Practical Data Protection in 2011

Alistair Crooks
agc@NetBSD.org

Abstract

This paper looks at various methods of protecting data from corruption at rest or during transmission. Some of the methods are old, but have been given a new slant by new code. Other methods showcase new codes and new techniques which have, until now, not been present or used in any BSD variant. Four different types of protection are presented: detection of changes to data, erasure coding, encryption, and dispersal of information redundantly, and various libraries and utilities are presented to show how data can be protected when faced with challenges in each of these categories. Legacy solutions are examined, and their benefits and drawbacks listed. Threshold scehemes for data protection are presented, providing a more flexible and scalable approach than existing RAID solutions, for example, whilst preserving data from being exposed to snooping methods. Erasure coding methods are presented, as are one-time shared key schemes, in which a threshold number of users can decrypt a secret; knowledge of that secret does not give any other privilege. The examination concludes by presenting practical methods - libraries and programs - of protecting our data.

1. Introduction

This paper looks at ways that we have for detecting changes to data, either at rest, or in transit. Four main categories of data protection are defined: modification detection, error correction, data privacy (protection of data from unauthorised viewing), and finally redundancy of data - resilience and duplication, through shares and threshold schemes. Traditionally we have used checksums and digests to perform this function. With the advent of netpgp, BSD systems can now use a BSD-licensed library to use digital signatures to discover data modification. Moving on from detection of modifications to data, we also look at the methods available for correcting errors, through forward erasure codes and similar. Our next category of protection is that of prevention of unauthorised
viewing of data. Finally, some of the methods used to protect data are put together, analysed and described, in order to provide functionality which can prove useful in a number of applications.

New functionality has been added to the NetBSD repository over the past year; this functionality is in the form of libraries, along with small programs sitting as a thin layer on top of each library to provide command line access to the libraries. In addition, other language bindings can be used to access each of the libraries, in the same way that there are perl, lua, python, and tcl bindings for libnetpgp.

The libraries are used as the basic building blocks of further solutions, and there are many, many further uses for these pieces of software. This paper aims to inspire the creation of many further pieces of software, and to protect our data from unauthorised modification, either planned or unplanned.

There is also a hope that we can review, and perhaps re-think, the way that we provide some protection to our data today. For instance, if an IP packet fails its checksum calculation, we silently discard that packet. At the same time as we ask for re-transmission of that packet in TCP, we kill the TCP windows which are used to buffer current communications, and also kill any SACK in progress. If we could use forward erasure codes to detect data modification, it may be possible to keep connections
running whereas before they would have been dropped.

We could also attempt to recover from storage errors on a disk, recovering data rather than considering a block as "gone bad". Obviously the damage that a bad block can do to a file system's operations is a function of whether the data which has gone bad is data or metadata; what cannot be denied is that all data is precious, and should not be discarded lightly.

2. Modification Detection

Traditionally, checksums, cyclic redundancy checks and digests have been used to verify that data has not been modified since the original digest or checksum was calculated. More recently, digital signatures hav been added using libnetpgp.

The libraries and utilities added to NetBSD's othersrc have expanded the functionality available for this.

2.1 CRC

libcrc(3) is a simple crc calculation library, with a corresponding program called crc(1). Its interface has been modelled on the MD5(1) and SHA1(1) set of programs. Different CRC polynomials can be specified on the command line, allowing different CRC calculations to be made.

The arguments to select polynomial and other parameters are set using the CRC_Args() function:

CRC_Args(CRC_CTX *model, int width, unsigned poly, int reverse)

where width is the crc value masked with is (2^width)-1, poly is the polynomial of the CRC, and reverse is non-zero if the returned value is to have its bits "reflected" in 32 bits.

Since the domain of the polynomials is that of 32-bit unsigned integers, any of the standard polynomials can be used. A good list of those polynomials is given in [WikipediaCRC2011]

This allows command-line or library/scripting level access to CRC calculations, such as those done in the IP header, and those in SCTP/iSCSI (which use different polynomials).

```
% crc Makefile ../dist/crctable.expected ../Makefile
CRC (Makefile) = b64267c5
CRC (../dist/crctable.expected) = 65abc317
CRC (../Makefile) = 674ae574
% crc < Makefile
b64267c5
%
```

Whilst calculating CRC values is usually done in optimised assembler deep in the kernel, having the ability to calculate these digests at the library level is useful, for such things as storage simulators, network simulators, traffic generators, attack traffic generators, fuzz testing and hardware failure protection.

2.2 Hash64

Our hash functions produce digest values, and the length of the hash value is defined by the hash algorithm itself. Accordingly, we can get collisions over the hash values.

For some uses, this can be benign - we may not want a unique value from the hash function, or the set of inputs which are hashed may be constrained in a way that hash collisions may not occur.

For other uses, there may be more consequences - our unique values may not be quite as unique as previously presumed.

By concatenating the output of a number of hash functions, the chances of a hash collision are greatly reduced. The idea is not new - almost all of the BSD packaging systems use it with their "checksum" calculations - pkgsrc uses RMD160 and SHA1 (and the size), OpenBSD use MD5, RMD160, SHA1, SHA256 (and the size), although FreeBSD uses simply SHA256 (and the size). The other benefit of this is that if someone is able to perform a second pre-image attack on a hash method, concatenation of hash values will not be effective.

Hash64 was implemented as a way of bringing the concatenation technique to existing 32-bit hash calculations - we have seen a number of email threads about the "best hash algorithm to use", and lots of timing and coverage discussions have ensued. 64-bit integer values, from a coding perspective, are just as easy to calculate and use as 32-bit integers these days.

This implementation of hash64 takes the known hashing algorithms, and implements them in the same function signatures as sha1, md5, rmd160 etc. The output from these hashing functions is a 64-bit integer, containing the concatenated hashed values over the input by the designated algorithms. Standard context structures are used to keep context between invocations, and the standard *_File and *_Data functions are used to calculate the hash values without exposing any contexts.

```
HASH64_CTX ctx;
uint64_t hash;
char *buf;
int n;

buf = "The quick brown fox jumps over the lazy dog";
n = strlen(buf);
HASH64_Init(&ctx, "perlhash,mousehash");
HASH64_Update(&ctx, (uint8_t *)buf, n);
HASH64_Final(&hash, &ctx);

 /* Print the digest as one long hex value */
 printf("%016" PRIx64 "\n", hash);
```

Alternately, the helper functions could be used in the following way:

```
HASH64_CTX ctx;
uint64_t hash;
const char *buf = "The quick brown fox jumps over the lazy dog";

printf("%016" PRIx64 "\n", HASH64_Data(buf, strlen(buf), &hash, "perl,mouse"));
```

This implementation uses 32-bit checksums. One extension to this is to apply this idea to our existing digest functions, to create combined SHA1/SHA256, or, using different digest families of algorithms, SHA1/TIGER or WHIRLPOOL/RMD160. Using these methods, hash values are much more resistant to second pre-image attack than a single hash value, which may be considered weak, or "broken".

