bcrypt を実装してみた。

bcrypt の特徴、
何故か分かるか？

- 72 文字までしかハッシュ化できない
- 「暗号化」処理がボトルネックで重い
- 入力した文字列を暗号化しているわけではない

bcrypt

を実装してみた。

- 72 文字までしかハッシュ化できない
- 「暗号化」処理がボトルネックで重い
- 入力した文字列を暗号化しているわけではない

bcrypt とは？

暗号学的ハッシュ関数

bcrypt とは？

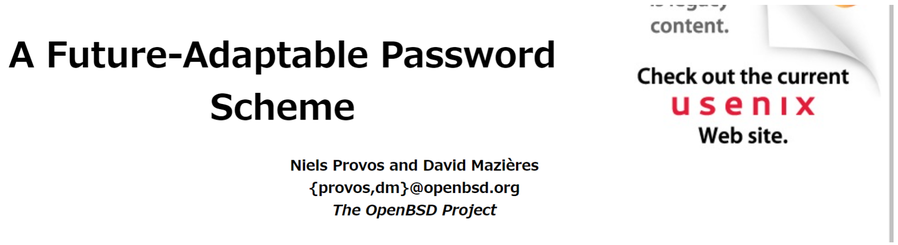
bcrypt はパスワードハッシュ関数ともよく言われる

暗号学的ハッシュ関数

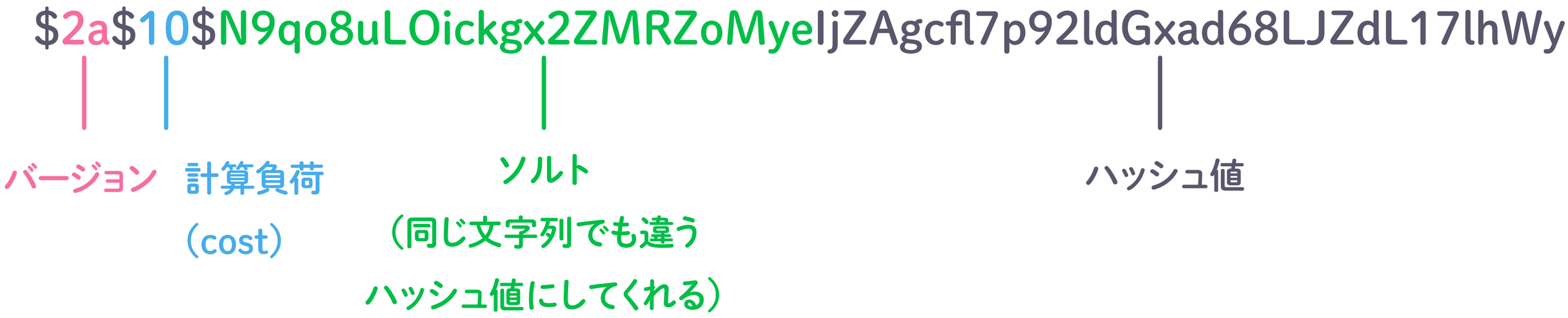
bcrypt の特徴

紅の魔術工房
Atelier kurenaif

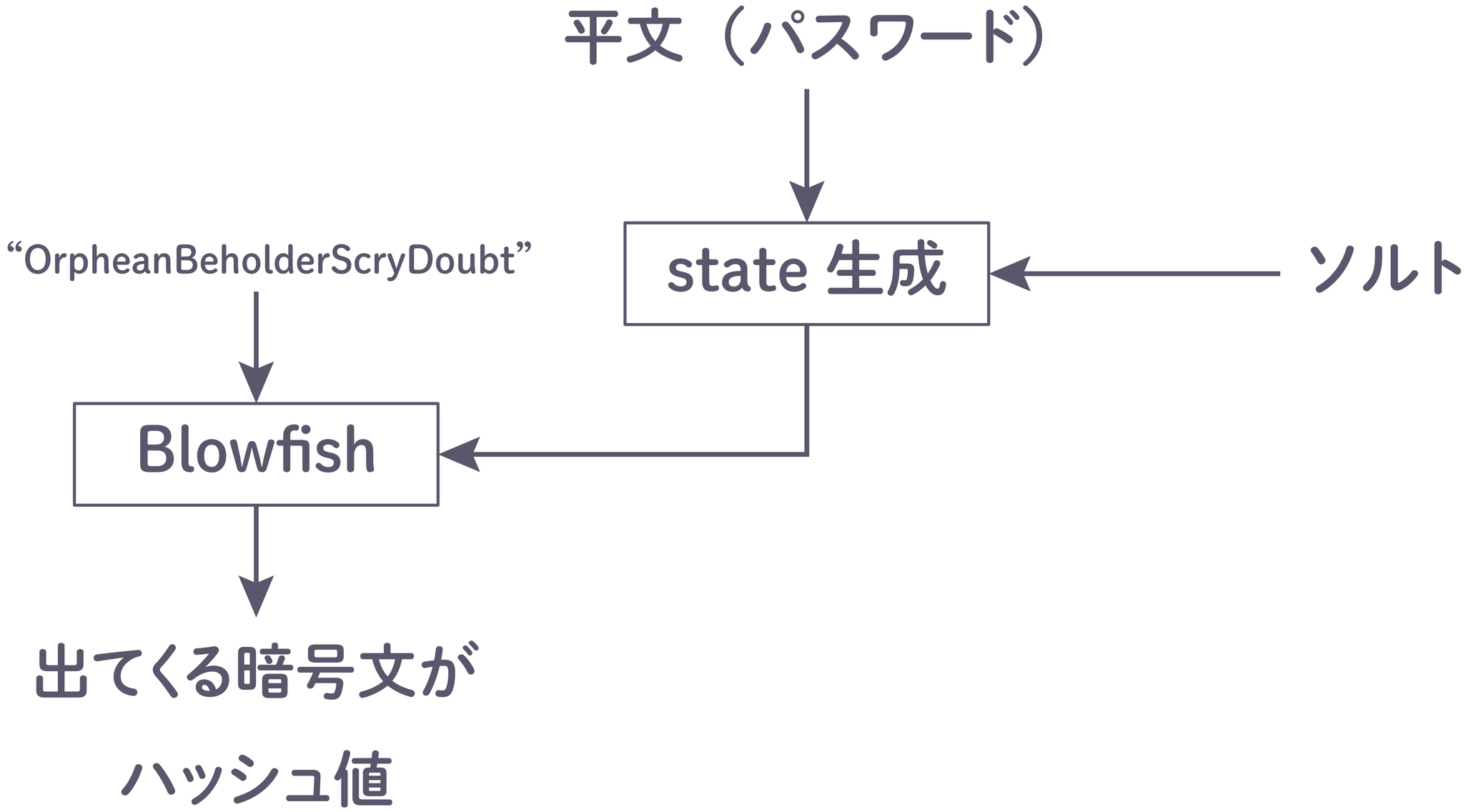
- ソルトが必須（同じパスワードでも異なる値になる）
- 計算負荷を調整できる
- 72bytes までしかハッシュ化できない
- 入力：バージョン、コスト、ソルト、パスワード
- 出力：バージョン、コスト、ソルト、ハッシュ値



