Crypto

Table of contents

1. XLattice's Crypto Facilities

What follows is a summary description; it will be improved on as the crypto API stabilizes. Much more information is available from the Javadoc descriptions, but probably the most useful documentation is the suite of tests. As with all XLattice software components, unit tests are to be found in a parallel directory structure. In this case
`xlattice/crytpo/src/java/org/xlattice/crypto`
contains the software being tested and
`xlattice/crypto/src/test/org/xlattice/crypto`
and subdirectories the corresponding unit tests. The software is compiled and tested by running

```
./build.sh test
```

For the near future much of XLattice's `crypto` component will be just a thin wrapper around Java 1.4's crypto functionality as documented in

- Java Security Architecture
- Java Cryptography Architecture and
- Java Cryptography Extension

It is likely that much of this functionality will be replaced by equivalent XLattice code, but this will probably have to wait until XLattice has servers outside of the United States.

1.1. SHA-1 Hash Generation

The Secure Hash Algorithm (SHA) maps a byte array of arbitrary size into a small fixed-size byte array, a hash of the input. For SHA-1 the hash is 160 bits (20 bytes) long.

In XLattice terminology, the 20-byte SHA1 hash of a data file's contents is referred to as a **content hash** or **content key.** Because the chance of a collision is so incredibly small (approximately 0.7E-48), the content key is for all practical purposes unique to a given document and therefore can be used as a unique identifier.

1.2. Bloom Filters using SHA1 Keys

The principal application of XLattice's Bloom filters is in cheaply determining whether an item with a given data key is on another node. Each such item (file) has a 20-byte key. This is most often an SHA1 hash of the file's contents. Rather than sending one another lists of keys, nodes can exchange Bloom filters which can be used to test whether a given file is present. The test never gives a false negative, but sometimes may give a false positive: it indicates that a key is present when it is not. The false positive rate decreases with increasing filtersize.

A Bloom filter uses a set of k hash functions to determine set membership. Each hash

function produces a value in the range 0..M-1. The filter is of size M bits. To add a member (the 20-byte key) to the set, apply each function to the new member and set the corresponding bit in the filter. For M very large relative to k, this will normally set k bits in the filter; that is, it is very improbable that two or more functions will set the same bit.

To check whether x is a member of the set, apply each of the k hash functions to x and check whether the corresponding bits are set in the filter. If any are not set, x is definitely not a member. If all are set, x may be a member. The probability of error (the false positive rate) is f = (1 - e^(-kN/M))^k, where N is the number of set members.

The XLattice class takes advantage of the fact that SHA1 digests are good- quality pseudo-random numbers. The k hash functions are the values of distinct sets of bits taken from the 20-byte SHA1 hash. The number of bits in the filter, M, is constrained to be a power of 2; M == 2^m. The number of bits in each hash function may not exceed floor(m/k).

There are actually two XLattice Bloom filter classes. `BloomSHA1` is a conventional filter and the type best used for summarizing contents to neighboring nodes. `CountingBloom` is a counting Bloom filter, one in which a four-bit counter is associated with each filter bit. With very high confidence this can be used to both add and subtract members. Typically a node will keep a counting Bloom filter to track its contents and then compress the filter to the non-counting form for copying to neighbors. This is much cheaper than generating a new `BloomSHA1` on each neighbor request.

*The Bloom filters are *not* part of Sun's Java crypto functionality.*

1.3. RSA Keys and Pair Generation

1.4. Digital Signatures

The only digital signature algorithm currently supported by XLattice is SHA1withRSA.

1.5. SignedList

A `SignedList` is typically a dated and digitally signed list of documents. The SignedList itself has a unique identifier based on SHA1. This **signed key** is not a content hash but a function of the public part of the RSA key used to sign the list and the title of the document.

The digital signature covers the entire document,

- the RSA public key, base64-encoded
- the title
- the timestamp in YY-MM-DD HH:MM:SS format
- and the content lines themselves

To make it easier for a human to read, the String version of the SignedList precedes and follows the content lines with delimiter lines.

Because the `SignedList` has a unique 20-byte identifier, it can be stored and retrieved in exactly the same way as other documents, using its signed hash as a key.

*The `SignedList` and its subclasses are *not* part of the Sun Java libraries.*