2.3 HMAC

HMACs are digest values, and incorporate a digest of a shared key as part of the digest calculations. They were invented in 1996. With HMACs, there are two digest calculations taking place, the inner and the outer digest calculation. The digest is broken up into blocks of a fixed size, and the hmac value is calculated on a concatenation of the outer key, and the hash value of the inner key concatenated with the message. The following pseudocode illustrates

```
const char *
hmac(const char *key, const uint8_t *message, size_t blocksize,
      const char *(*hash)(const uint8_t *))
{
      if (strlen(key) > blocksize)
            key = (*hash)(key);
      (void) memset(outerkey, 0x5c, blocksize);
      (void) memset(innerkey, 0x36, blocksize);
      (void) memcpy(outerkey, key, strlen(key) - 1);
      (void) memcpy(innerkey, key, strlen(key) - 1);
      return (*hash*)(concat(outerkey, (*hash)(concat(innerkey, message))));
}
```

HMACs are not new to any BSD operating system - openssl includes implementations of them, as do other sources - but this implementation was written to provide a small, self-contained, embeddable implementation of HMACs.

```
% hmac -k '  ' -a sha256 Makefile
hmac-sha256 (Makefile) =
9612652f4e2371dc41e75f7d1b80e305a7be54b171b6a6002ba5caa072f3e825
% hmac -a sha256 Makefile
hmac key:
hmac-sha256 (Makefile) =
9612652f4e2371dc41e75f7d1b80e305a7be54b171b6a6002ba5caa072f3e825
% hmac -a sha256 < Makefile
hmac key:
9612652f4e2371dc41e75f7d1b80e305a7be54b171b6a6002ba5caa072f3e825
%
```

The size of the output hash function is obviously the same as the digest function that is used. HMACs are defined in RFC 2104.

The cryptographic strength of the HMAC is actually dependent upon the key, and not the underlying digest algorithm. [WikipediaHMAC2011] In addition, although the underlying digest algorithms may be vulnerable/rendered ineffective by collisions, HMACs are much more resistant to such events. Therefore, HMAC-MD5, for example, does not suffer from the same weaknesses that have been found in MD5.

This means that, in normal use, we could use an HMAC with a shared key as a more collision-resistant digest value.

2.4 Merkle trees, Tiger trees and Hashtrees

Traditional hash or digest values are able to tell us that data was modified. They are not able to tell us which part of the data was modified. To do that, Merkle trees are used.

By splitting a file, or data, into a number of fixed size blocks, we can calculate a hash value on that block, using standard digest functions.

Arranging those hash values as the bottom layer of a tree, and calculating a hash value over each of the hash values in that layer produces another hash value. This process can be continued until one hash value exists at the root of a tree.

The implementation of hash trees described here takes various values as input, in order to create the output. We are also able to output various layers of the hashtree, which allows us to pinpoint the source of data modification in a more defined way.   The hashtree(1) utility has a command line switch for the blocksize (-b blocksize), or alternatively, a command line switch for the number of block calculations to be made (-n blocks). Some command line examples are:

```
% hashtree -n 1 dist/hashtree.c
HASHTREE/sha1/1/10358/10358 (dist/hashtree.c) =
253f185d39783fd29ca6a7936fee4b50e3775f39
% sha1 dist/hashtree.c
SHA1 (dist/hashtree.c) = 253f185d39783fd29ca6a7936fee4b50e3775f39
% hashtree dist/hashtree.c
HASHTREE/sha1/2/10358/1024 (dist/hashtree.c) =
6a765d5809534fa75fd2f4f2f88168e65cb1182f91ac14db77b0052ae083743c10ed0ccea98ebf44a945eb
72ee0303a8b5b89a2a590ec2e684418942bae3e0be34567bd12721167239fdf641c24ec51b59426924a327
60adf11370aca375d60a97364613f93e1e44f848a3557c141b7422aa7e63fef99ff4e056124513c228c9a3
3e0dff08e6baa2903f959f428dd60f9a6390b0d4108d0826713d33ab6e5520fe20e61ac700659eb5e6a1cf
5653da05d65c0d792cd3affaadd037372cfbf7085e14caea50dd89e2790cd982824fec985e2a97fcbbc415
e91c27dc4f3089ec446784262269677aa1736fba2e5a768114
% hashtree -b 4096 dist/hashtree.c
HASHTREE/sha1/2/10358/4096 (dist/hashtree.c) =
1a958db53705b1ef4ef4e7418f7aa6e25226c4468783684b4ff3d152a66a38e7ad3e5c8d88ad0851b5f916
e7d
015015c6e0ddf0cef4cf7af2d48a0ad6db45855c61ea5454b579221bcbeeb5e282e626d
%
```

As can be seen, the output includes the number of blocks, the total number of bytes which were iterated over, and the blocksize used in the calculation. The key is not displayed for obvious reasons. The output displays the concatenation of all the hash values in the output.

One example of hashtree usage will be shown in the next section.

2.5 Lamport signatures

Lamport signatures were invented in 1979, and are used as one-time signatures (since verification exposes parts of the public key, which is derived from the secret key). Lamport signatures can be built from any cryptographically-secure hash function. It is posited that Lamport signatures are still secure, depending on the hash function used, in the case of quantum computers; the same cannot be said of RSA or DSA, for example.

In combination with hashtrees or merkle trees, Lamport signatures can be generalised to work by exposing only the top element of the hashtree, and by providing the elements of the hash tree that make up the hash values of the other blocks. Whilst this exposes one hash value (out of 256 in the case of SHA256), the other hash values making up the key are not exposed, and the key can be used again.

