


Minisketch: a library for [BCH](#)-based set reconciliation

`libminisketch` is an optimized standalone MIT-licensed library with C API for constructing and decoding *set sketches*, which can be used for compact set reconciliation and other applications. It is an implementation of the PinSketch^[1] algorithm. An explanation of the algorithm can be found [here](#). 

Sketches for set reconciliation

Sketches, as produced by this library, can be seen as "set checksums" with two peculiar properties:

- Sketches have a predetermined capacity, and when the number of elements in the set is not higher than the capacity, `libminisketch` will always recover the entire set from the sketch. A sketch of b -bit elements with capacity c can be stored in bc bits.
- The sketches of two sets can be combined by adding them (XOR) to obtain a sketch of the [symmetric difference](#) between the two sets (*i.e.*, all elements that occur in one but not both input sets).

This makes them appropriate for a very bandwidth-efficient set reconciliation protocol. If Alice and Bob each have a set of elements, and they suspect that the sets largely but not entirely overlap, they can use the following protocol to let both parties learn all the elements:

- Alice and Bob both compute a sketch of their set elements.
- Alice sends her sketch to Bob.
- Bob combines the two sketches, and obtains a sketch of the symmetric difference.
- Bob tries to recover the elements from the difference sketch.
- Bob sends every element in the difference that he has to Alice.

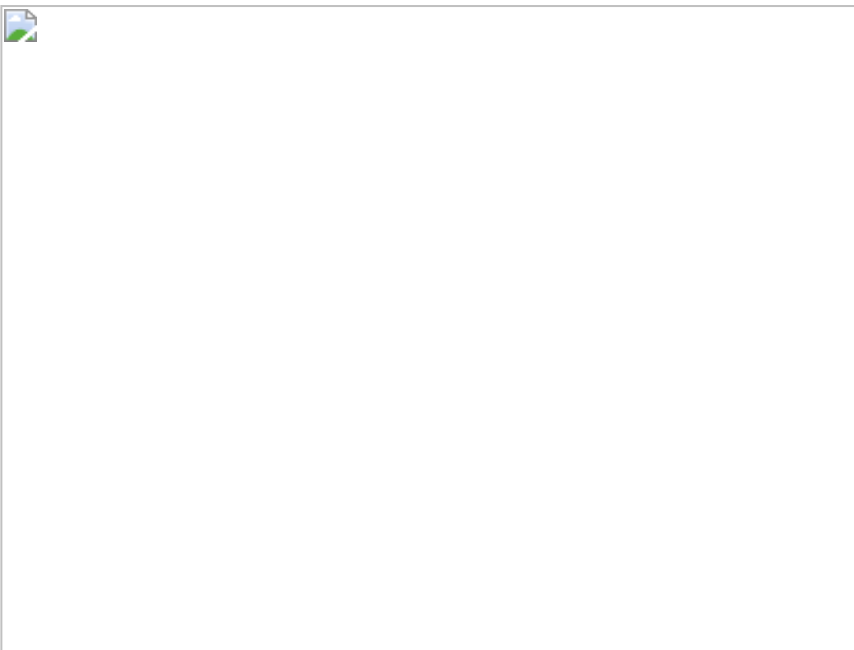
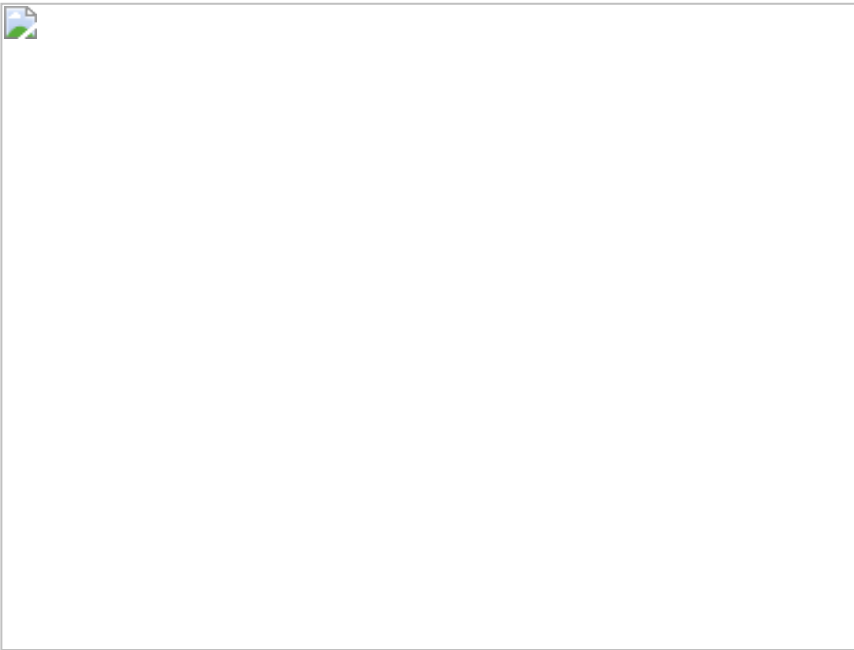
This will always succeed when the size of the difference (elements that Alice has but Bob doesn't plus elements that Bob has but Alice doesn't) does not exceed the capacity of the sketch that Alice sent. The interesting part is that this works regardless of the actual set sizes—only the difference matters.

