An Efficient PGP Keyserver without Prior Context

Alexander Rucker
acrucker@stanford.edu

December 13, 2017

Abstract

This paper describes the implementation of a synchronizing PGP key server based on a simple and efficient set reconciliation protocol. The key server is able to synchronize very quickly when the difference is small, and the estimation time scales linearly with the size of the difference. The estimation overhead does not increase with the size of the underlying dataset. The key server is able to reconcile a single-key difference with less than 5 kiB of overhead and in less than 10 ms.

1 Introduction

Pretty Good Privacy (PGP) is a public key encryption application suitable for encrypting email and other files [1]. It is based on the concept of a “web of trust,” which eliminates the need for a central certificate authority. Instead, users generate their own keys, and sign the keys of other users whose identifies they have verified. The user can then verify that a key belongs to someone he or she has never met before by following the web of trust back to keys he or she trusts. This requires the use of a key server, which allows users to publish their public keys so that they can be downloaded.

Because PGP does not rely on a centralized authority, there are multiple key servers that users can use to publish their keys. These key servers form a network, and periodically synchronize with each other to ensure that all keyservers contain all keys. The existing network is based on the SKS keyserver, and currently contains approximately 4.6 million keys with a total size of almost 10 GiB [2]. The average key size is around 2 kiB, but key sizes vary depending on the amount of included metadata, the selected key algorithm, and the strength of the key. There are approximately 1000 updates to the stored keys per day, which is an update rate of around 0.02 % per day.

Because of this, an efficient algorithm is needed to propagate updates throughout the network. The SKS network uses the algorithm presented in [3, 4], which represents the state of a keyserver as a polynomial over a finite field where the hash of each key is a root. The difference between two servers can be found by solving for the roots of the quotients of their polynomials. The polynomials are sampled at a set of points, and then the difference polynomial is found by interpolating the quotients of the sampling points.

The keyserver described in this paper uses the far simpler algorithm proposed in [5]. The algorithm is based around an Invertible Bloom Filter (IBF), and also uses a novel set difference algorithm. The use of a difference estimator that efficiently estimates small differences means that servers can frequently poll each other to check for updates without requesting a large amount of data.

2 Algorithm Summary

The key data structure used in synchronization is the Invertible Bloom Filter (IBF) proposed in [5]. This is a variant of a counting bloom filter, with the addition of two extra values per bucket: all of the data values xor’ed together, and all of the hashes of the data values xor’ed together. An example IBF is shown in Table 1(a). This filter would be decoded by looking at the buckets with one entry, removing that entry, and repeating until all buckets were empty.

The synchronization relies on the ability to subtract two IBFs and decode the difference. This is the reason for the addition of the hash field: the second element in Table 1(c) has a count of one, but actually contains 2 elements from the first IBF filter and one from the second. However, the hash field does not equal the hash of the value field, so decoding instead proceeds with the value from the last bucket, and completes successfully.

The IBFs also play a key role in determining the set difference. The Strata estimator uses a hierarchy of bloom filters, where filter n samples the original set with probability 2^{-n} [5]. For this implementation, the sampling is performed by counting the number of trailing zeros in each data element. All n bloom filters are subtracted, and are then decoded in reverse order (from smallest set to largest), counting the number of elements at each level. Once a bloom filter at level n does not decode, the estimated total number of elements is multiplied by 2^n.

The original implementation of the Strata estimator uses a full implementation, where all n bloom filters are sent. Because the data elements for the keyserver are 160-bit hashes, this would require a large overhead. Instead, the key server attempts to synchronize with a truncated estimator first. If none of the levels decode, it requests a larger estimator. This plays an important role in keeping synchronization fast for small set differences.