- to create a secret Lamport key, a random number generator is used to generate 256 pairs of random numbers, each number being 32 bytes in length. Each of these pairs has a "0" value and a "1" value. A secret key is thus 16KB. The public key is derived from the secret key by hashing each of the 512 32-byte random numbers to give 512 32-byte numbers, also 16 KB.

- to create a lamport signature, the message itself is hashed to give a 256-bit digest value. For each of the bits in this hash value, the corresponding hash from the private key is used. If this bit is a 0, the "0" value from the 256 pairs is used; if the bit is 1, the "1" value from the 256 pairs is used. In this way, 256 numbers of 256 bits each constitute the signature, making it 8KB in length. The unused values are discarded, and the key cannot be used again, since the private key is exposed during verification.

- to verify a signature, the same message is hashed to give a 256-bit digest value, and the corresponding bits in the public key are used to generate an 8KB computed signature. The hashed signature is calculated by hashing each of the 256 32-byte numbers in the signature itself. The resulting 8KB signature must match the calculated signature for a match to be made.

This implementation provides the simple, one-time key, and uses the SHA256 algorithm as the hash function.

3. Correction of Errors

3.1 Reed-Solomon Erasure Coding

The librs library implements Reed-Solomon encoding, to enable errors and erasures to be corrected, thereby protecting data in transit.

In the literature, an error is a character which has been modified, whilst an erasure is a character which has been modified, and whose location within the data is known. Due to its most common usage, erasures are not common, and errors are the most common use for the parity bytes. It takes one parity byte to correct an erasure, and two parity bytes to correct an error. librs is used to add ``parity'' bytes, which can be used to detect erasures in transmission of data, or of corruption of data at rest. As well as detecting erasures and errors in the data, the parity bytes can be used to reconstruct the original data, provided that enough data has been transmitted.

The total size of a buffer used internally in librs is 256 bytes, and that must also include the parity byte count. The default number of parity bytes is 32 bytes, which will allow the correction of up to 16 errors. If a buffer larger than 256 bytes is given to the rs_encode() function, the output will be split up into slices of 256 bytes, and the final buffer may contain less than 256 bytes.

An example of the rs(1) command is:

```
% cp Makefile a
% rs -o a.rs a
% sed -e 's|m|M|g' a.rs > a.rs2
% rs -d -o a.2 a.rs2
% ls -al a*
-rw-r--r--  1 agc  agc  588 Mar  8 07:42 a
-rw-r--r--  1 agc  agc  588 Mar  8 07:43 a.2
-rw-r--r--  1 agc  agc  684 Mar  8 07:42 a.rs
-rw-r--r--  1 agc  agc  684 Mar  8 07:43 a.rs2
% cmp a a.2
%
```

The output size from the rs_encode() function depends on the number of parity bytes, and the number of data bytes used. In addition, there is an overhead of 12 bytes to hold the encoding information. For example, a 28,24 encoding will use 4 parity bytes for every 24 data bytes. This means that the output will be at least (((datasize / 6) + 1) * 7) + 12 bytes long.

As we shall see in th enext section, Reed-Solomon encoding is an inherent part of CD and CD-ROM encoding.

A later form of erasure coding called Turbocodes were invented in 1993. Turbocodes are used extensively in 3G and 4G mobile phones.

Low-density parity-check (LDPC) codes were invented in 1960s by Robert Gallager, and then re-invented in 1990s by David Mackay and Radford Neal in 1990s. [Neal1996] claims the best performance for erasure codes are those of LDPCs:

> *"The decoding algorithm for LDPC codes is related to that used for Turbo codes, and to probabilistic inference methods used in other fields. Variations on LDPC and Turbo codes are currently the best practical codes known, in terms of their ability to transmit data at rates approaching channel capacity with very low error probability."*

The pkgsrc entry for ldpc can be found in pkgsrc/devel/ldpc.

Tornado codes are a class of erasure codes, and require more space than Reed-Solomon erasure codes, but are much faster to generate and can fix erasures faster. Software-based implementations of tornado codes are about 100 times faster on small lengths and about 10,000 times faster on larger lengths than Reed-Solomon erasure codes.

The drawback is that Turbocodes are patented in the USA.

Variations of Tornado codes include Raptor codes, and others.

3.2 Circa

In 1977, CDs were developed at Philips. Every CD's encoding includes a form of protection against media "going bad" - the quality of the data being depleted such that the disk cannot be read effectively. This was to ensure that media errors (measured at 1 in a million bits), when taken together over a whole CD, could not interfere with the final sound [ECMA1996]. To do that, various different steps are taken to ensure that data from a sector at the user level (currently 2348 bytes by default in a user-level sector) are converted into 3136 bytes (98 frames of 32 bytes) in an encoded CD sector.

The input data sector is split up into C3 frames of 24 bytes each.

The transformations are, in encoding order:

- **delay1** Every other 4 bytes in the input stream are put into a delay slot of 1 frame.
- **scramble** Within a C3 frame, the bytes are changed to occupy different positions. This is done to mitigate the problem where a number of bytes next to each other are the subject of errors.
- **Q-Parity** A Reed Solomon 28,24 erasure code is calculated over the 24 bytes of the C3 frame. This erasure code is inserted at the half-way point of the C3 frame. This creates a C2 frame of 28 bytes.
- **Dispersal** Each of the bytes in a C2 frame is striped across the sector, according to the byte offset from the start of the C2 frame. Byte b in C2 frame f will end up in frame (f + b) and byte b in the output sector.
- **P-Parity** A Reed Solomon 32,28 erasure code is calculated on the C2 frame, to create a 32-byte C1 frame.
- **delay2** Even bytes are delayed 1 frame in the output.

