Applications of the Chinese Remainder Theorem

L. Felipe Martins
Department of Mathematics
Cleveland State University
l.martins@csuohio.edu

February 9, 2009

1 Database Access Protection

In computer science, we can think of a simple database is a sequence of records stored in a file. Each record contains the data of interest to a particular individual, for instance, a customer in a company or a student at an university. Notice that an “individual” does not have to be a “person”. A city, for example, might keep a database with all its buildings.

Each record contains several fields, consisting of pieces of information related to the individual represented by the record. For a database of employees of a company, for example, the fields might store the employee’s name, SSN, hours worked and salary. We assume that the number and nature of fields in a record is fixed (that is, putting it loosely, all records have the same fields).

In a computer, each field is, ultimately, a sequence of bits. The number of bits needed to represent a field is called its bit length. We assume that the bit length of each field is fixed.

Every bit sequence can be interpreted as the binary representation of an integer. We can thus, think of each field in a record as an integer. The bit length of the field determines the maximum size of the integer that has to be used to represent it. If the bit length is $l$, the largest integer is $2^l - 1$. 
Thus, we think of each record as a sequence of integers, $F_1, F_2, \ldots, F_n$, where we assume that $0 \leq F_n \leq B_i$.

For example, suppose that we have three data fields, with bit lengths 5, 10 and 7. Then, we would have the bounds: $B_1 = 2^5 - 1 = 31$, $B_2 = 2^{10} - 1 = 1023$ and $B_3 = 2^7 - 1 = 127$. One possible record would be:

```
10111 1011100101 0101000
```

This corresponds to the integers:

$$F_1 = 23, \quad F_2 = 741, \quad F_3 = 40$$

Usually, not all people in an organization has access to all fields of the database. For example, some could have access a persons name but not their salary. We can use the CRT to set up the database in such a way that access is given only to those that should have it.

To set up the database, we start by choosing distinct prime numbers $p_1, p_2, \ldots, p_n$ such that $p_i > B_i$ for $i = 1, \ldots, n$. For our example, we could find these primes using Sage with the function `random_prime`. These are kept secret and chosen to be large enough to guarantee the security of the system. We choose our primes to be up to 19 bits long:

```
sage: lprimes = [random_prime (2^20) for i in range (3)]
sage: print lprimes
[465331, 348917, 520549]
```

Notice that `random_prime(k)` returns a random prime that it is less than or equal $k$, so it is possible that the generated prime is not bigger than $B_i$. If this is the case, we simply generate another random prime until we get one of the appropriate size.

Now, given the integers $F_1, F_2, \ldots F_n$ describing a record, we use the CRT to find $x$ such that:

$$x \equiv F_i \pmod{p_i} \quad \text{for } 1 \leq i \leq n,$$

and the integer $x$ is stored in the database.

In our example, we would do, in Sage:

```
sage: record=[23,741,40]
sage: CRT_list (record, lprimes)
4352404389629276
```
This is the integer that is stored in the database to represent the record.

If someone needs access to field \( F_i \), we give this person the value of \( p_i \), and he/she can find the value needed by computing \( x \mod p_i \). Suppose, in our example, that we want to grant access to the second field. Then, we give \( p_2 = 348917 \), and the person needing access computes:

\[
sage: 4352404389629276 \mod 348917
\]

741

2 Secret Sharing

2.1 Statement of the problem

It is not infrequent that passwords are lost. Passwords, for obvious security reasons, are encrypted when written to persistent storage (such as a hard disk). So, devising a secure method for recovering passwords in case they are lost is a non-trivial problem.

In this section study methods for secret sharing, which allow a group of entities to share some secret information, and yet there are restrictions on how these entities can access the information.

For a concrete example, suppose that a company has some information that must be shared among its five highest-level officials. The information, for example, might be the password that gives access to all of the company’s financial data.

Giving a copy of the information to each of the five officials is obviously not a good solution. An official might store the information unencrypted in his/her computer, or simply write it in a piece of paper and tape it under a desk (not uncommon!). A malicious entity would have to compromise only one of the officers to obtain the information. Also, any of the officers could access the information and use it in inappropriate ways.

