A Secure Sketch for Set Reconciliation

Soren Johnson
Professor Leonid Reyzin
May 7, 2005

Abstract

In this paper we present an implementation of an algorithm for creating a secure sketch for the set difference as defined in [1]. This sketch is a short, efficiently computable description of a set such that, given a second set that differs only in a small percentage of elements, the original can be reconstructed. However, this sketch is secure in the sense that in the absence of such a similar set, as little information as possible can be obtained about the elements of the original. This work has applications in cryptography and database synchronization.

1 Introduction

Consider the following scenario: given a password that is prone to error (e.g. a biometric measurement such as a fingerprint or a voiceprint) we would like to be able to authenticate a user without storing the password. Using public-key cryptography, this can be done for a normal password, but for an error-prone password, we need some sort of “sketch” of the password that allows a similar password with only a few errors to be transformed into the original. However, this sketch must contain as little information as possible about the original password so that someone who does not have access to a password within a given similarity will not be able to forge a similar password. In particular, we consider passwords represented as sets (techniques presented in [1] describe how to transform arbitrary strings into sets). The algorithm we present creates a secure sketch[1] of a set A that gives as little information about A as possible, but, given a similar set B, there exists a way to extract A from B and the sketch of A. This allows the authentication of an original set, without allowing forgery. Secure sketches can be used to build fuzzy extractors [1], which reliably extract the same uniformly random string from the input, even if the input changes slightly.

An entirely different application for the set difference is presented in [2]. The authors consider the periodic synchronization of databases with nearly optimal communication complexity. In other words, they consider the problem of reconciling two databases that have almost all the same elements (e.g. pgp_key databases) while exchanging as little information as possible. This scenario is slightly different from ours for several reasons. First, though databases
may be known to be similar, the upper bound on the difference may not be known and thus the
generator presented in [2] is expanded to accommodate this lack of exact information. Second,
since the databases are updated periodically, the authors consider ways to allow for a periodic
update of the database “sketch” rather than entirely recomputing it at each reconciliation.
However, the core algorithm performs much the same task as the one presented below.

2 Secure Sketch

A secure sketch[1] consists of two algorithms: a Sketch() procedure to create the sketch
of the original set and a Reconcile() procedure to retrieve the original set, given a sketch of the
original and a second set within a given similarity of the first. In other words, given sets A and B
both of size n that have at least n \(- t\) elements in common, Reconcile(B, Sketch(A)) = A. We
consider sets with an equal number of elements and we consider these elements to be positive
integers bounded from above by some maximum integer. The approach we follow is based on
the “modified Juels-Sudan” approach presented in [1] which was created using ideas from [6].

First, for any given sets A and B we must determine the following constants:

<table>
<thead>
<tr>
<th>q</th>
<th>a prime modulus larger than the largest element of A or B</th>
</tr>
</thead>
<tbody>
<tr>
<td>n</td>
<td>the number of elements in A and B</td>
</tr>
<tr>
<td>t</td>
<td>the maximum number of differences between A and B</td>
</tr>
</tbody>
</table>

By differences between A and B, we refer to the maximum number of elements in A that
are not in B (or equivalently the maximum number of elements in B that are not in A). We
define Sketch(A) to be the higher-order \(2t + 1\) coefficients of \(X_A(x)\), where \(X_A(x)\) is the
characteristic polynomial of A, i.e. the monic, square-free polynomial of degree \(n\) whose roots
are exactly the elements of A. Strictly speaking, we need not send the top coefficient since it will
always be 1, but for simplicity we will assume that we do send the leading 1. The pseudocode is
as follows:

```
Sketch(A = \{a_0, a_1, ... a_{n-1}\}) {
    X_A = 1
    FOR i = 0 TO n - 1
        X_A = X_A * (x - a_i)
    RETURN top 2t + 1 coefficients of X_A
}
```
We then claim that given this sketch and a set $B$ with no more than $t$ differences from $A$, we can completely recover $A$. Our strategy in doing this will be to find the lower-order coefficients of $O_A(x)$ so that we can recover $O_A(x)$ and then use a factorization algorithm to extract the elements of $A$. Let $H(x)$ be the polynomial of degree $n$ (and thus having $n+1$ coefficients) whose $2t+1$ higher-order coefficients are the same as that of $O_A(x)$ and whose $(n-2t)$ lower-order coefficients are $0$. Let $L(x)$ be the polynomial of degree $(n-2t-1)$ whose coefficients are exactly the $(n-2t)$ lower-order coefficients of $O_A(x)$. In other words, $H(x)$ is the polynomial given to us by the sketch and $L(x)$ is the polynomial we need to find because $O_A(x) = H(x) + L(x)$. However, we also have a set $B$ and we know that $B$ is the same as $A$ except for no more than $t$ elements. In other words, $O_A(x) = O_B(x)$ for all $x \in A \cap B$ meaning that $L(x) = X_B(x) - H(x)$ for at least $(n-t)$ elements of $B$.