The output from these steps is a sector of output data, made up of 32 byte C1 frames. When decoding the circa data, the original C3 frames are reconstructed. A worked example follows:

```
% circa Makefile > Makefile.circa
% hd Makefile.circa | head -20
00000 | 00 a0 4c c1 40 0c 00 00 00 5a 31 74 63 31 00 fd  | ..L.@....Z1tc1..
00016 | 00 00 00 00 00 35 00 3f 00 00 00 63 7f 00 00 7d  | .....5.?...c...}
00032 | 00 00 00 00 ac 5c e4 9c 37 00 00 0a 61 49 00 00  | .......7...aI..
00048 | 42 00 00 00 7e 32 99 00 63 00 00 64 00 00 00 00  | B...~2..c..d....
00064 | 55 00 00 00 fd 1d 05 71 00 20 9f 24 20 17 31 00  | U......q. .$ .1.
00080 | 2f fd 00 ff 52 00 16 00 63 00 00 52 00 00 00 00  | /...R...c..R....
00096 | 69 4d 00 00 c3 9a 12 cc 97 20 70 2e 28 c1 61 2e  | iM....... p.(.a.
00112 | 24 53 60 66 af 00 7d 00 20 00 00 00 69 00 00 00  | $S`f..}. ...i...
00128 | 69 36 00 00 af 71 b1 84 76 63 57 55 90 6b 69 67  | i6...q..vcWU.kig
```

```
00144 | 49 30 ff 3a a3 00 ff 00 20 20 00 00 34 00 00 00  | I0.:....  ..4...
00160 | 7d 42 00 00 22 18 74 74 36 75 64 74 00 00 c0 72  | }B..".tt6udt...r
00176 | 73 0a 65 2b ea 00 c2 00 41 20 00 00 3d 00 00 00  | s.e+....A ..=...
00192 | 00 72 00 00 4e 20 0f 72 62 09 43 00 00 00 00 65  | .r..N .rb.C....e
00208 | 52 54 35 0a 64 c2 0c 00 00 20 00 00 2e 00 00 00  | RT5.d.... ......
00224 | 00 72 00 00 2c 1e 13 5f 6c 24 20 00 00 00 00 00  | .r..,.._l$ .....
00240 | 40 75 52 3e 22 e9 f1 3c 00 61 00 00 0a 6c 00 00  | @uR>"..<.a...l..
00256 | 00 2f 65 00 38 b9 86 3b 0a 00 00 00 00 00 00 00  | ./e.8..;........
00272 | c0 44 0a 63 00 54 63 e2 00 3c 00 00 61 31 00 00  | .D.c.Tc..<..a1..
00288 | 00 00 3a 00 09 69 9d 52 20 00 00 00 00 00 00 00  | ..:..i.R .......
00304 | 00 00 6b 00 00 41 ad 27 00 24 00 00 00 20 00 00  | ..k..A.'.$... ..
% sed -e 's|M|N|g' Makefile.circa > Makefile.damaged
% cmp Makefile.circa Makefile.damaged
Makefile.circa Makefile.damaged differ: char 98, line 2
% circa -d -o Makefile.recon Makefile.damaged
% diff Makefile Makefile.recon
%
```

Our implementation ensures that no delay lines cross a sector border - this is to ensure that data from a single sector are kept in a single sector, for use in communications, for example, or for block storage.

4. Protection of Data from Unauthorised Viewing

4.1 Encryption, Decryption

To protect data from being viewed at rest or in flight, we need to encrypt the data. The use of netpgp [Crooks2009] to encrypt data in a BSD-licensed library, and using OpenPGP format (RFC 4880) keys. Whilst netpgp is not new, it is used by some of the tools and utilities described later on.

4.2 Signing and Verification

Once again, netpgp is used to sign and verify the provenance of files, and to make sure that data has not been modified since it was signed. In addition to netpgp, Lamport signatures can be used to sign files.

Netpgp can source its keys from OpenPGP format keys, ssh keys (both user and host keys), and from openssl certificates now (using the openssl2ssh utility). A similar utility has also been written to construct an openssl-compatible certificate from an ssh key, but due to the information required for an ssh key, and that required in an openssl certificate, a lot of the information has to be synthesised when moving in this direction, such as expiry, validity date, CA etc.

4.3 Related Work

This area of research is going to be more popular in years to come, with the advent of mass-scale cloud computing. We have seen efforts such as homomorphic cryptography, the ability to work on encrypted data without decrypting it, take small steps towards the mainstream. [Gentry2009]

5. Protection - Redundancy, Shares and Thresholds

Redundancy and resilience have been used in the past in the context of RAID storage, either in hardware or software, to make sure that the original data can be recovered even in the face of component failure in the storage subsystem. This is useful and necessary, but usually means that the available space has to be totally dedicated to the RAID storage.

ZFS changes this by allowing metadata to be made more resilient than the data.

Threshold schemes have been around since the 1970s, and are concerned with the reconstruction of the original data from a constituent number of shares of this data.

The number of shares with which the original data can be reconstituted is set at share creation time.

There are two methods we shall look at here, Shamir's Secret Sharing Scheme (SSSS), and Rabin's Information Dispersal Algorithm (IDA).