3 Implementation

To demonstrate the feasibility of using Strata set difference estimators and invertible bloom filters for synchronizing PGP keys, a proof-of-concept implementation was developed. The implementation uses the BerkeleyDB [6] key-value store for storing keys on disk and uses the Ulfius [7] framework to implement a RESTful HTTP API. The implementation supports synchronizing keys between servers, and places an emphasis on minimizing computation overhead. It is implemented in less than 2200 lines of C89, including header files, `-WAll` clean, and checked using Valgrind for memory leaks and uninitialized memory accesses. SHA-1 is used throughout the code as a hash function, because cryptographic hashes are not necessary for the IBF.

3.1 Design Choices

To use the bloom filters to effectively decode the set difference, it was necessary to choose some universe of values to map all keys to. A simple choice would be the key fingerprint, or some substring thereof. However, it is possible to upload multiple keys with the same fingerprint, because the fingerprint is only computed based on the public key data. As a user acquires signatures for his or her key, he or she will want to upload new revisions of the key that contain these new signatures. If the server already contained the key, though, it would not be able to handle the revision. A similar problem involves sending the revocation for some key, which must also be indexed by fingerprint.

Therefore, the value chosen is the full 160-bit SHA-1 hash of the entire key data. This means that two different revisions of the same key will hash to different values, and will propagate through the network separately. When a user requests the key data for a specific fingerprint, all keys with that fingerprint will be returned by the search. A space-efficient implementation may choose to store all packets with a key separately, and perform de-duplication of packets for updated keys. This would greatly increase the complexity of the database, and was not implemented because the expected rate of key updates and revocations is low.

Another implementation decision was the size and number of bloom filters and set difference estimators to include. For simplicity, all data structures were scaled along one axis: difference estimators by the number of levels, and bloom filters by the number of buckets. The strata estimators use 2^n levels, 4 buckets per key, and 40 total buckets for $0 \leq n < 5$, and the bloom filters use $10 * 2^n$ buckets with 4 buckets per key for $0 \leq n < 11$. An optimized implementation would most likely want to use an increased k for larger bloom filters, and smaller bloom filters for smaller strata.

Count	Values	Hashes
1	A	$H(A)$
2	$B \oplus C$	$H(B) \oplus H(C)$
1	C	$H(C)$
2	$B \oplus A$	$H(B) \oplus H(A)$

(a)

Count	Values	Hashes
1	A	$H(A)$
1	D	$H(D)$
1	D	$H(D)$
1	A	$H(A)$

(b)

Count	Values	Hashes
0	0	0
1	$B \oplus C \oplus D$	$H(B) \oplus H(C) \oplus H(D)$
0	$D \oplus C$	$H(D) \oplus H(C)$
1	B	$H(B)$

(c)

Table 1: Two invertible bloom filters and their difference.

3.2 Basic Operation

Before the server starts for the first time, the operator must load it with a recent dump of another keyserver because the synchronization protocol is not able to synchronize if the set difference is too large. Once the database is built, the server is started with a user-specified list of peer servers and synchronization intervals. The server runs one-way reconciliation against each peer. One-way reconciliation was chosen to simplify the process of adding new servers: a user does not have to ask permission to start a read-only keyserver, unlike SKS [2].

The synchronization protocol starts by requesting the smallest set difference estimator, and requesting progressively larger estimators until it is able to successfully decode the difference. Once it has decoded the difference, it requests a bloom filter with at least 3 times as many buckets as the estimated difference. This is based on the empirical result in [5], and intended to give an extremely high probability of successfully decoding the bloom filter (greater than 99 %). The server then decodes the bloom filter, producing a list of key hashes. It then downloads each key from the peer server using the hash.

3.3 API

The API for the keyserver is shown in Table 2. The API is designed for simplicity; by using HTTP requests for all updates the user can effectively test the behavior of the server using only a web browser. The server sends the values of bloom filters and strata estimators by Base-16 encoding the hash values and printing them with their counts, which allows easy manual verification of correct output. The API is also intended for effective use with a document cache. During normal operation, the server will serve the same difference estimator, IBF, and recently added keys. This is a small amount of data, on the order of a few kilobytes.