If the elements are large, it may be preferable to compute the sketches over *hashes* of the set elements. In that case an additional step is added to the protocol, where Bob also sends the hash of every element he does not have to Alice, who responds with the requested elements.

The doc/ directory has additional [tips for designing reconciliation protocols using libminisketch](#).

Evaluation





The first graph above shows a benchmark of `libminisketch` against three other set reconciliation algorithms/implementations. The benchmarks were performed using a single core on a system with an Intel Core i7-7820HQ CPU with clock speed locked at 2.4 GHz. The diagram shows the time needed for merging of two sketches and decoding the result. The creation of a sketch on the same machine takes around 5 ns per capacity and per set element. The other implementations are:

- [pinsketch](#) , the original PinSketch implementation.
- [cpisync](#) , a software project which implements a number of set reconciliation algorithms and protocols. The included benchmark analyzes the non-probabilistic version of the original CPISync algorithm^[5] only.
- A high-performance custom IBLT implementation using 4 hash functions and 32-bit checksums.

For the largest sizes currently of interest to the authors, such as a set of capacity 4096 with 1024 differences, `libminisketch` is forty-nine times faster than `pinsketech` and over eight thousand times faster than `cpisync`. `libminisketch` is fast enough on realistic set sizes for use on high-traffic network servers where computational resources are limited.

Even where performance is latency-limited, small minisketches can be fast enough to improve performance. On the above i7-7820HQ, a set of 2500 30-bit entries with a difference of 20 elements can be communicated in less time with a minisketch than sending the raw set so long as the communications bandwidth is 1 gigabit per second or less; an eight-element difference can be communicated in better than one-fifth the time on a gigabit link.

The second graph above shows the performance of the same algorithms on the same system, but this time keeping the capacity constant at 128, while varying the number of differences to reconcile between 1 and 128. It shows how `cpisync`'s reconciliation speed is mostly dependent on capacity, while `pinsketech` / `libminisketch` are more dependent on number of differences.

The third graph above shows the size overhead of a typical IBLT scheme over the other algorithms (which are near-optimal bandwidth), for various levels of failure probability. IBLT takes tens of times the bandwidth of `libminisketch` sketches when the set difference size is small and the required failure rate is low.

The fourth graph above shows the effect of the field size on speed in `libminisketch`. The three lines correspond to:

- CLMUL 64-bit: Intel Core i7-7820HQ system at 2.4 GHz
- Generic 64-bit: POWER9 CP9M06 system at 2.8 GHz (Talos II)
- Generic 32-bit: Cortex-A53 at 1.2 GHz (Raspberry Pi 3B)

It shows how CLMUL implementations are faster for certain fields (specifically, field sizes for which an irreducible polynomial of the form $x^b + x + 1$ over $GF(2)$ exists, and to a lesser extent, fields which are a multiple of 8 bits). It also shows how (for now) a significant performance drop exists for fields larger than 32 bits on 32-bit platforms. Note that the three lines are not at the same scale (the Raspberry Pi 3B is around 10x slower for 32-bit fields than the Core i7; the POWER9 is around 1.3x slower).

Below we compare the PinSketch algorithm (which `libminisketch` is an implementation of) with other set reconciliation algorithms:

Algorithm	Sketch size	Decode success	Decoding complexity	Difference type	Secure sketch
CPISync ^[2]	$(b+1)c$	Always	$O(n^3)$	Both	Yes
PinSketch ^[1]	bc	Always	$O(n^2)$	Symmetric only	Yes
IBLT ^{[6][7]}	αbc (see graph 3)	Probabilistic	$O(n)$	Depends	No

- **Sketch size:** This column shows the size in bits of a sketch designed for reconciling c different b -bit elements. PinSketch and CPISync have a near-optimal^[11] communication overhead, which in practice means the sketch size is very close (or equal to) bc bits. That is the same size as would be needed to transfer the elements of the difference naively (which is remarkable, as the difference isn't even known by the sender). For IBLT there is an overhead factor α , which depends on various design parameters, but is often between 2 and 10.
- **Decode success:** Whenever a sketch is designed with a capacity not lower than the actual difference size, CPISync and PinSketch guarantee that decoding of the difference will always succeed. IBLT always has a

chance of failure, though that chance can be made arbitrarily small by increasing the communication overhead.