5.1 SSSS

Shamir's Secret Sharing Scheme (SSSS) is a cryptographically-secure threshold scheme, see [Shamir1979]. It uses byte-wide Galois Field arithmetic (GF8) and cryptographically-secure random number generation to encode the input data, producing shares as output which are all the same size as the input data, but have the attribute that a casual (or even sustained) glance at the data will not leak any information whatsoever.

```
 % cp /etc/group a
% ssss -t 3/5 a
% ls -al a*
-rw-r--r--  1 agc  agc  526 Sep 25 04:21 a
-rw-r--r--  1 agc  agc  542 Sep 25 04:21 a.000
-rw-r--r--  1 agc  agc  542 Sep 25 04:21 a.001
-rw-r--r--  1 agc  agc  542 Sep 25 04:21 a.002
-rw-r--r--  1 agc  agc  542 Sep 25 04:21 a.003
```

```
-rw-r--r--  1 agc  agc  542 Sep 25 04:21 a.004
% hd a.000
00000 | 73 34 00 00 05 03 01 00 0e 02 00 00 00 00 00 00  | s4..............
00016 | a9 76 b6 3d 68 02 22 9b c0 c0 15 ed 8a 1d cc 96  | .v.=h.".........
00032 | cc 9f 0b b9 2a 56 54 ae 83 6b e4 bb 64 85 c0 24  | ....*VT..k..d..$
00048 | 05 7d 9d 8a 8d e0 9a 2b 0b 36 3a 18 c8 9d 59 4c  | .}.....+.6:...YL
00064 | 27 21 af 4a 86 2e 9e 94 23 0e ea 26 67 f3 50 ef  | '!.J....#..&g.P.
00080 | 8c 6f a0 d2 c9 5c 32 98 59 7c 91 2c 1d e6 ca 55  | .o...\2.Y|.,...U
00096 | e2 ae 62 df 34 64 33 c8 1d da 53 a1 73 e4 97 20  | ..b.4d3...S.s..
00112 | 9b db 5c 5b d4 64 b7 3a 41 66 ec 44 04 3a 7a 1d  | ..\[.d.:Af.D.:z.
00128 | 7d a1 28 05 95 4e 76 44 28 3d 7a bf 03 bd aa 94  | }.(..NvD(=z.....
00144 | 76 63 c4 36 8d 77 2a 95 77 c5 23 09 e1 f1 15 42  | vc.6.w*.w.#....B
00160 | 28 10 d5 fb da 1c 58 0a d6 8e a3 17 47 a6 bb 8f  | (.....X.....G...
00176 | 4d d9 20 df d0 5d 02 0f f7 96 90 00 d5 44 15 51  | M. ..]......D.Q
00192 | a0 20 dd 97 dc 73 92 6f 4c 26 cf b7 e5 ad 4b 67  | . ...s.oL&....Kg
00208 | 61 7b 3d db 4f 6f 58 c4 62 d3 50 63 84 41 57 c1  | a{=.OoX.b.Pc.AW.
00224 | 80 3c 86 a2 57 5e 47 60 a6 6c b9 14 5c a1 c5 8f  | .<..W^G`.l..\...
00240 | 45 47 74 d6 0b ef cc c8 83 42 8d 6b 7a d7 94 91  | EGt......B.kz...
00256 | a9 85 8d 42 2e 04 79 68 51 1f 20 d7 3a b4 4a 78  | ...B..yhQ. .:.Jx
00272 | c4 eb 28 7d cb 97 b8 d1 b4 1f 52 10 14 46 4a a9  | ..(}......R..FJ.
00288 | ed 75 38 85 8e d8 99 cd 78 c6 a8 e3 fc 69 5c e8  | .u8.....x....i\.
00304 | ad db 91 9f 1f 30 f2 33 be 6e 78 9a 16 87 b4 f1  | .....0.3.nx.....
00320 | eb ea 38 1b b8 7f 80 17 8f 5d df eb d1 40 49 af  | ..8......]...@I.
00336 | a0 66 bb 18 8f f1 5f a9 99 23 21 80 71 26 2b bf  | .f...._..#!.q&+.
00352 | ce c0 b6 32 78 95 5b ee 93 3f 72 01 fa 8f 2d d9  | ...2x.[..?r...-.
00368 | 25 38 0b 66 0e 06 b8 8b 69 54 03 78 7d 99 8e a6  | %8.f....iT.x}...
00384 | f1 b6 39 92 c7 6a 0a cf 75 f2 57 0f ac c2 a5 d0  | ..9..j..u.W.....
00400 | bd 53 25 6b c9 0c 12 55 3f b1 54 c0 3c 5d 05 a1  | .S%k...U?.T.<]..
00416 | 35 c9 61 66 df b4 5f 92 c4 92 57 09 a1 a1 df 21  | 5.af.._...W....!
00432 | b9 cc 7a 8a 0b ab e3 2d 10 80 fe eb 42 7a 03 56  | ..z....-....Bz.V
00448 | 01 23 7e fc 20 3d 72 48 d3 d5 21 a1 e1 83 40 0f  | .#~. =rH..!...@.
00464 | 7c c5 5e 83 17 44 62 31 a1 3b 72 ba 6a 45 42 e1  | |.^..Db1.;r.jEB.
00480 | 5e 3b c6 47 92 50 13 da 4b f7 47 c9 03 8a d3 7a  | ^;.G.P..K.G....z
00496 | 91 4b 35 7f b5 3e 1a 20 58 0b 8d d7 86 84 ba a6  | .K5..>. X.......
00512 | c1 ba be aa 93 2a c2 ef 08 4c 49 da fa e2 6c bc  | .....*...LI...l.
00528 | c4 30 ae 10 bd 6d e1 a2 ba df 06 46 54 79        | .0...m.....FTy
% ssss -j -o regroup a.003 a.002 a.000
% diff a regroup
%
```

5.2 Threshold Schemes, Rabin's Information Dispersal Algorithm

Rabin's Information Dispersal Algorithm has similar functionality, but different properties to SSSS. The IDA will create shares which are

(threshold shares) / (total shares) * (original data)

size in length. However, IDA suffers from the property that the data is not encrypted in any way - if information obscurity is desired, then SSSS should be used, or the data encrypted prior to passing through the IDA.

```
% cp /etc/group origfile
% threshold -t 3/10 origfile
% ls -al /etc/group origfile.split*
-rw-r--r--  1 root  wheel  526 Jun 26 01:34 /etc/group
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 origfile.split0
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 origfile.split1
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 origfile.split2
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 origfile.split3
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 origfile.split4
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 origfile.split5
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 origfile.split6
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 origfile.split7
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 origfile.split8
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 origfile.split9
% threshold -j -o origfile.recons origfile.split4 origfile.split2 origfile.split3
% diff origfile origfile.recons
% rm origfile.*
%
```

The following example uses shell redirection to split the data on standard input:

```
% threshold -t 3/10 < origfile
% ls -al threshold.split*
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 threshold.split0
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 threshold.split1
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 threshold.split2
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 threshold.split3
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 threshold.split4
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 threshold.split5
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 threshold.split6
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 threshold.split7
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 threshold.split8
-rw-r--r--  1 agc  agc  192 Sep 17 13:59 threshold.split9
% threshold -j -o origfile.mem threshold.split1 threshold.split2 threshold.split3
% diff origfile origfile.mem
% rm threshold.*
```

The following example prompts for input using getpass(3)

```
% threshold -t 3/10 -i
Data to share:
% ls -al threshold.split*
-rw-r--r--  1 agc  agc  20 Sep 17 13:59 threshold.split0
-rw-r--r--  1 agc  agc  20 Sep 17 13:59 threshold.split1
-rw-r--r--  1 agc  agc  20 Sep 17 13:59 threshold.split2
-rw-r--r--  1 agc  agc  20 Sep 17 13:59 threshold.split3
-rw-r--r--  1 agc  agc  20 Sep 17 13:59 threshold.split4
-rw-r--r--  1 agc  agc  20 Sep 17 13:59 threshold.split5
-rw-r--r--  1 agc  agc  20 Sep 17 13:59 threshold.split6
-rw-r--r--  1 agc  agc  20 Sep 17 13:59 threshold.split7
-rw-r--r--  1 agc  agc  20 Sep 17 13:59 threshold.split8
-rw-r--r--  1 agc  agc  20 Sep 17 13:59 threshold.split9
% threshold -j threshold.split4 threshold.split2 threshold.split3
hello world
%
```

Extreme cases of resilience through shares are the various parity forms of RAID protection, such as RAID4, RAID5 and RAID6. The threshold(1) utility also includes RAID4 and RAID5 functionality.