One solution to this problem is to split the secret information in five parts, and each part is printed in a different sheet of paper. Each officer keeps only one of the sheets, so none of the officers has individual access to the whole secret. If the secret information has to be retrieved, and the five officers agree to it, they can do so by revealing the piece of the information each one has.

This can be easily implemented mathematically using the CRT. Suppose that the secret information is encoded in an integer \( S \), that has to be shared among \( n \) entities, in such a way that it can only be recovered if all entities agree to it.
The system relies on a *dealer*, that sets up the framework for sharing. In the examples in this chapter we assume that the dealer is honest, and has full knowledge of the secret to be shared. (This assumption can be removed by more sophisticated protocols.)

The dealer chooses a set of pairwise coprime integers $a_1, a_2, \ldots, a_N$ such that $S < a_1 a_2 \cdots a_n$, and then computes:

$$S_i = S \mod a_i \quad \text{for } i = 1, \ldots, n.$$ 

Then, entity $i$ is given the integer $S_i$, which is called a *share*, or *shadow* of the secret.

If there is need to recover the secret, and all entities agree to it, each entity reveals the its value of $S_i$, and use the CRT to find the only $x$ such that:

$$x \equiv S_1 \pmod{a_1}$$
$$x \equiv S_2 \pmod{a_2}$$
$$\vdots$$
$$x \equiv S_n \pmod{a_n}$$

with $0 \leq x < a_1 a_2 \cdots a_n$. Then, by the uniqueness part of the CRT, we have $x \equiv S \pmod{a_1 a_2 \cdots a_n}$, and since both $x$ and $S$ are smaller than the modulus $a_1 a_2 \cdots a_n$, we must have $x = S$.

This scheme, however, has many flaws. It may be impossible, or impractical, to get all entities together in case the secret information is needed. Even worse, if a single entity loses its share of the information, it may be impossible to recover it.

To address these problems, one can use a *threshold scheme*. This kind of scheme is defined in terms of a pair of integers $(k, n)$, with $k < n$. Integer $n$ is, as above, the number of entities that share the secret. Integer $k$ is called the *threshold*, and the scheme is characterized in the following way:

Any subset of $k$ entities can recover the secret $S$, but no set of fewer than $k$ entities can do so.

We now consider two examples of threshold schemes.
2.2 Mignotte’s Threshold Scheme

The first approach we describe is called Mignotte’s threshold scheme for secret sharing. It requires use of a sequence of integers with the following property:

**Definition 2.1.** A \((k,n)\)-Mignotte sequence is a sequence of pairwise coprime integers \(a_1 < a_2 < \cdots < a_n\) such that:

\[
a_{n-k+2}a_{n-k+3}\cdots a_n < a_1a_2\cdots a_k
\]  

(1)

In words, condition (1) says that the product of the largest \(k-1\) integers in the sequence is smaller than the product of the smallest \(k\) integers. It is easy to see that this implies that:

The product of any \(k-1\) integers in a Mignotte sequence is smaller than the product of any product of \(k\) integers in the same sequence

For example, suppose that \(n = 5\), \(k = 3\) and we consider the sequence:

\[
a_1 = 1301, \quad a_2 = 1427, \quad a_3 = 1669, \quad a_4 = 1873, \quad a_5 = 1979.
\]  

(2)

We chose the integers in the sequence to be prime, so they are obviously pairwise coprime. To verify that this is a \((2,5)\)-Mignotte sequence, we have to check that:

\[a_4a_5 < a_1a_2a_3.\]

This can be done in Sage as follows:

```sage
sage: k, n = 3, 5
sage: aseq = [1301,1427,1669,1873,1979]
sage: prod_smallest = prod(aseq[:k])
sage: prod_largest = prod(aseq[:n-k])
sage: print prod_largest, prod_smallest
1856527 3098543563
```

Now suppose that the secret \(S\) is an integer such that:

\[
a_{n-k+2}a_{n-k+3}\cdots a_n < S < a_1a_2\cdots a_k.
\]  