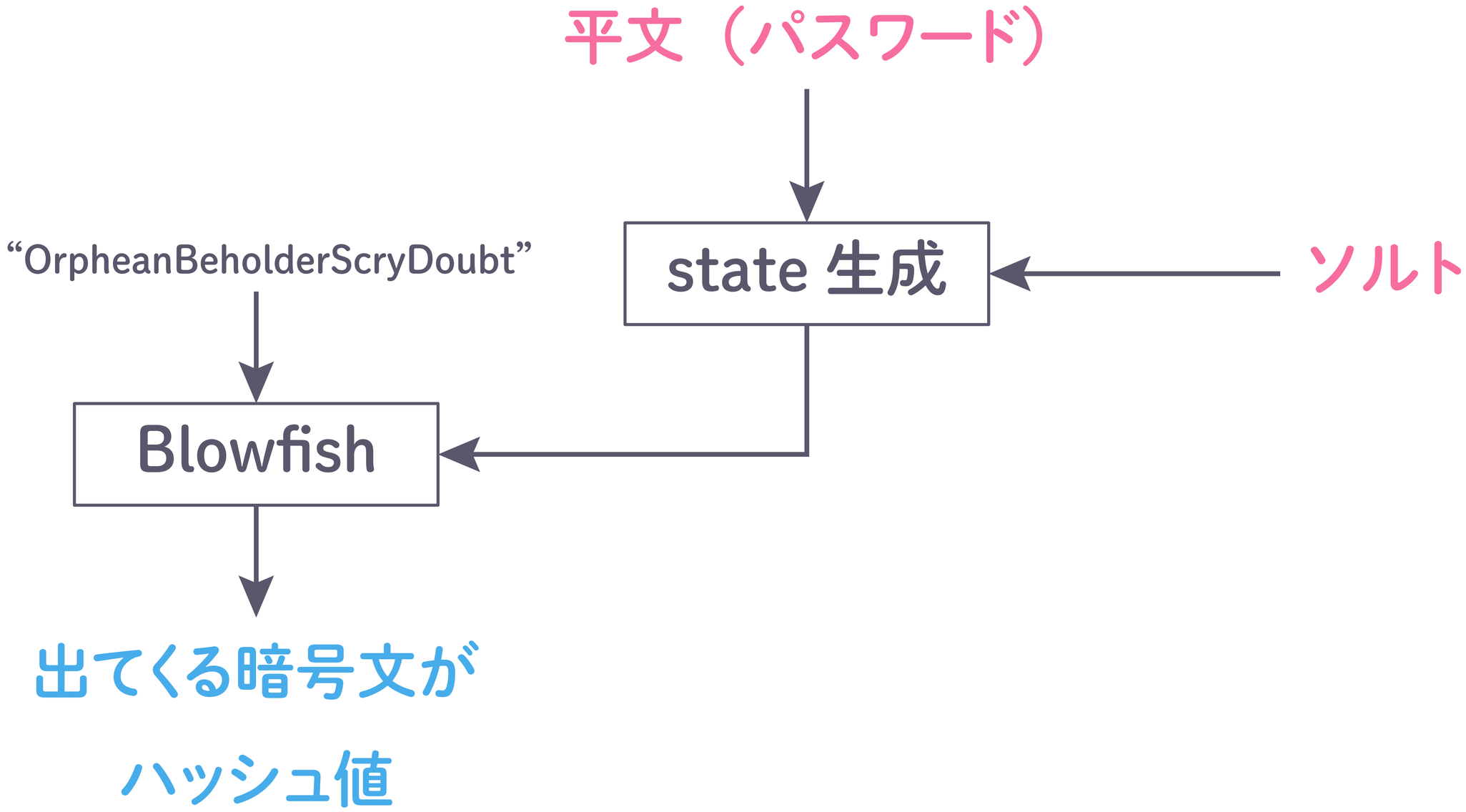
bcrypt の特徴


$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy
バージョン
計算負荷
(cost)
ソルト
(同じ文字列でも違う
ハッシュ値にしてくれる)
ハッシュ値

