


Băhēm

A Symmetric Cipher with Provable 128-bit Security

M. Rajululkahf ¹

May 6, 2022

Overview

This paper proposes a symmetric cipher, which I name *Băhēm*, with the following properties:

Practical: Requires pre-sharing only a 128-bits key.

Provably secure: No cryptanalysis can degrade its security below $\min[H(\mathbf{m}), H(\mathbf{k})]$ bits of entropy, even under Grover’s algorithm [1] or even if it turned out that $P = NP$.

Post-apocalypse: Should computers cease to exist or function, its encryption and decryption can be done by hand with a pen, a paper and some fair dies to roll, with relative ease. This is thanks to it requiring only one addition per-session, and two additions and one bitwise exclusive-or operation (XOR) per-block.

Fast: Runs fast on common hardware. Its early single-threaded implementation achieved similar run-time speeds to OpenSSL’s ChaCha20 [2]. Faster speed is easily doable with parallelism and better true random number generator (TRNG) optimisations.

This comes at a usually-negligible cost of having a $128 + 2|\mathbf{m}|$ -bit ciphertext output for a $|\mathbf{m}|$ -bit cleartext input; since space is usually not a bottleneck for most applications.

Băhēm is the only symmetric cipher to-date that is *practical* and *provably secure*. Other ciphers are only one of them, but not both. For example, the one-time pad (OTP) is provably secure but usually impractical, as it requires pre-sharing a key that is as large as the message to encrypt. On the other hand, state of art ciphers, such as ChaCha20 [2] or AES [3], are practical but not provably secure.

¹Author’s e-mail address: {last name}@pm.me

EF91FF90DF73A9D76E4841C76D5CB15E7E909C309B307BED15BFB4E1183B6B9903FA78447E87F166F93B002803B99C0C72C479C253E3D7A5D6BDF320DC0EDBDA

This work is licensed under a Creative Commons “Attribution 4.0 International” license.



Declarations

All data used in this study is included in this paper. The latest version of this paper can be found here², and the latest version of the implementation can be found here³.

Notation

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

$\mathbf{x} + \mathbf{y} \bmod 2^{128}$: Unsigned 128-bit addition.

$\text{random}(128)$: 128 bits generated by a TRNG.

\mathbf{k} : 128-bit pre-shared secret key. Must seem random and uniformly distributed with large enough $H(\mathbf{k})$. Ideally, $\mathbf{k} = \text{random}(128)$.

\mathbf{m} : A cleartext message of $|\mathbf{m}|$ many bits.

$\lceil \frac{|\mathbf{m}|}{128} \rceil$: Number of 128-bit blocks in cleartext \mathbf{m} .

\mathbf{m}_b : The b^{th} 128-bit block from \mathbf{m} . In other words: $\mathbf{m}_0 \parallel \mathbf{m}_1 \parallel \dots \parallel \mathbf{m}_{\lceil \frac{|\mathbf{m}|}{128} \rceil} = \mathbf{m}$.

$\mathbf{s} = \text{random}(128)$: Session key.

$\mathbf{p}_b = \text{random}(128)$: Pad key of the b^{th} block.

$\hat{\mathbf{s}}, \hat{\mathbf{p}}_b, \hat{\mathbf{m}}_b$: Encrypted \mathbf{s} , \mathbf{p}_b and \mathbf{m}_b , respectively.

Contents

1	Introduction	2
2	Proposed Algorithm: Băhēm	2
3	Security Analysis	2
4	Benchmark	3
5	Conclusions	4
A	Implementation Examples	4
A.1	Pens, Papers and Fair Dies	4
A.2	C Functions	4
A.3	A File Encryption Tool	5
A.3.1	Installation	5
A.3.2	Usage	5

²<https://codeberg.org/rajululkahf/baheem>

³<https://codeberg.org/rajululkahf/alyal>

1 Introduction

Today's state of art symmetric ciphers, such as ChaCha20 or AES, are attractive for their practicality (requiring only a small, say, 256-bit key to pre-share), and for their probable security which is supported by the failure of the many attempts to break them so far.

