Băhēm

A Provably Secure Symmetric Cipher

M. Rajululkahf[1]

April 13, 2022

Overview

This paper proposes Băhēm; a symmetric cipher that, when used with a pre-shared secret key $\mathbf{k}$, no cryptanalysis can degrade its security below $\mathrm{H}(\mathbf{k})$ bits of entropy, even under Grover's algorithm [1] or even if it turned out that $\mathsf{P} = \mathsf{NP}$.

Băhēm's security is very similar to that of the one-time pad (OTP), except that it does not require the communicating parties the inconvenient constraint of generating a large random pad in advance of their communication. Instead, Băhēm allows the parties to agree on a small pre-shared secret key, such as $|\mathbf{k}| = 128$ bits, and then generate their random pads in the future as they go.

For any operation, be it encryption or decryption, Băhēm performs only 4 bitwise exclusive-or operations (XORs) per cleartext bit including its 2 overhead bits. If it takes a CPU 1 cycle to perform an XOR between a pair of 64 bit variables, then a Băhēm operation takes $4 \div 8 = 0.5$ cycles per byte. Further, all Băhēm's operations are independent, therefore a system with n many CPU cores can perform $0.5 \div n$ cpu cycles per byte per wall-clock time.

While Băhēm has an overhead of 2 extra bits per every encrypted cleartext bit, its early single-threaded prototype implementation achieves a faster *decryption* than OpenSSL's ChaCha20's, despite the fact that Băhēm's ciphertext is 3 times larger than ChaCha20's. This support that the 2 bit overhead is practically negligible for most applications.

Băhēm's early prototype has a slower *encryption* time than OpenSSL's ChaCha20 due to its use of a true random number generator (TRNG). However, this can be trivially optimised by gathering the true random bits in advance, so Băhēm gets the entropy conveniently when it runs.

Aside from Băhēm's usage as a provably-secure general-purpose symmetric cipher, it can also be used, in some applications such as password verification, to enhance existing hashing functions to become provably one-way, by using Băhēm to encrypt a predefined string using the hash as the key. A password is then verified if its hash decrypts the Băhēm ciphertext to retrieve the predefined string.

[1] Author's e-mail address: {last name}`@pm.me`

Notation

$\mathrm{H}(\mathbf{x})$ Shannon's entropy of random variable $\mathbf{x}$.

$|\mathbf{x}|$ Number of bits in tuple $\mathbf{x} = (x_0, x_1, \ldots, x_{|\mathbf{x}|-1})$.

$\mathbf{x} \oplus \mathbf{y}$ Bitwise exclusive-or operation between two variables. If one variable is shorter than the other, then the shorter will repeat itself following modular arithmetics. For example, if $|\mathbf{x}| = 5$ and $|\mathbf{y}| = 2$, then the lacking bits y_2, y_3, y_4 will be assumed to be y_0, y_1, y_2.

$\mathrm{random}(n) = (r_0, r_1, \ldots, r_n)$ A sequence of n many random bits generated by a TRNG.

$\mathbf{k} = (k_0, k_1, \ldots, k_{|\mathbf{k}|-1})$ A pre-shared secret key with enough $\mathrm{H}(\mathbf{k})$ for use case. Ideally $\mathbf{k} = \mathrm{random}(|\mathbf{k}|)$. Size $|\mathbf{k}|$ can be chosen arbitrarily to offer adequate security for the use case, as there is no block structure in Băhēm.

$\mathbf{m} = (m_0, m_1, \ldots, m_{|\mathbf{m}|-1})$ An arbitrarily long cleartext message.

$\mathbf{p} = \mathrm{random}(|\mathbf{m}|), \mathbf{q} = \mathrm{random}(|\mathbf{m}|)$ A pair of uniformly distributed random one-time pads. This is generated dynamically by the implementation, transparently from the user, for every new communication session.

$\hat{\mathbf{p}}, \hat{\mathbf{q}}, \hat{\mathbf{m}}$ Encrypted forms of $\mathbf{p}, \mathbf{q}, \mathbf{m}$, respectively.

Contents

1 Background

When Alice and Bob privately met last time, they used a TRNG to generate enough random bits to use for encrypting their future communications over insecure channels.

Ideally, Alice and Bob wanted to use the TRNG to generate terabytes worth of truly random bits, for the purpose of using the OTP as their encryption technique in the future. They liked that the OTP is proven to be secure. However, they realised that trying to discretely carry terabytes worth of data, and maintaining their health, entails a needless overhead and risk.