3.4 Limitations

Because the server is intended as a proof of concept, there are several minor limitations. One limitation is that the server does not use BerkeleyDB's transaction subsystem. This was chosen to speed up database build time, and decrease the complexity of managing the server. Although for many systems this would be a major limitation, it is not a major problem assuming servers synchronize with each other: for a key to be lost in a server failure, it would be necessary for the server to fail without writing the key to disk and fail before any other peer downloaded the key. In this rare failure case, the user would only have to re-upload the public key. To avoid errors, the server traps the SIGTERM signal, and uses it to ensure clean shutdown at the operator's request.

The server is also not designed for bulk operations using the online `/pks/add` and `/pks/lookup` operations. This is not expected to be a major limitation; users are not supposed to upload many public keys to a keyserver. Additionally, the `/pks/lookup` endpoint is only intended for synchronizing small differences and downloading specific keys be end-users. When a sysadmin wants to initialize a new keyserver or synchronize a keyserver after a long time offline, he or she cannot do so by setting it up to peer with an existing keyserver; instead, the new server needs to be initialized with a relatively recent dump of PGP keys.

Finally, the server does not fully parse and val-

Endpoint	Method	Description
`/pks/lookup`	`GET`	Supports searching for keys by user ID string, key ID, fingerprint, or hash. Also supports downloading matching keys.
`/pks/add`	`POST`	Uploads a single ASCII armored key to the keyserver.
`/ibf/k/N`	`GET`	Returns the invertible bloom filter with k buckets per key and N buckets total.
`/strata/d/k/N`	`GET`	Returns the set-difference estimator with d invertible bloom filters, each with k buckets per key and N total buckets.
`/status`	`GET`	Produces a human-readable page describing the status of the keyserver, including the status of its synchronization peers.

Table 2: The API supported by the keyserver.

idate keys. This means that the server has no concept of key expiration dates, subkey signatures, revocations, or erroneous keys. Keys are parsed to ensure that they contain valid packets, which limits the ability to upload random data, and keys are verified to be at least Version 4 [8]. Although it would be useful for the server to fully validate keys, it would significantly increase the complexity of the server code and clients must perform their own validation anyway.

Endpoint	Requests/sec
`/strata/1/4/40`	10868
`/strata/4/4/40`	5374
`/ibf/4/10`	16851
`/ibf/4/20`	15227
`/ibf/4/40`	12436
Search for key	9
Download key	6805

Table 3: Operation throughput for several common operations on the full PGP key database.

4 Evaluation

There are three metrics necessary to evaluate the effectiveness of a distributed key server:

- Number of peer servers
- Zero-difference overhead
- Overhead scaling

4.1 Throughput

The first key metric is the synchronization throughput of the key server. Although this implementation is not optimized for maximum throughput (it uses `snprintf` to build each response, and performs several buffer copies per response that are not strictly necessary), it is still sufficiently fast for normal operation. The results of using ApacheBench [9] to perform requests against several key data structures are shown in Table 3.

This shows that they server is able to handle a large number of synchronization requests per second. Assuming two servers are peering with each other once every minute, they can expect to exchange approximately one update per peering operation. This can be handled using the minimum-size set difference estimator, a single key download, and the minimum-size bloom filter. In total, a user could expect to synchronize tens of thousands of servers over this protocol per minute, without considering the addition of a caching layer. With a caching layer, the synchronization throughput is effectively infinite.

The server does have a very poor response time for key searches, which is due to an unoptimized index that requires every search to scan the list of all keys. This was necessary for user ID searches, which return keys whose user IDs contain the provided search query. This could be improved by adding a more efficient data structure for some searches, such as email searches, or using an index that takes advantage of the typical `name (comment) <email>` structure of user IDs.

