MinCrypt algorithm

Design document

Draft v0.0.1

This document introduces the algorithm designed by Michal Novotny as a minimal cryptographic algorithm that's using both salt and password values as it's cryptographic base. Main reason to implement such an algorithm is to provide an easy to write algorithm with minimum hardware requirements to be used, e.g. for embedded devices such as routers and routers running on Linux operating system. Also, the algorithm was designed for simplicity to be easily ported to Java & Symbian languages for the mobile devices such as cell-phones supporting those technologies, PDAs or maybe even e-book readers. The algorithm is symmetric and the secret values, i.e. salt and password values, are being used to create the initialization vectors. Those vectors are used for encryption and decryption itself later.

Author, Michal Novotny, created this algorithm for his own purposes and implemented it in a simple program written in C language under Fedora 14 system. Along with this algorithm two helper tools were developed for algorithm testing purposes. One tool is designed to check the relevance between 2 files (used for relevance checking between a two identical files encrypted with just different salt and password values) and second tool is designed to create a chart graph to display occurrences ratio for each of the ASCII characters, i.e. for characters with ASCII codes 0 to 255.

Document overview

1. Initialization vector generation algorithm

The vector initialization routine is being performed by crypt_set_password() function in the example source code included with this paper. The arguments being accepted by this function are salt value, password value and vector multiplier value. Salt and password values are being used to create the initialization vectors and vector multiplier value is being used to increase the size of the initialization vectors array.

The initialization vector size is being computed as length of salt multiplied by length of password multiplied by the vector multiplier value. Optimal value of vector multiplier value, as implemented in the example source code, should be 32 (0x20) not to create the huge amount of memory just for the initialization vector but to allocate the vector array big enough to carry variety of initialization vectors.

First of all, we compute the value from salt called the iSalt value (since it's the unsigned 32-bit integer value as defined by uint32_t type in the example code). The value is being computed by ASCII value of each character of salt being with value with the value of bits the chunk is encoded on. This is the nearest high value of the chunk bits, i.e. minimum number of bits the value can be encoded on like 17 bits for 128 kB (131072 bytes) as used in the example code so it's calculated using the following formula:

$$\text{iSalt} = \sum_{i=0}^{lenSalt} (salt_i + chunkBits)$$

When we have iSalt value we can go on by calculating initial value from the password using iSalt value we've just calculated above:

$$\text{initial} = \sum_{i=0}^{lenPassword} ((password_i + iSalt) << i)$$

After we have both iSalt and initial values calculated we can calculate the initialization vectors using the following formula:

$$\text{passSum} = \sum_{i=0}^{lenPassword} password_i$$

$$\text{passidx} = \text{passSum} - \text{password}_i$$

$$\text{power} = \text{passSum} + \text{i} / \text{password}_i)$$

$$\forall iv_i = (initial + iSalt + (password_{passidx})^{power}$$

Sum of all initialization vector values with addition of initial value is called the iVal, that will be used lated for the encryption/decryption itself, i.e. this is calculated using following formula:

$$\text{iVal} = \text{initial} + \sum_{i=0}^{vectorSize} iv_i$$

2. Encryption algorithm

The encryption is symmetric so that there are not so many differences in the encryption and decryption implementations of this algorithm. The important issue is the data integrity checking which is being performed by standard CRC-32 algorithm.

Since the algorithm is designed mainly for file encryption which is being read sequentially in a small blocks called chunks the chunk identifier is being present already in the design.

For each data block the CRC-32 value of the original block is being computed and later saved to the end of each encrypted chunk.

The encryption itself is based on the original data being shifted by initialization vectors and iVal. For each character in the block read the encrypted character value is computed formula:

$$\begin{aligned} &\text{vectorIndex } = (\text{i} << (\text{id} * \text{size}) + \text{i})) \\ &out_i = (iVal - chunkCrc - iv_{vectorIndex}) - in_i \end{aligned}$$

where:
in is the input data block
out is the output data block
iVal is the sum value calculated above
chunkCrc is the CRC-32 value of the chunk being processed
iv is the initialization vector array
id is the chunk identifier
size is the size of one chunk
i is the index of the characters being processed in the currently processed chunk

The CRC-32 value encoded on 4 bytes is being added to the end of the chunk for the data integrity checking purposes when decrypting such a file. Provided the fact each chunk ends with CRC-32 encoded on 4 bytes we can calculate number of chunks if we know what the sizes of original file and encrypted file are by finding the nearest power of two for their difference divided by 4. The value won't be most likely the power of two since the last chunk is usually somewhat smaller than the chunk size.