However, it remains unknown whether they are actually secure. It is even known if it is possible for them to be secure at all, since it remains unknown whether $P \neq NP$. This uncertainty about their security is quite risky, as encrypted sensitive data is often exposed over public networks. Should such ciphers be discovered to be broken, one can decrypt the previously exposed sensitive data.

On the other hand, Shannon's OTP is more than just provably secure, as it satisfies the higher criteria of having *perfect secrecy*; that is, no cryptanalysis can degrade its security below $H(\mathbf{m})$ many bits.

However, the OTP is usually impractical as it requires the communicating parties to exchange keys that are as large as the size of the messages that they will be exchanging in the future. This often implies the necessity to exchange many gigabytes, or terabytes, of truly random bits in advance of the communication, which is too difficult to satisfy with most application scenarios.

Due to OTP's impracticality, most applications choose to rather adopt the *practically* secure (but not provably) ciphers like ChaCha20 or AES, in order to avoid the unscalable constraint of having to exchange large random bits in advance of their communication.

2 Proposed Algorithm: Bähēm

Algorithms 1 and 2 show Bähēm's encryption and decryption by which the process is repeated over every 128-bit blocks of \mathbf{m} : $\mathbf{m}_0, \mathbf{m}_1, \dots, \mathbf{m}_{\lceil \frac{|\mathbf{m}|}{128} \rceil}$.

Algorithm 1: Bähēm encryption

input : $\mathbf{k}, \mathbf{m}_0, \mathbf{m}_1, \dots$
output: $\hat{\mathbf{s}}, (\hat{\mathbf{p}}_0, \hat{\mathbf{m}}_0), (\hat{\mathbf{p}}_1, \hat{\mathbf{m}}_1), \dots$

```

s ← random(128)
ŝ ← s + k mod 2128
for b ∈ (0, 1, ..., ⌈|m|/128⌉ - 1) do
    pb ← random(128)
    p̂b ← pb + k mod 2128
    m̂b ← mb ⊕ (pb + s mod 2128)

```

Algorithm 2: Bähēm decryption

input : $\mathbf{k}, \hat{\mathbf{s}}, (\hat{\mathbf{p}}_0, \hat{\mathbf{m}}_0), (\hat{\mathbf{p}}_1, \hat{\mathbf{m}}_1), \dots$
output: $\mathbf{m}_0, \mathbf{m}_1, \dots$

```

s ← ŝ - k mod 2128
for b ∈ (0, 1, ..., ⌈|m|/128⌉ - 1) do
    pb ← p̂b - k mod 2128
    mb ← m̂b ⊕ (pb + s mod 2128)

```

3 Security Analysis

The Bähēm encryption is essentially the XOR cryptosystem:

$$\hat{\mathbf{m}}_b \leftarrow \mathbf{m}_b \oplus \underbrace{(\mathbf{p}_b + \mathbf{s} \bmod 2^{128})}_{\text{One-time encryption pad}}$$

It trivially follows from Shannon's perfect secrecy proof of the OTP [4] that Bähēm is secure if its encryption pad maintains its security.

To simplify the analysis, suppose that the size of a block in Bähēm is 3 bits only, and that the cleartext block \mathbf{m}_b is known to the adversary, which implies that the adversary can trivially know that:

$$\mathbf{p}_b + \mathbf{s} \bmod 2^3 = \hat{\mathbf{m}}_b \oplus \mathbf{m}_b$$

in addition to adversary's knowledge of the public variables $\hat{\mathbf{s}}$ and $\hat{\mathbf{p}}_b$. More specifically, suppose that the adversary found that:

$$\begin{aligned} 0 &= \hat{\mathbf{s}} = \mathbf{s} + \mathbf{k} \bmod 2^3 \\ 3 &= \hat{\mathbf{p}}_b = \mathbf{p}_b + \mathbf{k} \bmod 2^3 \\ 5 &= \hat{\mathbf{m}}_b \oplus \mathbf{m}_b = \mathbf{p}_b + \mathbf{s} \bmod 2^3 \end{aligned}$$

Then, the question is: will this information reduce the space from which the key \mathbf{k} is chosen from? In other words, what are the possible values of \mathbf{k} that can lead to the outputs 0, 3 and 5 above? Table 1 visualises this.