- **Decoding complexity:** The space savings achieved by near-optimal algorithms come at a cost in performance, as their asymptotic decode complexity is quadratic or cubic, while IBLT is linear. This means that using near-optimal algorithms can be too expensive for applications where the difference is sufficiently large.
- **Difference type:** PinSketch can only compute the symmetric difference from a merged sketch, while CPISync and IBLT can distinguish which side certain elements were missing on. When the decoder has access to one of the sets, this generally doesn't matter, as he can look up each of the elements in the symmetric difference with one of the sets.
- **Secure sketch:** Whether the sketch satisfies the definition of a secure sketch^[1], which implies a minimal amount about a set can be extracted from a sketch by anyone who does not know most of the elements already. This makes the algorithm appropriate for applications like fingerprint authentication.

Building

The build system is very rudimentary for now, and [improvements](#) are welcome.

The following may work and produce a `libminisketch.a` file you can link against:

```
git clone https://github.com/sipa/minisketch
cd minisketch
./autogen.sh && ./configure && make
```

Usage

In this section Alice and Bob are trying to find the difference between their sets. Alice has the set $[3000 \dots 3009]$, while Bob has $[3002 \dots 3011]$.

First, Alice creates a sketch:

```
#include <stdio.h>
#include <assert.h>
#include "../include/minisketch.h"
int main(void) {

    minisketch *sketch_a = minisketch_create(12, 0, 4);
```

The arguments are:

- The field size b , which specifies the size of the elements being reconciled. With a field size b , the supported range of set elements is the integers from 1 to $2^b - 1$, inclusive. Note that elements cannot be zero.
- The implementation number. Implementation 0 is always supported, but more efficient algorithms may be available on some hardware. The serialized form of a sketch is independent of the implementation, so different implementations can interoperate.
- The capacity c , which specifies how many differences the resulting sketch can reconcile.

Then Alice adds her elements to her sketch. Note that adding the same element a second time removes it again, as sketches have set semantics, not multiset semantics.

```
for (int i = 3000; i < 3010; ++i) {
    minisketch_add_uint64(sketch_a, i);
}
```

The next step is serializing the sketch into a byte array:

```
size_t sersize = minisketch_serialized_size(sketch_a);
assert(sersize == 12 * 4 / 8); // 4 12-bit values is 6 bytes.
unsigned char *buffer_a = malloc(sersize);
minisketch_serialize(sketch_a, buffer_a);
minisketch_destroy(sketch_a);
```

The contents of the buffer can then be submitted to Bob, who can create his own sketch:

```
minisketch *sketch_b = minisketch_create(12, 0, 4); // Bob's own sketch
for (int i = 3002; i < 3012; ++i) {
    minisketch_add_uint64(sketch_b, i);
}
```

After Bob receives Alice's serialized sketch, he can reconcile:

```
sketch_a = minisketch_create(12, 0, 4); // Alice's sketch
minisketch_deserialize(sketch_a, buffer_a); // Load Alice's sketch
free(buffer_a);

// Merge the elements from sketch_a into sketch_b. The result is a sketch_b
// which contains all elements that occurred in Alice's or Bob's sets, but not
// in both.
minisketch_merge(sketch_b, sketch_a);

uint64_t differences[4];
ssize_t num_differences = minisketch_decode(sketch_b, 4, differences);
minisketch_destroy(sketch_a);
minisketch_destroy(sketch_b);
if (num_differences < 0) {
    printf("More than 4 differences!\n");
} else {
    ssize_t i;
    for (i = 0; i < num_differences; ++i) {
        printf("%u is in only one of the two sets\n", (unsigned)differences[i]);
    }
}
}
```

In this example Bob would see output such as:

```
$ gcc -std=c99 -Wall -Wextra -o example ./doc/example.c -Lsrc/ -lminisketch -lstdc++
&& ./example
3000 is in only one of the two sets
```

```
3011 is in only one of the two sets
3001 is in only one of the two sets
3010 is in only one of the two sets
```

The order of the output is arbitrary and will differ on different runs of `minisketch_decode()`.

Applications

Communications efficient set reconciliation has been proposed to optimize Bitcoin transaction distribution^[8], which would allow Bitcoin nodes to have many more peers while reducing bandwidth usage. It could also be used for Bitcoin block distribution^[9], particularly for very low bandwidth links such as satellite. A similar approach (CPSync) is used by PGP SKS keyservers to synchronize their databases efficiently. Secure sketches can also be used as helper data to reliably extract a consistent cryptographic key from fuzzy biometric data while leaking minimal information^[11]. They can be combined with [dcnets](#) to create cryptographic multiparty anonymous communication^[10].

Implementation notes

`libminisketch` is written in C++11, but has a [C API](#) for compatibility reasons.