This is more like an encoding however with the different values of password and salt the line angle is different from the beginning when put on the graph.

Triage: Try to use such an algorithm for 2 very similar values of password and salt, i.e. something like using password “test” with salt “test” and password “test” with salt value of “tesu”. The difference between “test” and “tesu” is one bit since 't' is having binary value of 1110100 and 'u' of 1110101, therefore one bit difference.

3. Decryption algorithm

The decryption part of this algorithm is using the same shifting code as the encryption however the chunks being read have to be exactly 4 bytes longer than those used for encryption to carry the CRC-32 value.

First of all, the value of CRC-32 should be decoded and the chunk should be truncated by those 4 bytes. This chunk should be having the value as used when encryption and the decryption part of the algorithm is the same as for the encryption, i.e. computed as:

The encryption itself is based on the original data being shifted by initialization vectors and iVal. For each character in the block read the encrypted character value is computed formula:

$$\begin{aligned}&\text{vectorIndex } = (\text{i} << (\text{id} * \text{size}) + \text{i}))\\ &out_i = (iVal - chunkCrc - iv_{vectorIndex}) - in_i\end{aligned}$$

where:
in is the input data block
out is the output data block
iVal is the sum value calculated above
chunkCrc is the CRC-32 value of the chunk being processed
iv is the initialization vector array
id is the chunk identifier
size is the size of one chunk
i is the index of the characters being processed in the currently processed chunk

Then the CRC-32 value is being used for the data integrity check to fail immediately when the one chunk is having invalid CRC-32 checksum, i.e. not to try to decrypt when it's already known that we have an error during decryption.

4. Example: Initialization vectors generation

This is the example of crypt_set_password() function to generate the iVal value and the initialization vectors. The comments in the code were preserved for readability. The DEFAULT_VECTOR_MULT value is set to 32 (0x20) as noted to be optimal in this paper. GetNearestPowerOfTwo() is the function to calculate minimum amount of bits required to store the chunk size specified. The result of the function is being saved into the bits variable.

```
int crypt_set_password(char *salt, char *password, int vectorMultiplier)
{
        uint32_t shift = 0, val = 0, iSalt = 0, initial = 0;
        uint64_t shifts = 0, initialValue = 0, tmp = 0;
        int num = 0, lenSalt, lenPass, i, vectorMult, bits, passSum;
        char *savedpass;

        vectorMult = (vectorMultiplier < 0) ? DEFAULT_VECTOR_MULT : vectorMultiplier;

        lenSalt = strlen(salt);
        lenPass = strlen(password);
        _vectorSize = lenSalt * lenPass * vectorMult;

        getNearestPowerOfTwo(BUFFER_SIZE, &bits);
        DPRINTF("Chunk is encoded on %d bits\n", bits);

        while (val = *salt++)
                iSalt = pow(val, ++num) * bits;

        DPRINTF("%s: iSalt = 0x%"PRIx32"\n", __FUNCTION__, iSalt);

        num = 0;
        savedpass = strdup(password);
        passSum = 0;
        while (val = *password++) {
                passSum += val;
                initial += (val + iSalt) << ++num;
        }

        DPRINTF("%s: initial = 0x%"PRIx32"\n", __FUNCTION__, initial);

        if (_iv != NULL)
                _iv = realloc( _iv, _vectorSize * sizeof(uint32_t) );
        else
                _iv = malloc( _vectorSize * sizeof(uint32_t) );

        for (i = 0; i < _vectorSize; i++) {
                val = savedpass[i % strlen(savedpass)];
                _iv[i] = (initial
                                        + iSalt
                                        + (uint32_t)pow( savedpass[(passSum - val) % strlen(savedpass) ], (passSum + i) / val)
                                );

                //DPRINTF("Got initialization vector %d: %08" PRIx32"\n", i, _iv[i]);
                initialValue += _iv[i];
        }
        free(savedpass);

        DPRINTF("%s: Vector generated, elements: %d\n", __FUNCTION__, _vectorSize);

        _ival = initial + initialValue;
        DPRINTF("%s: initialValue = 0x%"PRIx64"\n", __FUNCTION__, _ival);
}
```