As shown in table 1, the total number of horizontal, or vertical, intersections that simultaneously cross all of the outputs 0, 3 and 5, remain 2^3 . Meaning, the total number of values of \mathbf{k} that could lead to the outputs remains 2^3 .

This 3-bit example can be trivially extended by induction to show that the same conclusions hold even with a 128-bit unsigned addition and any other output numbers than 0, 3 and 5.

Therefore, we can conclude that adversary's knowledge of the public variables $\hat{\mathbf{s}}$, $\hat{\mathbf{p}}_b$, $\hat{\mathbf{m}}_b$ and the cleartext \mathbf{m}_b , which leads to deducing $\mathbf{p}_b + \mathbf{s} \bmod 2^{128}$,

		\mathcal{Y}							
		0	1	2	3	4	5	6	7
\mathcal{X}	0	0	1	2	3	4	5	6	7
	1	1	2	3	4	5	6	7	0
	2	2	3	4	5	6	7	0	1
	3	3	4	5	6	7	0	1	2
	4	4	5	6	7	0	1	2	3
	5	5	6	7	0	1	2	3	4
	6	6	7	0	1	2	3	4	5
	7	7	0	1	2	3	4	5	6

Table 1: Exhaustive unsigned 3-bit addition. For a given output $\mathbf{x} + \mathbf{y} \bmod 2^3$, there are 2^3 many possible input values of $(\mathbf{x}, \mathbf{y}) \in \mathcal{X} \times \mathcal{Y}$ that map to $\mathbf{x} + \mathbf{y} \bmod 2^3$.

can not reduce the space from which \mathbf{k} , \mathbf{s} and \mathbf{p}_b are sampled.

If \mathbf{k} , \mathbf{s} and \mathbf{p}_b are generated by a TRNG, then any of the 2^{128} many possibilities are equally likely to correspond to the actual values of \mathbf{k} , \mathbf{s} and \mathbf{p}_b . In other words:

$$H(\mathbf{k}, \mathbf{s}, \mathbf{p}_b | \hat{\mathbf{s}}, \hat{\mathbf{p}}_b, \hat{\mathbf{m}}_b, \mathbf{m}_b) = 128$$

However, since \mathbf{k} could be derived from a password, such that it looks random, but with an entropy $H(\mathbf{k}) \leq 128$, and since finding any of the numbers \mathbf{k} , \mathbf{s} and \mathbf{p}_b deterministically leads to finding the others, therefore it follows that:

$$H(\mathbf{k}, \mathbf{s}, \mathbf{p}_b | \hat{\mathbf{s}}, \hat{\mathbf{p}}_b, \hat{\mathbf{m}}_b, \mathbf{m}_b) = H(\mathbf{k})$$

The numbers \mathbf{s} and \mathbf{p}_b are generated by a TRNG by definition, therefore the weakest element in the chain can only be \mathbf{k} .

Since the public variables $\hat{\mathbf{s}}$, $\hat{\mathbf{p}}_b$ and $\hat{\mathbf{m}}_b$, and the cleartext \mathbf{m}_b are exhaustively all of the outputs of Bähēm that can be accessible to an adversary, and since they can not reduce Bähēm’s private variables’ space below $H(\mathbf{k})$, therefore no cryptanalysis can reduce their entropy below $H(\mathbf{k})$.

Lemma 1 (Secure private values).

$$H(\mathbf{k}, \mathbf{s}, \mathbf{p}_b | \hat{\mathbf{s}}, \hat{\mathbf{p}}_b, \hat{\mathbf{m}}_b, \mathbf{m}_b) = H(\mathbf{k})$$

It is trivially implied from lemma 1 that, since the private values \mathbf{s} and \mathbf{p}_b maintain an entropy of $H(\mathbf{k})$, so does their 128-bit summation $\mathbf{s} + \mathbf{p}_b \bmod 2^{128}$, which is Bähēm’s XOR encryption pad. Therefore, Bähēm’s encryption pad has to be secure as well.