6. Using the elements

Firstly, an interlude to introduce mat, the minimalist archiving tool

6.1 Mat

mat is an archiver program (the "minimalist archiving tool" program), similar to tar or pax, but holding digests of the individual components in the archive (right now, these are sha256 digests). in addition, mat has been designed to have no arbitrary restrictions on sizes of meta-data fields, or of data itself. it holds linked files internally as efficiently as possible, and displays these linked files in a manner analogous to ls(1) for a symbolic link.

mat and mattar are a tar-like frontend for libmat(3). a pax-like frontend is called matpax(1).

There is a separate mat_verify() function exposed in the library, and it will perform sanity and integral validity checks on an archive that is presented to it. internally, every archive is validated before any listing or extraction can take place. no filenames which start with '/' are allowed, neither is any filename containing a "/../" string.

Because mat is intended to be used as part of a layered solution, mat does not include any compression, since it is usually much more efficient to construct the archive itself, and then to compress the whole archive, since there are "economies of scale" here.

Mat also includes no signing facility, but has been designed to be layered underneath such a signing program - by signing an inventory of a mat archive, with the sha256 digests in the inventory, it would be difficult to modify any part of the archive and still have a valid signature on the archive.

The mat(1) examples follow:

```
   % cp /etc/group group
   % ln -s Makefile m
   % mat cvf archive.mat group m /etc/group
   can't stat './etc/group'
   % mat tf archive.mat
   group
   m
   % mat tvf archive.mat
   -rw-r--r--  1 agc  agc    510 Jun 16 18:27 group
   lrwxr-xr-x  1 agc  agc      8 Jun 16 18:27 m -> Makefile
   % mat tvvf archive.mat

-rw-r--r--  1 agc  agc    510 Jun 16 18:27
d8008905d91a083115b2fb8bd469a36a94f07f9cafeb88c17a44add769d3c0a0 group

lrwxr-xr-x  1 agc  agc      8 Jun 16 18:27
7a6378e1ee1f9098b9fb5219e6bb2ae17df33230171ee5c2a56f0cc4335d5dfd m -> Makefile
   %
```

an example of multiply-linked files in a mat archive:

```
  % mat tvf a.mat
  drwxr-xr-x  2 agc  agc    512 Jun 16 18:27 a
  -rw-r--r--  4 agc  agc   1827 Jun 16 18:27 a/Makefile
  -rw-r--r--  4 agc  agc   1827 Jun 16 18:27 a/m1 == a/Makefile
  -rw-r--r--  4 agc  agc   1827 Jun 16 18:27 a/m2 == a/Makefile
  -rw-r--r--  4 agc  agc   1827 Jun 16 18:27 a/m3 == a/Makefile
  % mat tvvf a.mat
drwxr-xr-x  2 agc  agc    512 Jun 16 18:27
0000000000000000000000000000000000000000000000000000000000000000 a

-rw-r--r--  4 agc  agc   1827 Jun 16 18:27
13bc52eeb726edc6298ac9a4d1ca2f4ccc79a8b62412c5c4ab210ffbfefe1e35 a/Makefile

-rw-r--r--  4 agc  agc   1827 Jun 16 18:27
13bc52eeb726edc6298ac9a4d1ca2f4ccc79a8b62412c5c4ab210ffbfefe1e35 a/m1 == a/Makefile

-rw-r--r--  4 agc  agc   1827 Jun 16 18:27
13bc52eeb726edc6298ac9a4d1ca2f4ccc79a8b62412c5c4ab210ffbfefe1e35 a/m2 == a/Makefile

-rw-r--r--  4 agc  agc   1827 Jun 16 18:27
```

```
13bc52eeb726edc6298ac9a4d1ca2f4ccc79a8b62412c5c4ab210ffbfefe1e35 a/m3 == a/Makefile
  % mat tvvf a.mat '*/*3'

-rw-r--r--  4 agc  agc   1827 Jun 16 18:27
13bc52eeb726edc6298ac9a4d1ca2f4ccc79a8b62412c5c4ab210ffbfefe1e35 a/m3 == a/Makefile
  %
```

mat can also take its input from a template file, in the same format as NetBSD's build.sh METALOG file. The resulting mat archive uses the uid and gid provided in the template file. When extracting, using the -p switch, the "original" uid and gid are used:

```
  % cat dist/template
  ./d type=dir uname=root gname=wheel

 ./d/group type=file uname=root gname=wheel mode=0644 size=510
sha256=d8008905d91a083115b2fb8bd469a36a94f07f9cafeb88c17a44add769d3c0a0
  ./d/m type=link uname=root gname=wheel mode=0755 link=Makefile
  % mat cvf matty -T dist/template
  % mat tvvf matty

drwxr-xr-x  2 root  wheel    512 Jun 16 18:27
0000000000000000000000000000000000000000000000000000000000000000 ./d

-rw-r--r--  1 root  wheel    510 Jun 16 18:27
d8008905d91a083115b2fb8bd469a36a94f07f9cafeb88c17a44add769d3c0a0 ./d/group

lrwxr-xr-x  1 root  wheel      8 Jun 16 18:27
7a6378e1ee1f9098b9fb5219e6bb2ae17df33230171ee5c2a56f0cc4335d5dfd ./d/m -> Makefile
  % mkdir e
  % cd e && sudo mat xvf ../matty -p
  ./d
  ./d/group
  ./d/m
  % cd e && ls -ald d d/*
  drwxr-xr-x  2 root  wheel  512 Jun 16 18:27 d
  -rw-r--r--  1 root  wheel  510 Jun 16 18:27 d/group
  lrwxr-xr-x  1 root  wheel    8 Jun 16 18:27 d/m -> Makefile
  %
```

A substitution phase can be added via -s '|regexp|replace|' to the tar and pax frontends to mat(1) and libmat(3). This allows filenames to be manipulated on entry to or exit from an archive, so that entries can be created in the mat archive with generic filenames (such as "@extractiondir@" or similar), which can be manipulated when extracting archive components accordingly.