As a result, Alice and Bob agreed to use the TRNG to generate only 28 bits of truly random data, $\mathbf{k} = [k_0, k_1, \ldots, k_{127}]$, as their pre-shared secret key.

Alice's and Bob' reasoning is that, securely maintaining small amounts of data, such as 128 bits, is much easier than that of larger data, such as terabytes, and yet 128 bits of entropy is enough to render Eva's brute-forcing attempts impractical.

However, they found that today's state-of-art symmetric ciphers, such as ChaCha20 [2] and AES [3], are not provably secure, but rather simply that no one could fully break them *yet* [4]. Further, one-way functions may not even exist, as the P versus NP is still one of the unsolved Millennium problems[2].

Băhēm solves the problems above by offering the proven security of the OTP, without the inconvenience of having to exchange large one-time pads in advance, for a negligible expense of accompanying each ciphertext with 2 extra bits, only.

2 Proposed Algorithm: Băhēm

Algorithms 1 and 2 show Băhēm's encryption and decryption in batch mode. The batched mode is generally not very practical for most applications, as data often comes in streams. However, the batch mode looks simpler, and this simplicity can aid explaining Băhēm's concept more efficiently.

Algorithms 3 and 4 show the same, except that the inputs and the outputs are interleaved on bit-by-bit basis. This interleaved version is identical to the batched one, except for re-ordering its output bits.

An implementation may choose a different data format where interleaving happens on the basis of other data structures than bits. The bit-by-bit interleaving algorithms are only shown to demonstrate that Băhēm is practically useful when dealing with data streams.

Algorithm 1: Batched Băhēm encryption

input : $\mathbf{k}, \mathbf{m}$
output: $\hat{\mathbf{p}}, \hat{\mathbf{q}}, \hat{\mathbf{m}}$

$\mathbf{p} \leftarrow \text{random}(|\mathbf{m}|)$
$\mathbf{q} \leftarrow \text{random}(|\mathbf{m}|)$
$\hat{\mathbf{p}} \leftarrow \mathbf{p} \oplus \mathbf{k}$
$\hat{\mathbf{q}} \leftarrow \mathbf{q} \oplus \mathbf{k}$
$\hat{\mathbf{m}} \leftarrow \mathbf{m} \oplus \mathbf{p} \oplus \mathbf{q}$
return $\hat{\mathbf{p}}, \hat{\mathbf{q}}, \hat{\mathbf{m}}$

Algorithm 2: Batched Băhēm decryption

input : $\mathbf{k}, \hat{\mathbf{p}}, \hat{\mathbf{q}}, \hat{\mathbf{m}}$
output: $\mathbf{m}$

$\mathbf{p} \leftarrow \hat{\mathbf{p}} \oplus \mathbf{k}$
$\mathbf{q} \leftarrow \hat{\mathbf{q}} \oplus \mathbf{k}$
$\mathbf{m} \leftarrow \hat{\mathbf{m}} \oplus \mathbf{p} \oplus \mathbf{q}$
return $\mathbf{m}$

Algorithm 3: Interleaved Băhēm encryption

input : $\mathbf{k}, m_0 \| m_1 \ldots$
output: $\hat{p}_0 \| \hat{q}_0 \| \hat{m}_0 \| \hat{p}_1 \| \hat{q}_1 \| \hat{m}_1 \ldots$

while $m_i \leftarrow \text{read}(1)$ **do**
    $p_i \leftarrow \text{random}(1)$
    $q_i \leftarrow \text{random}(1)$
    $j \leftarrow i \mod |\mathbf{k}|$
    $\hat{p}_i \leftarrow p_i \oplus k_j$
    $\hat{q}_i \leftarrow q_i \oplus k_j$
    $\hat{m}_i \leftarrow m_i \oplus p_i \oplus q_i$
    $\text{write}(\hat{p}_i \| \hat{q}_i \| \hat{m}_i)$

Algorithm 4: Interleaved Băhēm decryption

input : $\mathbf{k}, \hat{p}_0 \| \hat{q}_0 \| \hat{m}_0 \| \hat{p}_1 \| \hat{q}_1 \| \hat{m}_1 \ldots$
output: $m_0, m_1, \ldots$

while $\hat{p}_i, \hat{q}_i, m_i \leftarrow \text{read}(3)$ **do**
    $j \leftarrow i \mod |\mathbf{k}|$
    $p_i \leftarrow \hat{p}_i \oplus k_j$
    $q_i \leftarrow \hat{q}_i \oplus k_j$
    $m_i \leftarrow \hat{m}_i \oplus p_i \oplus q_i$
    $\text{write}(m_i)$

[2] `http://claymath.org/millennium-problems`

3 Security Proof

Băhēm can be thought as multiple OTPs, one of which recurses into itself for once. Therefore, the proving strategy that is adopted in this paper is to show that Băhēm is made of recursion of OTPs, and that this recursion is also an OTP.