Lemma 2 (Secure encryption pad).

$$H(\mathbf{s} + \mathbf{p}_b \bmod 2^{128} | \hat{\mathbf{s}}, \hat{\mathbf{p}}_b, \hat{\mathbf{m}}_b) = \min[H(\mathbf{m}_b), H(\mathbf{k})]$$

Since Bähēm is an XOR cryptosystem, and since its encryption pad is $H(\mathbf{k})$ -bits secure (lemma 2), therefore it necessarily follows by Shannon’s perfect secrecy [4] that Bähēm’s encryption is either $H(\mathbf{k})$ -bits secure, or $H(\mathbf{m}_b)$ -bits secure, whichever is smaller.

Theorem 1 (Secure encryption).

$$H(\mathbf{m}_b | \hat{\mathbf{s}}, \hat{\mathbf{p}}_b, \hat{\mathbf{m}}_b) = \min[H(\mathbf{m}_b), H(\mathbf{k})]$$

Note: Bähēm does not aim at achieving perfect secrecy, as perfect secrecy requires a usually-impractical key that is as long as the length $|\mathbf{m}|$ of the cleartext message \mathbf{m} itself, with an unnecessarily too large upper security bound that is worth $|\mathbf{m}|$ many bits of entropy.

Bähēm rather aims at achieving a security that is worth $\min[H(\mathbf{m}_b), H(\mathbf{k})]$ entropy bits. Unlike the OTP, this is *practically* secure and only requires pre-sharing a small 128-bit key.

Shannon’s proof of perfect secrecy of the OTP is cited only for its relevance as an XOR cryptosystem.

4 Benchmark

This is a benchmark that was performed on a computer with a 3.4GHz Intel Core i5-3570K CPU, 32GB RAM, 7200 RPM hard disks, Linux 5.17.4-gentoo-x86-64, OpenSSL 1.1.1n and Alyal v3.

	OpenSSL ChaCha20	Alyal Bähēm	
		/dev/random	file.rand
Encrypt	0.90 secs	2.58 secs	1.38 secs
500MB	1.06 secs	2.60 secs	1.35 secs
	1.04 secs	2.58 secs	1.35 secs
Decrypt	0.89 secs	0.82 secs	
500MB	1.12 secs	0.87 secs	
	1.06 secs	0.82 secs	

Table 2: Wall-clock run-time comparison between OpenSSL’s ChaCha20, and Alyal’s Bähēm implementation with two sources as the TRNG: `/dev/random` and `file.rand`; the latter is simply `/dev/random` that was prepared in advance.

Table 2 shows that, while the early Bähēm prototype, Alyal, has a faster decryption run-time than OpenSSL’s ChaCha20, it has a slower encryption run-time. However:

1. The differences in run-time are insignificant for most applications, which proves Bähēm’s practical utility in the real world.
2. Bähēm’s provable security should arguably justify waiting the extra seconds, or fractions of seconds in case the TRNG is prepared in advance, for the 500MB data, specially that many user applications involve encrypting much smaller data sizes with unnoticeable time difference
3. Preparing the random bits in advance significantly reduces the encryption time as shown with the `file.rand` case in table 2, and can be optimised further should it be prepared in memory.
4. Alyal is currently single-threaded despite Bähēm’s capacity for high parallelism as all blocks are independent. This gives room for future versions to be significantly faster.

5 Conclusions

This paper proposed Bähēm with the following properties:

Provably secure: No cryptanalysis can degrade its security below $\min[H(\mathbf{m}), H(\mathbf{k})]$ bits.

Fast: Requires only three additions (one per-session, two per-block) and a single XOR per encryption or decryption alike.

Highly parallelisable as the encryption, or decryption, of any bit is independent of other bits.