The substitution pattern uses the first character of the argument as the delimiter, which can itself be escaped using the '\\' character, should anyone be massochistic as to want to use the '/' character to delimit file and directory names in a substitution pattern. Extended regular expressions (not wildcards, fnmatch or glob patterns) are used to match file name patterns.

The same archive security and integirty checks are made for archives when extracting or listing, so it will not be possible to extract any archive which has been constructed with a component name starting with a '/', or which contains the "/../" pattern. Should such a pattern be wanted or needed, (and the big question there is "*why?*"), then a generic pattern can be substituted at mat archive creation time, and the extracting user will explicitly need to specify that pattern at extraction time. This is, however, really not advised, and a different way of creating the mat archive should be found.

6.2 Bag

For the next generation packaging system, we require an archiver which is accessible as a library (to enable scripting language and other language access), and which includes a digital signature ability, and state of the art compression.

A "bag" ia a way of encapsulating a mat archive with

- digital signature (to protect against data modification)
- xz compression (to minimise the amount of data transferred or stored)
- and circa data protection (to protect against data corruption)

With these protection methods in place, we then have a single indivisible atom of a packaging system, a binary package. We know who created the package and when, and whether we should trust it (due to our trust levels of the public key).

Our bag(1) utility uses mat as the underlying archiving utility.

```
% mat cvf test1.mat -C dist bag.c bag.h frontends.c main.c bag.1 bagpax.1 libbag.3
% bag -o test1.bag test1.mat
netpgp: warning - using pubkey from secring
signature  2048/RSA (Encrypt or Sign) 1b68dcfcc0596823 2004-01-12
Key fingerprint: d415 9deb 336d e4cc cdfa 00cd 1b68 dcfc c059 6823
uid              Alistair Crooks <agc@netbsd.org>
uid              Alistair Crooks <agc@pkgsrc.org>
uid              Alistair Crooks <agc@alistaircrooks.com>
uid              Alistair Crooks <alistair@hockley-crooks.com>
netpgp passphrase:
% bag -d -o test2.mat test1.bag
Good signature for /tmp/bag.09341a/inventory.gpg made Sun Jul 24 02:04:45 2011
using RSA (Encrypt or Sign) key 1b68dcfcc0596823
signature  2048/RSA (Encrypt or Sign) 1b68dcfcc0596823 2004-01-12
Key fingerprint: d415 9deb 336d e4cc cdfa 00cd 1b68 dcfc c059 6823
uid              Alistair Crooks <alistair@hockley-crooks.com>
uid              Alistair Crooks <agc@pkgsrc.org>
uid              Alistair Crooks <agc@netbsd.org>
uid              Alistair Crooks <agc@alistaircrooks.com>
```

```
   uid              Alistair Crooks (Yahoo!) <agcrooks@yahoo-inc.com>
   encryption 2048/RSA (Encrypt or Sign) 79deb61e488eee74 2004-01-12
   % cmp test1.mat test2.mat
   %
```

mat has the ability to verify the checksums of the constituent parts of the archive from the library level or the command line. If the digital signature does not verify properly, then the archive itself cannot be extracted. circa(1) protects against data modification in transit or at rest. mat(1) also has a mode to verify the embedded checksums in an archive against the same files, directories, symbolic and hard links found on the file system.

6.3 Shared key

There is sometimes a need to share a single piece of data amongst a group of people. Conventional (gnupg) practice for this is to allow the data to be decrypted by a number of users, any one of which may decrypt the data.

Sometimes, a group may not be as clearly defined as a number of users. At other times, we may want the original data to be recovered by a number of people working together, rather than by any single individual.

To effect this, the sharedkey(1) utility has been built, which generates a one-time RSA key, which may be passphrase-protected to protect against collusion (although the side effect of a central passphrase is to introduce a single point of failure to the process). The original data is encrypted to the new key, and then the secret key itself is shared, using ssss(1), to provide a threshold-based key which can be used to decrypt the encrypted data.

The threshold properties of the ssss scheme are up to the user to specify. Each individual share of the threshold-split key can be encrypted to its intended recipient's public key before transmission, to ensure that only the intended recipients may be able to reconstruct the original data, when a threshold of them is available.

```
 % sharedkey -t 2/3 /etc/group
 Generating a one-time key
 signature  2048/RSA (Encrypt or Sign) 6018e4d1edcd8801 2011-09-17
 Key fingerprint: e6ba 6252 120b 2f6a b702 4223 6018 e4d1 edcd 8801
 uid              /etc/group shared 2/3 by agc Sat Sep 17 14:32:28 2011
 netpgp: generated keys in directory /tmp/share.04718a/6018e4d1edcd8801
 Enter passphrase for 6018e4d1edcd8801:
 Repeat passphrase for 6018e4d1edcd8801:
 Shared secrets are in: /tmp/share.04718a

 % mat tvf /tmp/share.*/share000.mat
 drwx------  2 agc  wheel    512 Sep 17 14:32 share000
 lrwxr-xr-x  1 agc  wheel     16 Sep 17 14:32 share000/keyid -> 6018e4d1edcd8801
 -rw-r--r--  1 agc  wheel    623 Sep 17 14:32 share000/secret.gpg
```

```
-rw-r--r--  1 agc  wheel    617 Sep 17 14:32 share000/pubring.gpg
-rw-r--r--  1 agc  wheel   1323 Sep 17 14:32 share000/secring.gpg.share
% ls -laR /tmp/share.*
total 18
drwx------  2 agc   wheel   512 Sep 17 14:32 .
drwxrwxrwt  5 root  wheel   512 Sep 17 14:32 ..
-rw-r--r--  1 agc   wheel   617 Sep 17 14:32 pubring.gpg
-rw-------  1 agc   wheel  3444 Sep 17 14:32 share000.mat
-rw-------  1 agc   wheel  3444 Sep 17 14:32 share001.mat
-rw-------  1 agc   wheel  3444 Sep 17 14:32 share002.mat
%
```

Two of the shares can then be used to recover the secret in the following way:

```
% sharedkey -o group.recover -r /tmp/share.*/share002.mat /tmp/share.*/share000.mat
signature  2048/RSA (Encrypt or Sign) 6018e4d1edcd8801 2011-09-17
Key fingerprint: e6ba 6252 120b 2f6a b702 4223 6018 e4d1 edcd 8801
uid              /etc/group shared 2/3 by agc Sat Sep 17 14:32:28 2011
netpgp passphrase:
% diff /etc/group group.recover
% ls -l /etc/group group.recover
-rw-r--r--  1 root  wheel  535 Sep  4 21:44 /etc/group
-rw-------  1 agc   wheel  535 Sep 18 16:46 group.recover
%
```

7. Further development

At the present time, there are a number of areas in which these methods and libraries are used. Over the next few years, with the advent of cloud computing, it is highly likely that we will see much more need for encrypted storage, and other forms of protection based around signatures and encryption.