This is a similar scenario to decoding a Reed-Solomon error-correcting code, as discussed below. A typical decoding scheme for this type of code tries to recover a polynomial $p(x)$ of degree $< (n-2t)$ given a set of $n$ points $X$ and a set of $n$ points $Y$ where $p(x_i) = y_i$ for at least $n-t$ of the $x_i$ in $X$ and $y_i$ in $Y$. This is exactly what we need because we have a set $B$ and can compute $X_B(b_i) - H(b_i)$ for each $b_i \in B$ since we know $X_B(x)$ and $H(x)$. Since $O_A(b_i) = X_B(b_i)$ for at least $(n-t)$ elements of $B$, then the collections of points $X = B$ and $Y = \{X_B(b_i) - H(b_i) \mid b_i \in B\}$ have the property that $L(x_i) = y_i$ for all $x_i \in X$ and $y_i \in Y$ except for no more than $t$ elements.

Also note that $X_B(b_i) = 0$ for each $b_i \in B$ by definition and so to compute $X_B(b_i) - H(b_i)$ for each $b_i \in B$, we need only compute $-H(b_i)$. Thus, the reconciliation procedure is as follows:

```plaintext
Reconcile(H, B = \{b_0, b_2, \ldots b_{n-1}\}) {
    FOR i = 0 TO n - 1
        y_i = -H(b_i)
    L = Decode(B, Y = \{y_0, y_2, \ldots y_{n-1}\})
    RETURN Factor(H + L)
}
```

In section 5, we will replace this vague $Decode()$ function with a definite algorithm to complete our construction of a rigorous method for set reconciliation. However, to do this, we will need to understand some concepts of error-correcting codes.

## 3 Reed-Solomon Codes

For this application, we consider Reed-Solomon codes. These are a special type of BCH
code that involve treating the data to be encoded as a polynomial, i.e. converting the data to a series of characters and considering the polynomial \( p(x) \) whose coefficients are exactly those characters. Then, given \( k \) characters, we can correct up to \( t \) errors by evaluating the polynomial on \( n = k + 2t \) standard points \( \{x_1, \ldots, x_n\} \) that are known to the receiver. Then, when these points \( \{y_1, \ldots, y_n\} \) are transmitted and contain no more than \( t \) errors, the receiver knows that there exists a polynomial \( p(x) \) of degree \(< k\) for which \( p(x_i) = y_i \) for at least \( k + t \) of the \((x_i, y_i)\) pairs. As shown below, the receiver can then uniquely determine \( p(x) \) and thus extract the error-free original data.

However, typical Reed-Solomon implementations make assumptions that render them useless for our purposes. First, most implementations consider a small finite field, usually of size \( n \). Since most applications require only that there be \( n \) distinct points in the field on which to evaluate and that there be enough points to accommodate all possible characters (of which there are usually fewer than \( n \)), then the field size need not be larger than \( n \). However, in our application, we consider sets whose data elements are not simply small characters, but are themselves large objects of data (encoded as positive integers). This means that our field size \( q \) must be large enough to accommodate all possible data items and so must be orders of magnitude larger than the typical field size.