4.2 Synchronization Overhead

The second key metric, which is closely related to throughput, is the zero-difference synchronization overhead. As servers start synchronizing more and more frequently (less than once per minute), they can expect to synchronize with peers that do not have any differences. If there is no expected difference, the server will only have to serve the smallest set difference estimator, which is less than 4 kiB, and could be made even smaller by decreasing the number of buckets. If there is a difference, the smallest bloom filter must also be served, and is less than 1 kiB.

Another key concern is how the synchronization overhead scales with dataset size. Figure 1 shows the total number of bytes transferred to synchronize varying numbers of keys, and Figure 2 shows the time to synchronize two keyservers running on the same laptop. Neither the synchronization time nor data increase with the size of the underlying database; they only increase with the number of key that actually have to be transferred. The increase in bloom filter overhead is linear in the number of keys, and the increase in set difference estimator overhead is roughly logarithmic.

Additionally, the server can synchronize less than 10 keys within 100 ms, and 1 key within 10 ms. Although running the synchronization over the Internet would increase the transfer latency, this shows that the time to encode, decode, and process the relevant data structures is very low.

5 Conclusion

In this paper, I implemented and benchmarked a new PGP key server with an efficient set reconciliation algorithm. The key server is implemented in a very small amount of code: less than 2200 lines of C89, excluding the HTTP framework and the database, with core difference algorithm taking a few dozen lines of code. This demonstrates the simplicity of the difference estimation and reconciliation procedure. Benchmarks of the key server show that it is able to efficiently reconcile differences and that the difference reconciliation time is low and scales gracefully with increasing differences.

References

[1] gpg(1) – linux man page. `https://linux.die.net/man/1/gpg`.

[2] sks-keyserver. `https://bitbucket.org/skskeyserver/sks-keyserver/wiki/Home`.

[3] Yaron Minsky, Ari Trachtenberg, and Richard Zippel. Set reconciliation with nearly optimal communication complexity. *IEEE Transactions on Information Theory*, 49(9):2213–2218, 2003.

[4] Yaron Minsky and Ari Trachtenberg. Practical set reconciliation. In *40th Annual Allerton Conference on Communication, Control, and Computing*, volume 248, 2002.

[5] David Eppstein, Michael T Goodrich, Frank Uyeda, and George Varghese. What's the difference?: efficient set reconciliation without prior context. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 218–229. ACM, 2011.

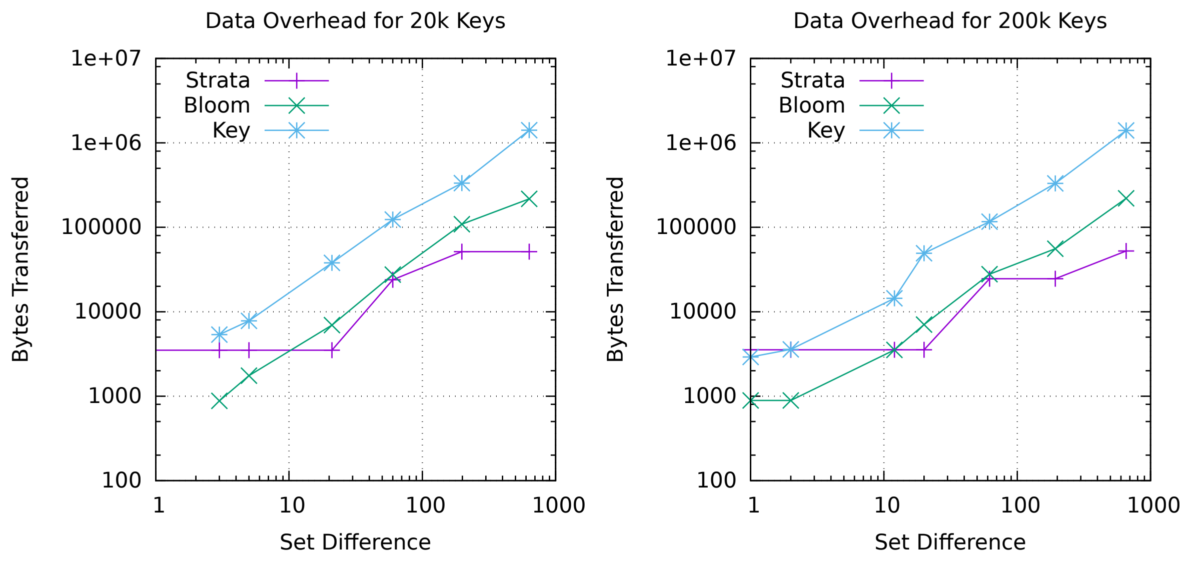
[6] Oracle berkeley db. `http://www.oracle.com/technetwork/database/database-technologies/berkeleydb/overview/index.html`.