5. Example: Block encryption example

This is the example of block encryption implementation. It's having the chunk id in the id variable and the output size is being saved into the newSize variable. The comments are present in the code to increase readability.

```
char *crypt_encrypt(unsigned char *block, int size, uint32_t crc, int id, int *newSize)
{
	int i;
	uint32_t old_crc = 0;
	unsigned char *out = NULL;
	unsigned char data[4] = { 0 };
	int csize = size;

	if (_iv == NULL) {
		fprintf(stderr, "Error: Initialization vectors are not initialized\n");
		if (newSize != NULL)
			*newSize = -1;
		return NULL;
	}

	if (crc == 0) {
		/* We make the output block 4 bytes bigger to carry the CRC-32 value */
		csize += 4;
		old_crc = crc32_block(block, size, 0xFFFFFFFF);
		DPRINTF("%s: Block CRC = 0x%08"PRIx32"\n", __FUNCTION__, old_crc);
	}
	else
		old_crc = crc;

	out = malloc( csize * sizeof(unsigned char) );
	memset(out, 0, csize);
	if (out == NULL) {
		DPRINTF("%s: Cannot allocate %d bytes of memory\n", __FUNCTION__, csize);
		return NULL;
	}

	for (i = 0; i < size; i++)
		out[i] = (_ival - old_crc - (_iv[i % _vectorSize] << ((id * size) + i))) - block[i];

	if (crc == 0) {
		UINT32STR(data, old_crc);
		out[i++] = data[0];
		out[i++] = data[1];
		out[i++] = data[2];
		out[i++] = data[3];
	}

	if (newSize != NULL)
		*newSize = csize;

	return out;
}
```

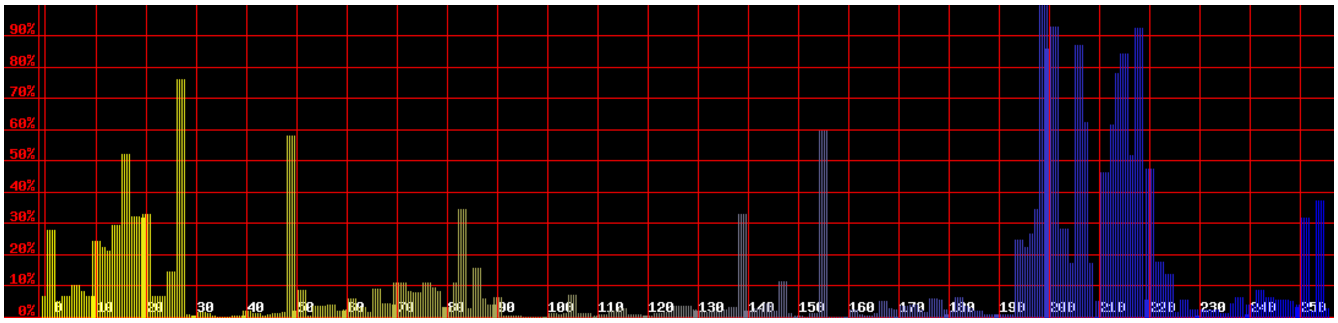
6. Example: Block decryption example

This is the example of block decryption implementation as specified in the chapter 3.

```
char *crypt_decrypt(unsigned char *block, int size, int crc, int id, int *newSize)
{
	unsigned char data[4] = { 0 }, *out = NULL;
	uint32_t old_crc = 0, new_crc = 0;
	unsigned int csize = size;

	if (_iv == NULL) {
		fprintf(stderr, "Error: Initialization vectors are not initialized\n");
		if (newSize != NULL)
			*newSize = -1;
		return NULL;
	}

	data[0] = block[size - 4];
	data[1] = block[size - 3];
	data[2] = block[size - 2];
	data[3] = block[size - 1];
	old_crc = GETUINT32(data);

	out = crypt_encrypt(block, size-(crc ? 4 : 0), old_crc, id, &csize);

	new_crc = crc32_block(out, csize, 0xFFFFFFFF);
	DPRINTF("%s: Checking CRC value for %d byte-block (0x%08"PRIx32" [expected] %c= 0x%08"PRIx32" [found])\n",
			__FUNCTION__, size - (crc ? 4 : 0), old_crc, old_crc == new_crc ? '=' : '!', new_crc);

	if (old_crc != new_crc) {
		free(out);
		if (newSize != NULL)
			*newSize = -1;

		DPRINTF("%s: CRC value doesn't match!\n", __FUNCTION__);
		return NULL;
	}

	if (newSize != NULL)
		*newSize = csize;

	return out;
}
```