Also, most implementations fix the points \( \{x_1, \ldots, x_n\} \) as some specific elements of the field. This makes sense in error-correcting codes since these points must already be known to the sender and receiver before any communication occurs, but is insufficient for our purposes since the set of first coordinates \( B \) given to our \texttt{Decode()} function above is an arbitrary set. Furthermore, most implementations consider \( \{x_1, \ldots, x_n\} \) to be powers of a generator of the field which has efficiency benefits for field of small size, but requires a discrete logarithm and so is totally infeasible for a field size of any real magnitude.

In the next section we describe a decoding algorithm that is not affected by these issues. For another, so far unsuccessful, but potentially more efficient approach, see Appendix A.

4 Welch-Berlekamp Algorithm

To implement the decoding required, we consider the Welch-Berlekamp algorithm. This algorithm is \( O(n^3) \) and while faster algorithms exist (see Appendix A), it may not be possible to adapt these algorithms to the specific requirements of our application. This algorithm is conceptually simpler and makes fewer assumptions about the nature of the points to be decoded. As described in [3], the Welch-Berlekamp algorithm tries to determine \( p(x) \) given sets \( X \) and \( Y \) by defining \( E(x) \) to be a polynomial of degree \( t \) satisfying \( E(x_i) = 0 \) if \( p(x_i) \neq y_i \), and a polynomial \( N(x) = p(x) \cdot E(x) \). In other words, \( E(x) = \Pi(x - e_i) \) where each error location must be one of the \( e_i \) (though each \( e_i \) may not necessarily be an error location if there are fewer than \( t \) errors). This means that \( E(x) \) is not necessarily unique nor does it exactly specify the error locations if there
are fewer than $t$ errors, but in this algorithm that is not important. All that really matters are the following facts[3]:

\[\deg_x(E) = t\]
\[\deg_x(N) < k + t\]
\[\forall i \in [n] \ (N(x_i) = y_i E(x_i))\]

In this context $k$ is the number of coefficients of the polynomial we wish to find, and is equal to $n - 2t$. These statements are actually sufficient to find such an $E$ and $N$, after which we can simply calculate $p(x) = N(x) / E(x)$ since we defined $N(x)$ to be $p(x) \cdot E(x)$. Even though there may be many such $E(x)$ that satisfy these equations (and thus many such $N(x)$), $p(x)$ is always unique as shown in [3], and so any solutions for $E(x)$ and $N(x)$ provide $p(x)$. To find an $E(x)$ and an $N(x)$ with the above properties requires solving the following system of linear equations:

\[
\begin{pmatrix}
1 & x_0 & x_0^2 & \ldots & x_0^{k-1} \\
1 & x_1 & x_1^2 & \ldots & x_1^{k-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{k-1}
\end{pmatrix}
\begin{pmatrix}
N_0 \\
N_1 \\
\vdots \\
N_{k-1}
\end{pmatrix}
= 
\begin{pmatrix}
y_0 & y_0 x_0 & y_0 x_0^2 & \ldots & y_0 x_0^{t-1} \\
y_1 & y_1 x_1 & y_1 x_1^2 & \ldots & y_1 x_1^{t-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
y_{n-1} & y_{n-1} x_{n-1} & y_{n-1} x_{n-1}^2 & \ldots & y_{n-1} x_{n-1}^{t-1}
\end{pmatrix}
\begin{pmatrix}
E_0 \\
E_1 \\
\vdots \\
E_{t-1}
\end{pmatrix}
\]

This equation is simply a matrix representation of the $n$ equations $N(x_i) = y_i E(x_i)$ in terms of the coefficients of $N$ and $E$. However, we also know that the leading coefficient of $E$ is not 0 because we defined $E$ to be of degree $t$. In fact, we may as well assume that $E$ is a monic polynomial since any polynomial that has a leading coefficient other than 1 can be normalized by dividing by that coefficient and its roots will be unaffected. Thus, if we let $E_t = 1$ and combine the matrices into a single larger matrix, we get:

\[
\begin{pmatrix}
1 & x_0 & x_0^2 & \ldots & x_0^{k-1} & -y_0 & -y_0 x_0 & -y_0 x_0^2 & \ldots & -y_0 x_0^{t-1} \\
1 & x_1 & x_1^2 & \ldots & x_1^{k-1} & -y_1 & -y_1 x_1 & -y_1 x_1^2 & \ldots & -y_1 x_1^{t-1} \\
\vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\
1 & x_{n-1} & x_{n-1}^2 & \ldots & x_{n-1}^{k-1} & -y_{n-1} & -y_{n-1} x_{n-1} & -y_{n-1} x_{n-1}^2 & \ldots & -y_{n-1} x_{n-1}^{t-1}
\end{pmatrix}
\begin{pmatrix}
N_0 \\
N_1 \\
\vdots \\
N_{k-1}
\end{pmatrix}
= 
\begin{pmatrix}
y_0 x_0^t \\
y_1 x_1^t \\
\vdots \\
y_{n-1} x_{n-1}^t
\end{pmatrix}
\begin{pmatrix}
E_0 \\
E_1 \\
\vdots \\
E_{t-1}
\end{pmatrix}
\]

As shown in [3], there must exist a solution for $E$ and $N$ and so we can use this system of $n$ equations with $n$ unknowns to uniquely determine the coefficients of a valid $E$ and $N$. We
know \( E_i = 1 \), so we can then derive \( E \) and \( N \) and simply divide \( N \) by \( E \) to get \( p \). The pseudocode is as follows (the functions \texttt{SetCoefficient()} and \texttt{Solve()} are described below):

\[
\text{Welch-Berlekamp}\{\{x_0, x_2, \ldots x_{n-1}\}, \{y_0, y_2, \ldots y_{n-1}\}\} \{
\text{FOR } i = 0 \text{ TO } n - 1 \\
\quad \text{FOR } j = 0 \text{ TO } k + t - 1 \\
\quad \quad A[i][j] = (x_i)^j \\
\quad \text{FOR } j = 0 \text{ TO } t - 1 \\
\quad \quad A[i][k+t+j] = -y_i(x_i)^j \\
\quad A[i][n] = y_i(x_i)^t \\
\}
\]

\texttt{Solve(A)}

\[
\text{FOR } i = 0 \text{ TO } k + t - 1 \\
\quad \texttt{SetCoefficient(N, i, A[i][n])} \\
\]

\[
\text{FOR } i = 0 \text{ TO } t - 1 \\
\quad \texttt{SetCoefficient(E, i, A[k+t+i][n])} \\
\quad \texttt{SetCoefficient(E, t, 1)} \\
\]

\texttt{RETURN (N/E)}

\texttt{SetCoefficient(P, i, a)} \texttt{simply sets the } \textit{i} \texttt{th coefficient of } \texttt{P} \texttt{ to } \texttt{a}.

The \texttt{Solve()} function takes as input a solveable augmented matrix and returns an arbitrary solution. We can accomplish this by using Gaussian elimination to put the matrix in upper triangular form, setting free variables to an arbitrary value and finally solving the remaining system of equations through row reduction. We can now make good on our promise to complete the pseudocode of \texttt{Reconcile()} in section 2 by simply replacing \texttt{Decode(B, Y)} with \texttt{Welch-Berlekamp(B, Y)}, a call to the above function.

5 Implementation

To handle number theory applications, we chose to use a number theory code base called NTL[5]. The tools provided in the NTL code library include both a function to solve matrix equations such as the one above, and tools to do division and other polynomial operations. Also, we chose to use fields of residues modulo a prime number because of their simplicity. We conjecture that this application could be made more efficient by using fields of the form \( \text{GF}(2^m) \) and the code should allow for this modification, but unfortunately we were unable to test this.
The code consists of five modules:

<table>
<thead>
<tr>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>main.cpp</td>
<td>the user interface</td>
</tr>
<tr>
<td>sketch.cpp</td>
<td>Sketch()</td>
</tr>
<tr>
<td>reconcile.cpp</td>
<td>Reconcile()</td>
</tr>
<tr>
<td>welchberlekamp.cpp</td>
<td>Welch-Berlekamp()</td>
</tr>
<tr>
<td>sketchio.cpp</td>
<td>various set input/output functions</td>
</tr>
</tbody>
</table>

Each has an appropriately named header file (except for main.cpp) to link them together and there is a “constants.h” used by each of the modules. For information on how to compile and test the program, consult the README file contained with the code. The user interface to the executable is as follows:

Usage: sketch [-h] [-c infile (outfile)] [-r recfile sketchfile (outfile)] [-m digits (outfile)]

- **-h** prints this help text.
- **-c** creates a secure sketch of infile and writes it to outfile, if specified, 'sketch.out' otherwise.
- **-r** reconciles the set in recfile with the set sketched in sketchfile and writes it to outfile, if specified, 'rec.out' otherwise.
- **-m** creates a prime modulus with a given number of digits and writes it to outfile, if specified, 'mod.out' otherwise.

