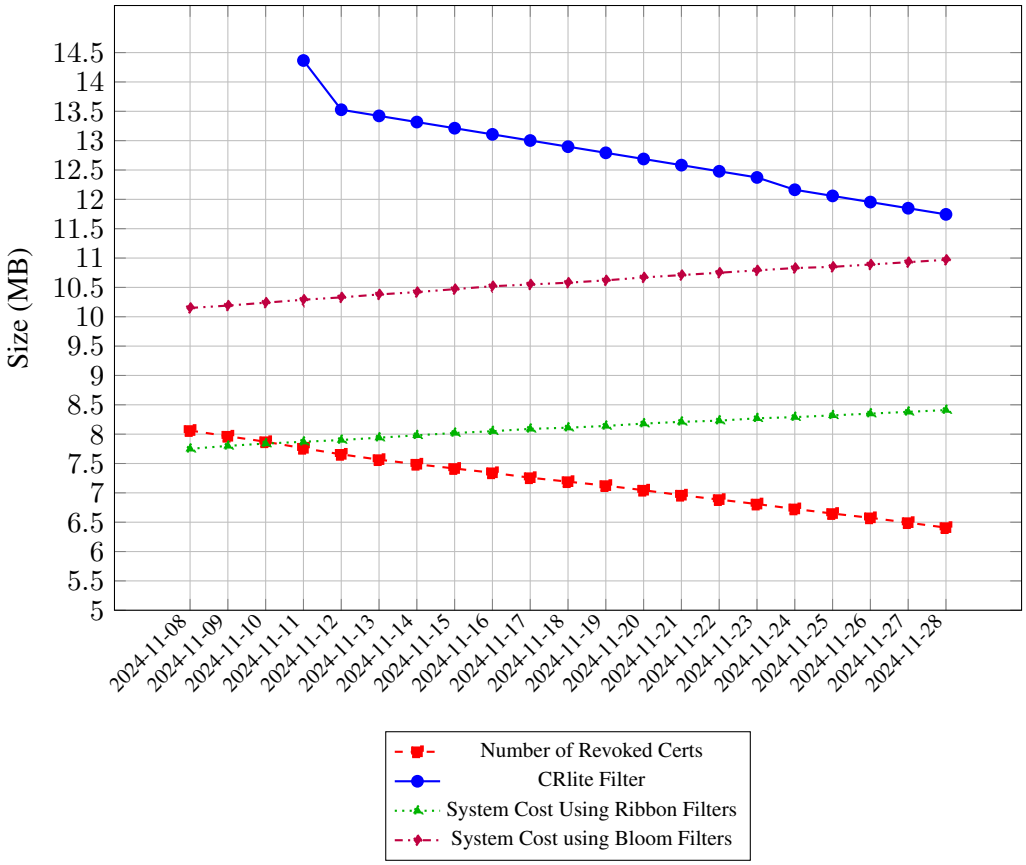

FIGURE 2.

Filter Cascade Type	**Total Size (Bytes)**
CRlite	13,722,419
Global Ribbon Filter Cascade	10,519,224
Local Ribbon Filter Cascade	7,916,744
Global Bloom Filter Cascade	13,705,030
Local Bloom Filter Cascade	10,332,856

TABLE 2. Total Sizes on 11-11-2024