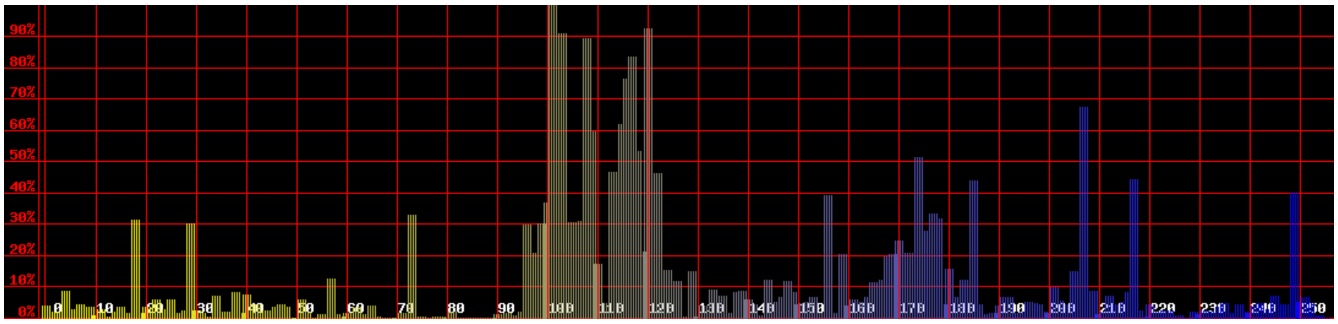
7. Analysis: Relevance testing

a) Using salt “*test*” and password “*test*”



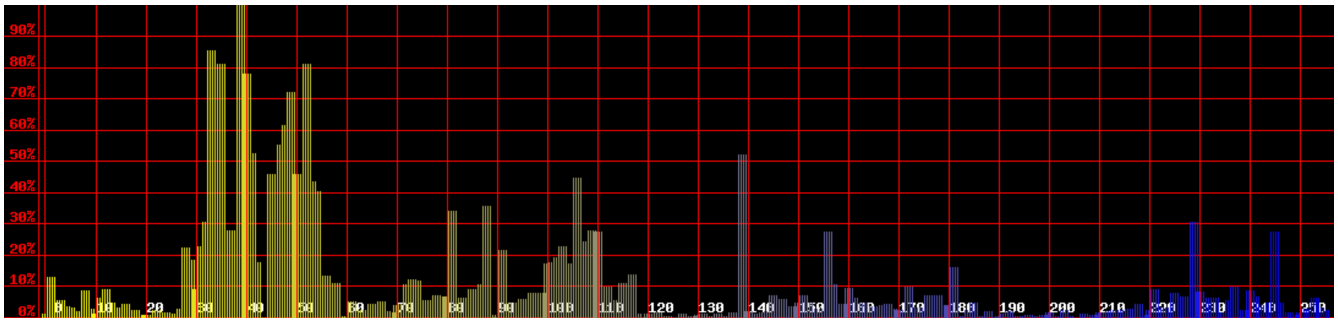
Relevance with file encrypted using salt “test” and password “tesu”: 5 of 6857 bytes (0.0735%)
Relevance with file encrypted using salt “tesu” and password “tesu”: 57 of 6857 bytes (0.08382%)
Relevance with file encrypted using salt “tesu” and password “test”: 5 of 6857 bytes (0.0735%)

b) Using salt “*test*” and password “*tesu*”



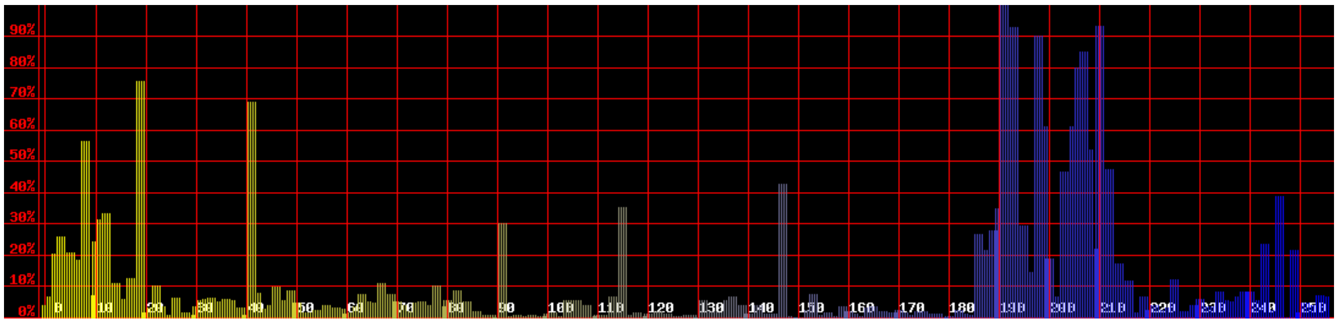
Relevance with file encrypted using salt “test” and password “test”: 5 of 6857 bytes (0.0735%)
Relevance with file encrypted using salt “tesu” and password “tesu”: 5 of 6857 bytes (0.0735%)
Relevance with file encrypted using salt “tesu” and password “test”: 5 of 6857 bytes (0.0735%)