The -m option is not necessary and has nothing to do with the algorithm described above, but provides a useful tool for generating a large prime number to use as a modulus. Note that files representing a set of n elements \{a_1, a_2, ... a_n\} in a field modulo q with a similarity of t must be in the form q t \[a_1, a_2, ... a_n\].

6 Conclusions

In this paper, we explored various methods for computing a secure sketch of a set that allows it to be completely reconstructed, given a set within a certain similarity. We presented an algorithm for doing this that relied on a Reed-Solomon decoding procedure. We discussed the issues involved in Reed-Solomon decoding and presented the Welch-Berlekamp algorithm as a way to perform the required decoding. Finally, we described and documented an implementation.
of an application that performs simple sketch and reconcile procedures.

References


A Berlekamp-Massey Algorithm

Our first attempt at decoding involved the Berlekamp-Massey algorithm. This algorithm takes a set Y of n points that correspond to the evaluation of a polynomial of degree n – 2t, of which at most t points are erroneous and determines the error points. We accomplish this by computing the error-locator polynomial, a polynomial that can be factored to uniquely determine which points in Y cannot correspond to an evaluation of a polynomial of degree n – 2t. This algorithm, while conceptually complex, is known to be efficient. It involves first computing the 2t syndromes of the of the code (discussed below) and then iteratively computing the error-locator polynomial. Once the error-locator polynomial is known, it can be factored to determine which points of Y are incorrect. If we then remove those points we can determine the correct polynomial by simple interpolation of the correct points. In many cases other methods, such as the Forney algorithm, can do this final step even more efficiently.

In this particular algorithm, the error-locator polynomial is defined to be the polynomial whose roots are exactly the inverse of the points in which there is an error, i.e. \( \prod (1 - (y_i)x) \) for
each point $y_i \in Y$ at which there is an error. The process involves scalar values $k$ and $L$, an array of $2t$ scalars $\{\Delta_0, \ldots, \Delta_{2t-1}\}$, a polynomial $B(x)$ and an array of $2t + 1$ polynomials $\{\Lambda^{(0)}(x), \ldots, \Lambda^{(2t)}(x)\}$ where $\Lambda^{(2t)}$ is the final error locator polynomial. This $k$ is merely a counting variable and should not be confused by the constant $k = n - 2t$ used elsewhere. The pseudocode is as follows:

```
Berlekamp-Massey(Y) {
    \{S_0, \ldots, S_{2t-1}\} = ComputeSyndromes(Y)
    \Lambda^{(0)} = 1
    B = 1
    L = 0
    k = 1
    DO
        \Delta_{k-1} = \sum_{i=0}^{k-1} (\Lambda^{(k-1)}(i) \cdot S_{k-1})
        \Lambda^{(k)} = \Lambda^{(k-1)} - (\Delta_{k-1} \cdot (x \cdot B))
        IF $\Delta_{k-1} \neq 0$ AND $2L \leq k - 1$
            L = k - L
            B = $(\Delta_{k-1})^{-1} \cdot \Lambda^{(k-1)}$
        ELSE
            B = $x \cdot B$
        WHILE (k \leq 2t)
    RETURN $\Lambda^{(2t)}$
}
```

However, a problem arises in our application, because it is not clear how to create $2t$ syndromes from the $n$ points of $A$. The usual definition of syndromes assumes that the polynomial in question has been evaluated on powers of a generator for the field in which we are working instead of arbitrary points, as is our case. It is not clear how this definition can be adapted for the functionality we require, nor is it obvious that this is even possible.