(3)

The shares, then, are given by:

\[S_i = S \mod a_i \quad \text{for} \ i = 1, \ldots, n.\]
Suppose that $k$ of the entities, $i_1, i_2, \ldots, i_k$ need to recover the secret. Then, they can use the CRT to find $x$ such that:

$$
\begin{align*}
x &\equiv S_{i_1} \pmod{a_{i_1}} \\
x &\equiv S_{i_2} \pmod{a_{i_2}} \\
& \vdots \\
x &\equiv S_{i_k} \pmod{a_{i_k}}
\end{align*}$$

with $0 \leq x < a_{i_1}a_{i_2}\cdots a_{i_k}$. Notice that, since $a_{i_1}a_{i_2}\cdots a_{i_k} > a_1a_2\cdots a_k > S$, we must have $x = S$.

For example, suppose that the secret is $S = 525706856$, and we use the sequence (2). Then, the shares are:

```python
sage: S = 525706856
sage: shares = [S % a for a in aseq]
sage: shares
[77, 56, 229, 708, 1338]
```

Suppose now that entities 2, 3 and 5 wish to recover the secret. Then, they solve:

$$
\begin{align*}
x &\equiv 56 \pmod{1427} \\
x &\equiv 229 \pmod{1669} \\
x &\equiv 1338 \pmod{1979}.
\end{align*}
$$

Using Sage (notice that we adjust the indices, since Python/Sage lists start at index 0):

```python
sage: ilist = [1, 2, 4]
sage: blist = [shares[i] for i in ilist]
sage: mlist = [aseq[i] for i in ilist]
sage: x = CRT_list(blist, mlist)
sage: x
525706856
```

Thus, the 3 entities can, working together, recover the secret. Now suppose that two of the entities, say 2 and 4, attempt to find the secret. They can still solve the CRT system (which now has only two equations):

```python
sage: ilist = [1, 3]
sage: blist = [shares[i] for i in ilist]
sage: mlist = [aseq[i] for i in ilist]
```
sage: x = CRT_list(blist, mlist)
sage: x
1843740

So, they get \( x = 1843740 \), and don’t directly recover \( S \). The reason they don’t get the right secret is that the solution of the CRT system is unique modulo \( a_2a_4 < a_4a_5 \), which is less than \( S \) by assumption.

Mignotte’s scheme, however, has a weakness: even though parties of fewer than \( k \) of the entities cannot recover the full secret, they can still gather some information about it. In the context of last example, parties 2 and 4, from their calculations, can conclude that:

\[
S = 1843740 \mod a_2a_4,
\]

that is, they know that the secret has the form \( S = 1843740 + ja_2a_4 = 1843740 + 2672771j \), for some integer \( j \). As a consequence, while the original number of possible values for the secret \( S \) is \( 3098543563 - 1856527 = 3096687036 \), entities 2 and 4 are able to reduce the number of possible secrets to:

\[
3096687036 \div 2672771 = 1158.
\]

In the next subsection, we examine an improvement to Mignotte’s scheme that deals with this weakness.

## 3 The Asmooth-Bloom Secret Sharing Scheme

Analogously to the Mignotte scheme, the Asmooth-Bloom scheme relies on the choice of an appropriate sequence of integers:

**Definition 3.1.** A \((k,n)\)-Asmooth-Bloom sequence is a sequence of pairwise co-prime integers \( a_0, a_1 < a_2 < \ldots < a_n \) such that:

\[
a_0a_{n-k+2}a_{n-k+3} \ldots a_n < a_1a_2\cdots a_k
\]

The secret \( S \) is now assumed to be an integer such that \( 0 \leq S < a_0 \). To generate the shares, the dealer first chooses a random number \( \gamma \) such that:

\[
S + \gamma a_0 < a_1a_2\cdots a_k.
\]