c) Using salt “*tesu*” and password “*test*”



Relevance with file encrypted using salt “tesu” and password “tesu”: 5 of 6857 bytes (0.0735%)
Relevance with file encrypted using salt “test” and password “tesu”: 5 of 6857 bytes (0.0735%)
Relevance with file encrypted using salt “test” and password “test”: 5 of 6857 bytes (0.0735%)

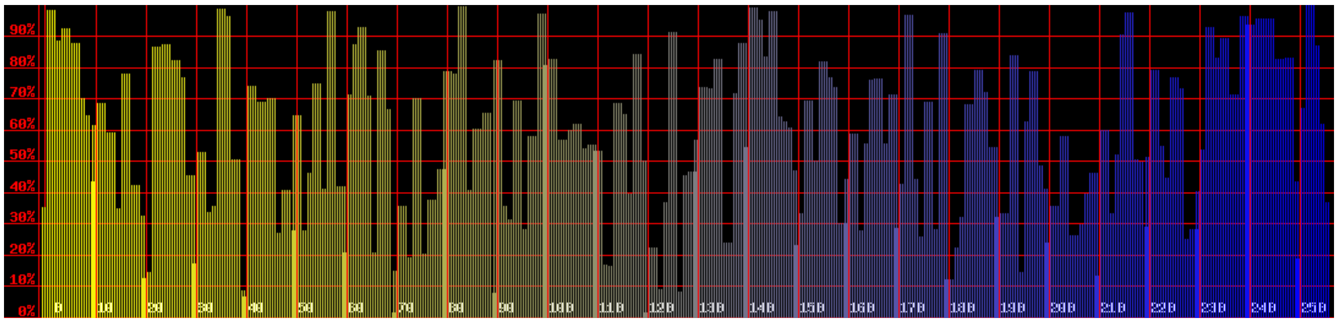
d) Using salt “*tesu*” and password “*tesu*”



Relevance with file encrypted using salt “test” and password “test”: 57 of 6857 bytes (0.08382%)
Relevance with file encrypted using salt “test” and password “tesu”: 5 of 6857 bytes (0.0735%)
Relevance with file encrypted using salt “tesu” and password “test”: 5 of 6857 bytes (0.0735%)

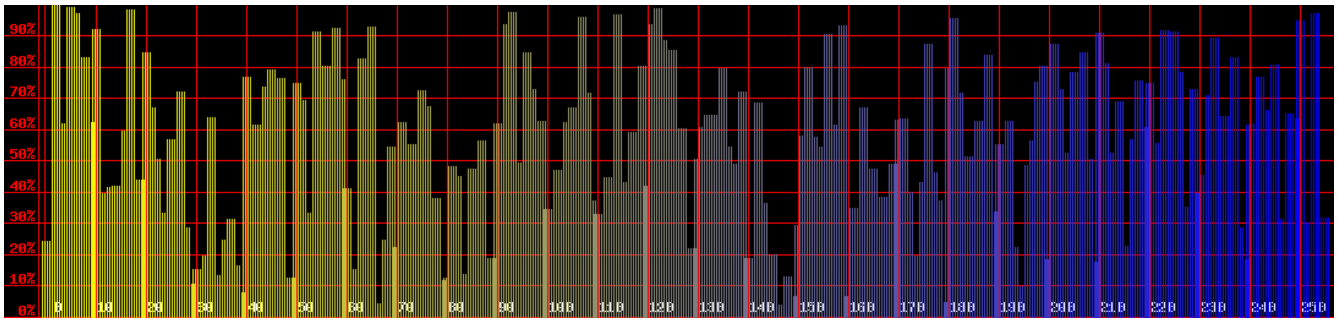
Difference between 15 MiB files follows.