Specific algorithms and optimizations used:

- Finite field implementations:
 - A generic implementation using C unsigned integer bit operations, and one using the [CLMUL instruction](#) where available. The latter has specializations for different classes of fields that permit optimizations (those with trinomial irreducible polynomials, and those whose size is a multiple of 8 bits).
 - Precomputed tables for (repeated) squaring, and for solving equations of the form $x^2 + x = a$ ^[2].
 - Inverses are computed using an [exponentiation ladder](#)^[12] on systems where multiplications are relatively fast, and using an [extended GCD algorithm](#) otherwise.
 - Repeated multiplications are accelerated using runtime precomputations on systems where multiplications are relatively slow.
 - The serialization of field elements always represents them as bits that are coefficients of the lowest-weight (using lexicographic order as tie breaker) irreducible polynomials over $GF(2)$ (see [this list](#)), but for some implementations they are converted to a different representation internally.
- The sketch algorithms are specialized for each separate field implementation, permitting inlining and specific optimizations while avoiding dynamic allocations and branching costs.
- Decoding of sketches uses the [Berlekamp-Massey algorithm](#)^[3] to compute the characteristic polynomial.
- Finding the roots of polynomials is done using the Berlekamp trace algorithm with explicit formula for quadratic polynomials^[4]. The root finding is randomized to prevent adversarial inputs that intentionally trigger worst-case decode time.
- A (possibly) novel optimization combines a test for unique roots with the Berlekamp trace algorithm.

Some improvements that are still TODO:

- Explicit formulas for the roots of polynomials of higher degree than 2
- Subquadratic multiplication and modulus algorithms
- The [Half-GCD algorithm](#) for faster GCDs
- An interface for incremental decoding: most of the computation in most failed decodes can be reused when attempting to decode a longer sketch of the same set
- Platform specific optimizations for platforms other than x86
- Avoid using slow `uint64_t` for calculations on 32-bit hosts

- Optional IBLT / Hybrid and set entropy coder under the same interface

References

- [1] Dodis, Ostrovsky, Reyzin and Smith. *Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data*. SIAM Journal on Computing, volume 38, number 1, pages 97-139, 2008). [\[URL\]](#) [\[PDF\]](#)
- [5] A. Trachtenberg, D. Starobinski and S. Agarwal. *Fast PDA synchronization using characteristic polynomial interpolation*. Proceedings, Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies, New York, NY, USA, 2002, pp. 1510-1519 vol.3. [\[PDF\]](#)
- [2] Cherly, Jørgen, Luis Gallardo, Leonid Vaserstein, and Ethel Wheland. *Solving quadratic equations over polynomial rings of characteristic two*. Publicacions Matemàtiques (1998): 131-142. [\[PDF\]](#)
- [3] J. Massey. *Shift-register synthesis and BCH decoding*. IEEE Transactions on Information Theory, vol. 15, no. 1, pp. 122-127, January 1969. [\[PDF\]](#)
- [4] Bhaskar Biswas, Vincent Herbert. *Efficient Root Finding of Polynomials over Fields of Characteristic 2*. 2009. hal-00626997. [\[URL\]](#) [\[PDF\]](#)
- [6] Eppstein, David, Michael T. Goodrich, Frank Uyeda, and George Varghese. *What's the difference?: efficient set reconciliation without prior context*. ACM SIGCOMM Computer Communication Review, vol. 41, no. 4, pp. 218-229. ACM, 2011. [\[PDF\]](#)
- [7] Goodrich, Michael T. and Michael Mitzenmacher. *Invertible bloom lookup tables*. 2011 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton) (2011): 792-799. [\[PDF\]](#)
- [8] Maxwell, Gregory F. [Blockonly mode BW savings, the limits of efficient block xfer, and better relay](#) Bitcointalk 2016, [Technical notes on mempool synchronizing relay](#) #bitcoin-wizards 2016.
- [9] Maxwell, Gregory F. [Block network coding](#), Bitcoin Wiki 2014, [Technical notes on efficient block xfer](#) #bitcoin-wizards 2015.
- [10] Ruffing, Tim, Moreno-Sanchez, Pedro, Aniket, Kate, *P2P Mixing and Unlinkable Bitcoin Transactions* NDSS Symposium 2017 [\[URL\]](#) [\[PDF\]](#)
- [11] Y. Misky, A. Trachtenberg, R. Zippel. *Set Reconciliation with Nearly Optimal Communication Complexity*. Cornell University, 2000. [\[URL\]](#) [\[PDF\]](#)
- [12] Itoh, Toshiya, and Shigeo Tsujii. "A fast algorithm for computing multiplicative inverses in GF (2^m) using normal bases." Information and computation 78, no. 3 (1988): 171-177. [\[URL\]](#)