Then, the shares are defined by:

\[
S_i = S + \gamma a_0 \mod a_i \quad \text{for } i = 1, \ldots, n.
\]
To generate an example with $k = 3$ and $n = 5$, we will use the following integers:

$$a_0 = 1297, \quad a_1 = 1571, \quad a_2 = 1601, \quad a_3 = 1721, \quad a_4 = 1759, \quad a_5 = 1783.$$  

We can check that this is a $(3,5)$-Asmooth-Boom sequence in the following way:

```python
sage: k, n = 3, 5
sage: a0 = 1297
sage: aseq = [1571, 1601, 1721, 1759, 1783]
sage: prod_smallest = prod(aseq[:k])
sage: prod_largest = prod(aseq[:n-k])
sage: print(a0 * prod_largest, prod_smallest)
3262176787 4328609291
```

Suppose that the secret to be shared is $S = 910$. We choose $\gamma = 156397$, and verify that it satisfies condition (5):

```python
sage: S = 910
sage: gamma = 156397
sage: x = S + gamma * a0
sage: print(x, prod_smallest)
202847819 4328609291
```

We are now ready to generate the shares:

```python
sage: shares = [x % a for a in aseq]
sage: shares
[299, 1119, 433, 1698, 1258]
```

Let’s now consider secret recovery. Suppose that $k$ of the entities, $i_1, i_2, \ldots, i_k$ agree to access the secret. The first step is exactly the same as in Mignotte’s scheme. They reveal their shares, and solve the CRT system:

$$
\begin{align*}
x &\equiv S_{i_1} \pmod{a_{i_1}} \\
x &\equiv S_{i_2} \pmod{a_{i_2}} \\
&\vdots \\
x &\equiv S_{i_k} \pmod{a_{i_k}}
\end{align*}
$$

with $0 \leq x < a_{i_1} a_{i_2} \cdots a_{i_k}$. Then, $x \equiv S + \gamma a_0 \pmod{a_{i_1} a_{i_2} \cdots a_{i_k}}$, and the restrictions on the range of $x$ and $S + \gamma a_0$ guarantee that $x = S + \gamma a_0$. Then, the secret $S$ is recovered by computing $x \mod a_0$.

Going back to the example, let’s suppose again that entities 2, 3 and 5 want to access the secret. This is how they would do it in Sage:
Thus the secret is recovered. Suppose now that two of the entities, say 2 and 4, attempt to recover the secret. They can solve the corresponding CRT system:

```python
sage: ilist = [1, 2, 4]
sage: blist = [shares[i] for i in ilist]
sage: mlist = [aseq[i] for i in ilist]
sage: x = CRT_list(blist, mlist)
sage: print x
sage: print x % a0
202847819
910
```

So they can't directly recover the secret. However, our "cheaters" are more resourceful. They know that the value they computed, $x$, is congruent to $S + \gamma a_0$ modulo $a_2a_4 = 2816159$. They could, then, attempt to compute $(x + ja_2a_4) \mod a_0$ for successive values of $j$, and hope to eventually hit on the secret. In our example, the number of tries to be made is $a_1a_2a_3 \div a_2a_4 = 1537$. So, the cheaters can compute:

```python
sage: guesses = set([])
sage: for y in range(x, prod_smallest, mm):
...     guesses.add(y % a0)
...
sage: print guesses
set([0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
... 1291, 1292, 1293, 1294, 1295, 1296])
```

The output above was truncated, but if you run the example, you will see that all integers from 0 to $a_0 - 1$ are represented! We can verify this computationally in Sage, too:

```python
sage: guesses == set(range(a0))
9
That is, according to the cheater’s computations, the secret could have any value whatsoever in the interval where it was initially selected.

This seems to indicate that the Asmooth-Bloom scheme does not give any information about the secret to parites of \( k - 1 \) or fewer of the entities. This, however, is not quite true. To understand why, we consider the frequencies of each of the possible guesses for the secret. Ideally, this should be a uniform distribution in the range \( 0, 1, \ldots a_0 - 1 \), but this is not the case. For the example above, however, each possible integer value in the interval \([1, 1296]\) appears either one or two times as a possible guess, so the distribution is not uniform.