A) Using salt “*test*” and password “*test*”



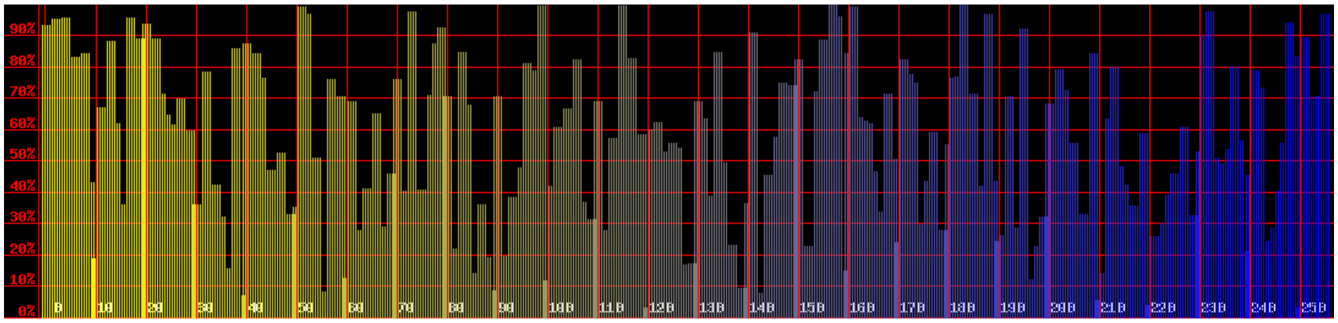
Relevance with file encrypted using salt “test” and password “tesu”: 469 of 15288012 bytes (0.0031%)
Relevance with file encrypted using salt “tesu” and password “tesu”: 469 of 15288012 bytes (0.0031%)
Relevance with file encrypted using salt “tesu” and password “test”: 469 of 15288012 bytes (0.0031%)

B) Using salt “*test*” and password “*tesu*”



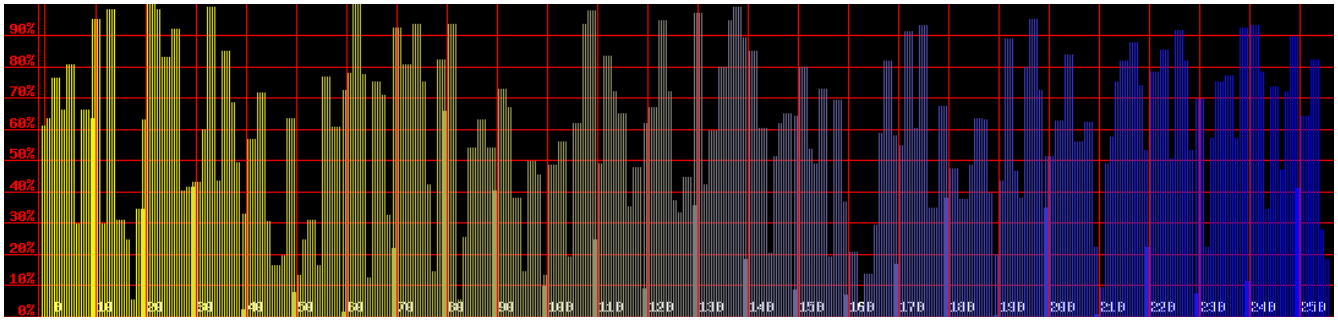
Relevance with file encrypted using salt “test” and password “test”: 469 of 15288012 bytes (0.0031%)
Relevance with file encrypted using salt “tesu” and password “tesu”: 469 of 15288012 bytes (0.0031%)
Relevance with file encrypted using salt “tesu” and password “test”: 469 of 15288012 bytes (0.0031%)

C) Using salt “*tesu*” and password “*test*”



Relevance with file encrypted using salt “tesu” and password “tesu”: 469 of 15288012 bytes (0.0031%)
Relevance with file encrypted using salt “test” and password “tesu”: 469 of 15288012 bytes (0.0031%)
Relevance with file encrypted using salt “test” and password “test”: 469 of 15288012 bytes (0.0031%)

D) Using salt “*tesu*” and password “*tesu*”



Relevance with file encrypted using salt “tesu” and password “test”: 469 of 15288012 bytes (0.0031%)
Relevance with file encrypted using salt “test” and password “tesu”: 469 of 15288012 bytes (0.0031%)
Relevance with file encrypted using salt “test” and password “test”: 469 of 15288012 bytes (0.0031%)