Bähēm’s single-threaded prototype, Alyal, outperformed OpenSSL’s ChaCha20 when decrypting files, despite Bähēm’s 1-bit overhead, which demonstrates that such overhead is negligible in practice.

While the prototype has a slower encryption run-time due to its use of a TRNG, optimising it is trivial by preparing the TRNG in advance.

Simple: Bähēm’s simplicity implies fewer expected number of implementation bugs, and therefore higher practical security.

Another interesting advantage of this simplicity is that it allows Bähēm to be used with a mere pen and a paper should one lack a computer, such as the case with post-apocalyptic scenarios.

For example, in a post-apocalyptic scenario, one can generate the random numbers \mathbf{p}_b and \mathbf{q}_b by rolling dies enough number of times until adequate entropy is obtained, and then using a pen

and a paper to calculate the ciphertext as per algorithm 1.

Since Bähēm does not require repeating rounds over and over, Bähēm is significantly simpler to perform using a pen and a paper than, say, ChaCha20, AES [3], etc, which require many repeated rounds that make it too tedious for a human to perform by the pen and paper method.

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] C. E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, 1949.

A Implementation Examples

A.1 Pens, Papers and Fair Dies

A.2 C Functions

Listings 1 and 2 show example C functions for encrypting and decrypting session keys.

Listings 3 and 4 show the same but for encrypting and decrypting cleartext and ciphertext blocks, respectively.

In these examples, all encryptions and decryptions happen in-place whenever possible, so the caller does not have to allocate separate memory for the output. The only exception is listing 1, where the unencrypted session key is required to encrypt the subsequent cleartext blocks. Also, since 128-bit wide CPU instructions are not common, the examples operate in 64-bit basis, each time with a different 64-bit part of the pre-shared and session keys.

Listing 1: Session key encryption function example.

```
void baheem_session_enc(
    uint64_t *k,      /* pre-shared key */
    uint64_t *s,      /* session key */
    uint64_t *s_enc /* encrypted s */)
```

```

) {
    s_enc[0] = s[0] + k[0];
    s_enc[1] = s[1] + k[1];
}

```

Listing 2: Session key decryption function example.

```

void baheem_session_dec(
    uint64_t *k, /* pre-shared key */
    uint64_t *s /* session key */
) {
    s[0] -= k[0];
    s[1] -= k[1];
}

```

Listing 3: Block encryption function example.

```

void baheem_block_enc(
    uint64_t *k, /* pre-shared key */
    uint64_t *s, /* session key */
    uint64_t *p, /* pad keys */
    uint64_t *m, /* message */
    size_t len /* length of m and p */
) {
    size_t i;
    for (i = 0; i < len; i += 2) {
        m[i] ^= p[i] + s[0];
        m[i+1] ^= p[i+1] + s[1];
        p[i] += k[0];
        p[i+1] += k[1];
    }
}

```

Listing 4: Block decryption function example.

```

void baheem_block_dec(
    uint64_t *k, /* pre-shared key */
    uint64_t *s, /* session key */
    uint64_t *p, /* pad keys */
    uint64_t *m, /* message */
    size_t len /* length of m and p */
) {
    size_t i;
    for (i = 0; i < len; i += 2) {
        p[i] -= k[0];
        p[i+1] -= k[1];
        m[i] ^= p[i] + s[0];
        m[i+1] ^= p[i+1] + s[1];
    }
}

```

A.3 A File Encryption Tool

Alyal is a single-threaded implementation to demonstrate Bāhēm’s practical utility with real-world scenarios. Internally, Alyal uses the functions in listings 1 to 4.

A.3.1 Installation

```

git clone \
    https://codeberg.org/rajululkahf/alyal
cd alyal
make
make test

```

A.3.2 Usage

```

alyal (enc|dec) IN OUT [TRNG]
alyal help

```

To encrypt a cleartext file **a** and save it as file **b**:

```

alyal enc a b

```

To decrypt the latter back to its cleartext form and save it as file **c**:

```

alyal dec b c

```