Theorem 3.1 (Shannon's perfect secrecy for OTP)**.** *For any pair of bit tuples* $\mathbf{x}$ *and* $\mathbf{y}$*, the cryptosystem* $\mathbf{x} \oplus \mathbf{y} = \mathbf{z}$ *is said to have* perfect secrecy *when, for any* i^{th} *bit,* $\Pr(x_i = 0|z_i) = \Pr(x_i = 0)$*, which is true if and only if* $\Pr(y_i = 0) = 0.5$*.*

Proof. Algorithm 1 shows that Băhēm's encryption outputs $\hat{\mathbf{p}}$, $\hat{\mathbf{q}}$ and $\hat{\mathbf{m}}$, each of which can be viewed as the ciphertext output of an OTP cryptosystem as shown below:

$\mathbf{k} \oplus \mathbf{p} = \hat{\mathbf{p}}$**.** Since $\mathbf{p} \leftarrow \text{random}(|\mathbf{m}|)$ by definition, it is implied that, for any $i \in \{0, 1, \ldots, |\mathbf{m}|\}$, $\Pr(p_i = 0) = 0.5$. Therefore, it follows by theorem 3.1 that this cryptosystem has perfect secrecy; that is, reveals no information about the pre-shared secret key $\mathbf{k}$.

$\mathbf{k} \oplus \mathbf{q} = \hat{\mathbf{q}}$**.** Since $\mathbf{q} \leftarrow \text{random}(|\mathbf{m}|)$ by definition, this is identical to the previous cryptosystem, and therefore has perfect secrecy as well.

$\mathbf{p} \oplus \mathbf{q} \oplus \mathbf{m} = \hat{\mathbf{m}}$**.** This can be viewed as two OTP cryptosystems one recursing into the other:

$\mathbf{p} \oplus \mathbf{q} = \mathbf{z}$**.** For any $i \in \{0, 1, \ldots, |\mathbf{m}|\}$, $\Pr(q_i = 0) = 0.5$ is implied by definition as stated earlier, therefore this is cryptosystem has perfect secrecy; that is, it reveals no information about $\mathbf{p}$ should an adversary get $\mathbf{z}$. Likewise, since $\Pr(p_i = 0) = 0.5$ is also true as stated earlier as well, it also follows that no information is revealed about $\mathbf{q}$ either should an adversary get $\mathbf{z}$.

Since no information can be revealed about $\mathbf{p}$ and $\mathbf{q}$, in the case the adversary obtains bits of $\mathbf{z}$, it has to follow that no information can be revealed about the pre-shared secret key $\mathbf{k}$.

$\mathbf{m} \oplus \mathbf{z} = \hat{\mathbf{m}}$**.** Since $\mathbf{z}$ is the ciphertext of a cryptosystem with perfect secrecy, and since Băhēm does not share it, it has to follow that, for any $i \in \{0, 1, \ldots, |\mathbf{m}|\}$, $\Pr(z_i = 0) = 0.5$. Therefore, it follows by theorem 3.1 that this cryptosystem has perfect secrecy; that is, reveals no information about the cleartext message $\mathbf{m}$.

Since algorithm 3 is identical to algorithm 1, except for only adopting a different data storage format, it has to follow that, both, algorithms 1 and 3 offer perfect secrecy in that no information about $\mathbf{k}$ or $\mathbf{m}$ can be revealed from $\hat{\mathbf{p}}, \hat{\mathbf{q}}, \hat{\mathbf{m}}$.

Theorem 3.2 (Băhēm's perfect secrecy)**.** *An adversary that obtains* $\hat{\mathbf{p}}$*,* $\hat{\mathbf{q}}$ *and* $\hat{\mathbf{m}}$*, cannot gain information about the pre-shared secret key* $\mathbf{k}$ *or the cleartext message* $\mathbf{m}$*.*

Since no information can be revealed about $\mathbf{k}$ or $\mathbf{m}$ from Băhēm's encrypted output, it has to follow that, asymptotically, no cryptanalysis can reduce Băhēm's key brute-forcing space below $2^{\mathrm{H}(\mathbf{k})}$.