[7] Ulfius. `https://github.com/babelouest/ulfius`.

[8] J Callas, L Donnerhacke, H Finney, D Shaw, and R Thayer. Openpgp message format. RFC 4880, RFC Editor, 11 2007.

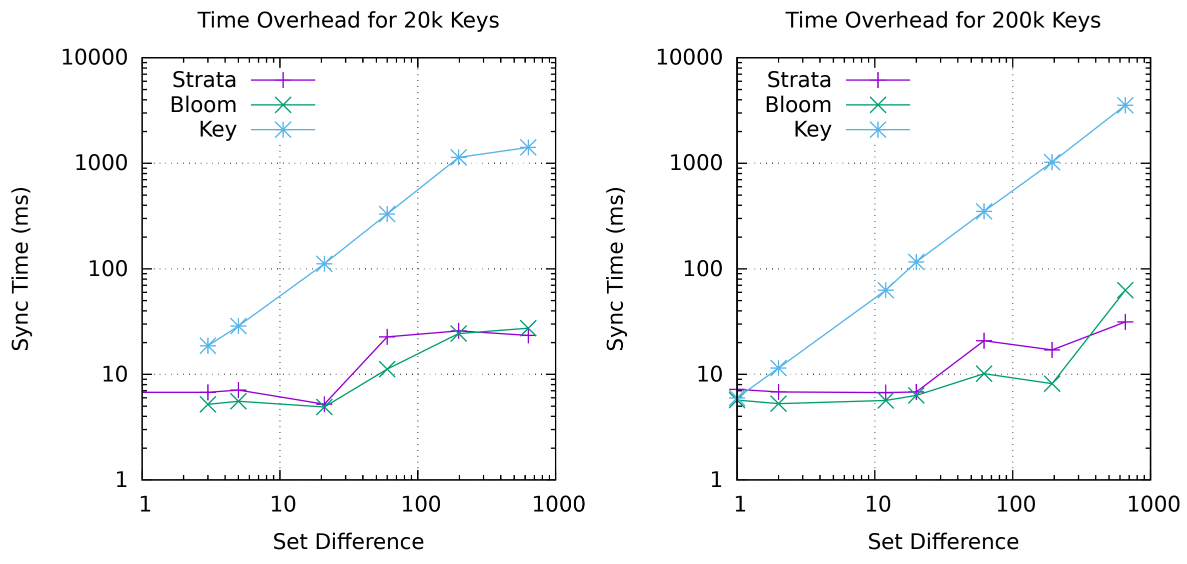
[9] ab – Apache HTTP server benchmarking tool. `https://httpd.apache.org/docs/2.4/programs/ab.html`.

[10] David Shaw. The OpenPGP keyserver protocol (HKP). Internet-Draft draft-shaw-openpgp-hkp-00.txt, IETF Secretariat, 3 2003.



(a) (b)

Figure 1: Synchronization data sent for keyservers with approximately 20k and 200k entries.



(a) (b)

Figure 2: Synchronization time for keyservers with approximately 20k and 200k entries.