Whilst homomorphic cryptography is in its infancy, other more tried and tested means are available to developers right now.

We have also outgrown the era of *"Doesn't match the checksum? Discard, and re-send"*. The use of erasure codes to detect and correct data changes now gives us the ability to get good communications even over lossy links, or failing hardware, without invoking any bottlenecks on congestion control for network links, or destructive and wasteful multiple block reads on failing disks.

The possible areas for future development that are possible are:

- Communications - as previously mentioned, strict conformance to a 32-bit or smaller checksum, with a failure mode as “re-set caching, and re-send data” can be considered too draconian in a time when data can be protected with erasure and error codes.

- File systems - in an era where data protection can be done at a much finer grained level than a device, or a complete file system, we should be able to protect the more important metadata with better protection than the data itself (if the metadata cannot be read, then the data will not be able to be read either). The extreme case with this is that a file system's headers could be protected with circa or similar protection, rather than having fixed copies across the devices. Right now, if a file system check fails, it is up to the administrator to debug the file system - it is to be hoped that circa protection against media corruption might prove more effective
- p2p - torrent downloading relies on strict checksum matching to detect problems with downloaded data slices. circa-style protection might prove more beneficial and effective in this application as well
- digests - even when using existing (possibly weak) digests to protect file integrity, the use of an HMAC might prove more effective, since HMACs are more resistant to collision than their underlying digests.
- threshold schemes are very effective in a number of application areas, and do not require shared keys to be distributed amongst a group, or even for a group membership to be consistent.
- IDA/SSSS devices - it is quite possible now to develop a device using SSSS or IDA technology, perhaps using PUD to implement in userspace, providing resilience across the network for data which could not otherwise be viewed by an attacker.
- Investigation of Turbocodes and LDPC as a more efficient means than Reed-Solomon for providing forward erasure codes, and error correction.

8. Related work

We have seen a number of related pieces of work, including Reliable Multicast Data Protocol, which included an early version of Forward Erasure Coding. A development of that concept was the Tahoe-LAFS file system, which is a distributed, resilient file system, built on top of IDA, erasure codes, and encryption.

9. Conclusion

A number of different types of data protection were defined - modification detection, error and erasure coding as a means to correction of changes to data, protection of data from unauthorised viewing by means of encryption, and redundancy of data through threshold schemes and "shares" - and the means to effect these types of protection were discussed. Some basic building blocks have been written and made available; these were introduced and discussed. Finally, some combinations of the basic building blocks were discussed, leading to new applications, new areas for research, and some future work.

In all, the schemes for data protection can be applied to many of the areas in which we work every day - devices and device drivers, communications systems from layer 2 upwards, file systems, application libraries and utilities. We need to keep re-visiting some of the technology which we just use without querying, asking if there are ways to make things more efficient by using newer technology which has been invented and discovered.

The data protection building blocks are designed to let these newer technologies be built and used, and we have shown how newer versions of protocols which include erasure coding can perform better than existing "match the checksum or discard, making sure to take more care when receiving the same block the second time around".

10. References

```
Crooks2009         Netpgp - BSD-licensed encryption
Crooks2010         SSH and PGP Key Convergence
ECMA1996           ECMA-130:
                   Data interchange on read-only 120mm optical data disks (CD-ROM),
           http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-130.pdf,
                   Ecma International, June 1996.
Gentry2009         http://www-03.ibm.com/press/us/en/pressrelease/27840.wss
Neal1996           http://www.cs.utoronto.ca/~radford/ftp/LDPC-2006-02-08/index.html
Shamir1979         "How to share a secret", Communications of the ACM 22 (11): 612-613,
                   http://dx.doi.org/10.1145%2F359168.359176
WikipediaHMAC2011  http://en.wikipedia.org/wiki/HMAC
WikipediaCRC2011   http://en.wikipedia.org/wiki/Cyclic_redundancy_check
```

11. Repository References

[1] crc http://cvsweb.netbsd.org/bsdweb.cgi/othersrc/external/bsd/crc/?only_with_tag=MAIN
[2] hmac http://cvsweb.netbsd.org/bsdweb.cgi/othersrc/external/bsd/hmac/?only_with_tag=MAIN
[3] merkle/tigertree/hashtree http://cvsweb.netbsd.org/bsdweb.cgi/othersrc/external/bsd/hashtree/?only_with_tag=MAIN
[4] reed-solomon http://cvsweb.netbsd.org/bsdweb.cgi/othersrc/external/bsd/rs/?only_with_tag=MAIN
[5] lamport http://cvsweb.netbsd.org/bsdweb.cgi/othersrc/external/bsd/lamport/?only_with_tag=MAIN
[6] circa http://cvsweb.netbsd.org/bsdweb.cgi/othersrc/external/bsd/circa/?only_with_tag=MAIN
[7] bag http://cvsweb.netbsd.org/bsdweb.cgi/othersrc/external/bsd/bag/?only_with_tag=MAIN # not yet
[8] encryption http://cvsweb.netbsd.org/bsdweb.cgi/src/crypto/external/bsd/netpgp/?only_with_tag=MAIN
[9] sharedkey http://cvsweb.netbsd.org/bsdweb.cgi/othersrc/crypto/external/bsd/sharedkey/?only_with_tag=MAIN
[10] threshold http://cvsweb.netbsd.org/bsdweb.cgi/othersrc/external/bsd/threshold/?only_with_tag=MAIN
[11] ssss http://cvsweb.netbsd.org/bsdweb.cgi/othersrc/crypto/external/bsd/ssss/?only_with_tag=MAIN