Theorem 3.3 (Băhēm's security)**.** *No cryptanalysis can reduce Băhēm's security below* $\mathrm{H}(\mathbf{k})$ *bits.*

∎

4 Implementation Example

4.1 C Functions

In this example, the caller is expected to initialise a 128 bits key $\mathbf{k}$, a pair of random pads, $\mathbf{p}$ and $\mathbf{q}$, each of which is `len` $\times$ 64 bits long, in order to encrypt a `len` $\times$ 64 bits long cleartext message $\mathbf{m}$. The encryption happens in-place, so the caller does not have to allocate separate memory for the ciphertext.

```
void baheem_enc(
    uint64_t *k, /* 128bit pre-shared key */
    uint64_t *p, /* random pad 1          */
    uint64_t *q, /* random pad 2          */
    uint64_t *m, /* message               */
    size_t  len  /* length of m = p = q   */
) {
    size_t i;
    for (i = 0; i < len; i++) {
        m[i] ^= p[i] ^ q[i];
        p[i] ^= k[0];
        q[i] ^= k[1];
    }
}
```

Likewise, the following is an example implementing the corresponding in-place decryption function.

```
void baheem_dec(... same input ...) {
    size_t i;
    for (i = 0; i < len; i++) {
        p[i] ^= k[0];
        q[i] ^= k[1];
        m[i] ^= p[i] ^ q[i];
    }
}
```

4.2 An Early Prototype: Alyal

Alyal is an early single-threaded prototype implementation that uses Băhēm to encrypt and decrypt files, mainly to demonstrate Băhēm's practical utility with real-world scenarios. Internally, Alyal uses the `baheem_enc` and `baheem_dec` functions that were presented earlier in this section.

4.2.1 Installation and Usage

```
> git clone \
    https://codeberg.org/rajululkahf/alyal
> cd alyal
> make
> dd bs=1MB count=500 \
    if=/dev/zero of=test.txt
> ./alyal enc test.txt test.enc
> ./alyal dec test.enc test.enc.txt
> shasum *
```

4.2.2 Benchmark

Table 1 shows an early benchmark that was performed on a machine with Intel(R) Core(TM) i5-3570K CPU @ 3.40GHz.

	OpenSSL ChaCha20	Alyal Băhēm
Encrypt 500MB	**1.07** secs	4.25 secs
	1.03 secs	4.28 secs
	1.05 secs	4.30 secs
Decrypt 500MB	1.07 secs	**0.85** secs
	1.09 secs	**0.91** secs
	1.15 secs	**0.84** secs

Table 1: Wall-clock run-time comparison between OpenSSL's ChaCha20, and Alyal's Băhēm implementation.

The encryption and decryption functions of Băhēm are identical in their number of instructions and the amount of IO operations, except that the encryption one pulls bits from a TRNG, while the decryption gets the random bits by decrypting them from the input file.

This early benchmark servers is an evidence that the 2 bits overhead is practically negligible, and that preparing the random bits in advance, to avoid calling a TRNG on demand, will ensure a faster Băhēm even during its encryption (since calling the TRNG is the only aspect that makes the encryption differ from the decryption).

5 Conclusion

This paper proposed Băhēm with the following properties:

Secure. Băhēm is proven that no cryptanalysis can degrade its security below H(**k**) bits.

Fast. Requires only 4 XORs per encryption or decryption alike. Highly parallelisable as the encryption, or decryption, of any bit is independent of other bits.

A single-threaded early prototype (Alyal) outperformed OpenSSL's ChaCha20 when decrypting files, despite Băhēm's 2 bits overhead, which proves that such overhead is negligible in practice.

While Alyal underperformed during the encryption for its use of a TRNG, optimising it is trivial by preparing the TRNG in advance. This is confirmed by the currently fast decryption speed, which only differs from the encryption by the fact that it does not pull bits from the TRNG.

Simple. Băhēm's simplicity implies into fewer expected number of implementation bugs, and therefore higher practical security.

Future work may include developing a multi-threaded implementation with TRNG optimisations.

References

[1] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 212–219, New York, NY, USA, 1996. Association for Computing Machinery.

[2] Daniel Bernstein. Chacha, a variant of salsa20. 01 2008.

[3] Joan Daemen and Vincent Rijmen. AES Proposal: Rijndael, 1999.

[4] Jean-Philippe Aumasson, Simon Fischer, Shahram Khazaei, Willi Meier, and Christian Rechberger. New features of latin dances: Analysis of salsa, chacha, and rumba. Cryptology ePrint Archive, Report 2007/472, 2007. https://ia.cr/2007/472.