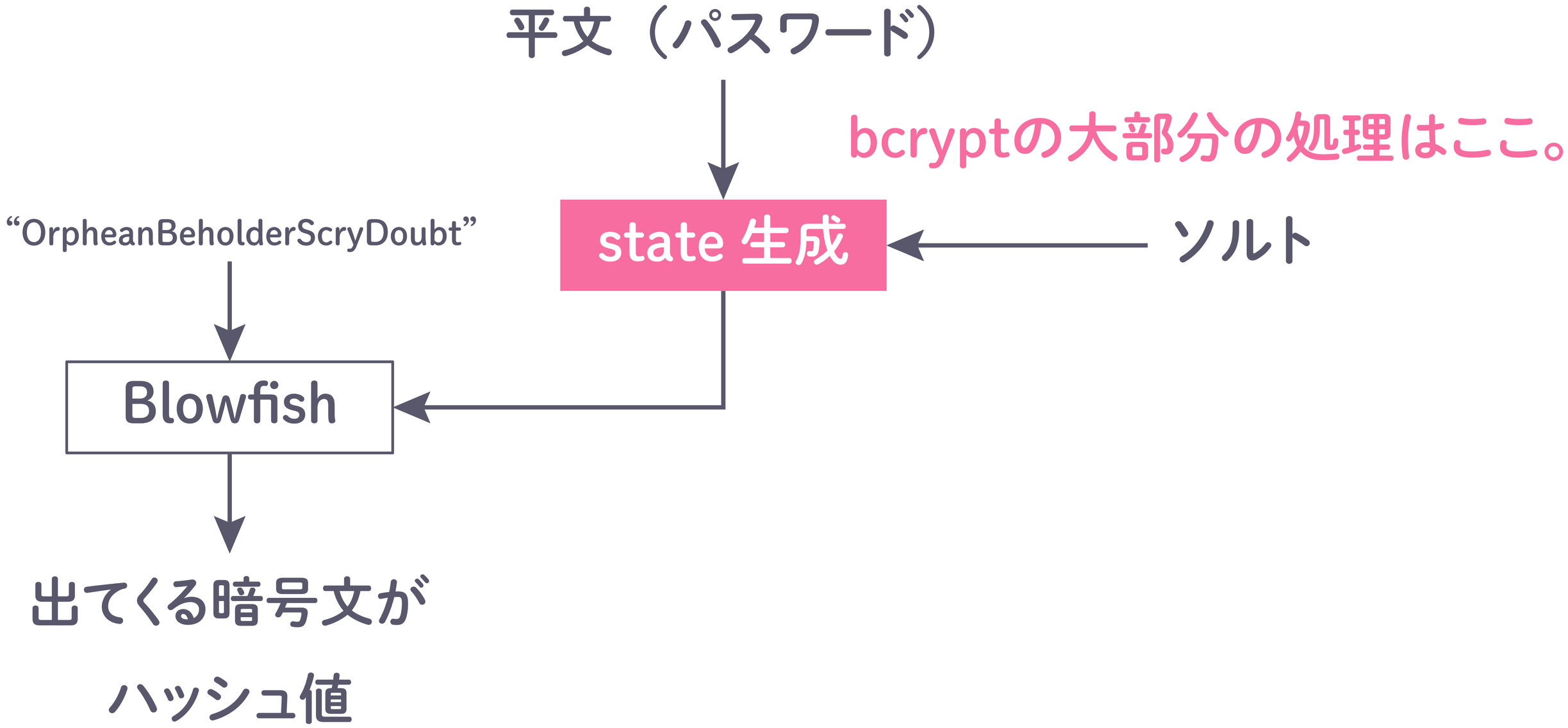
bcrypt のアルゴリズムの概要



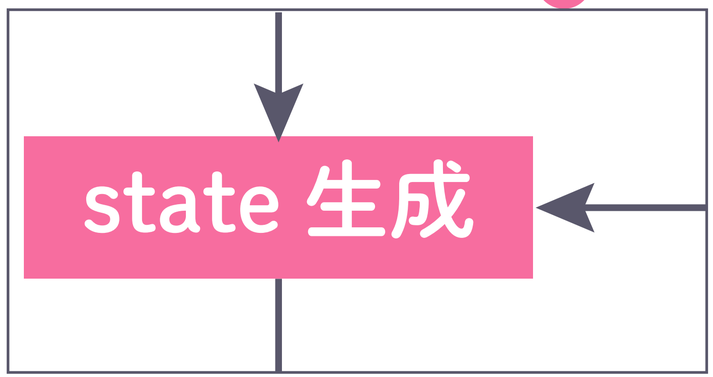
bcrypt のアルゴリズムの概要



bcrypt のアルゴリズムの概要



state って何？



・s0, s1, s2, s3, p からなる配列

・Blowfish は「平文」と「state」で暗号化を行う。

・鍵に近い存在。

```
#[derive(Debug,PartialEq)]
pub struct State {
    pub s0: [u32; 256],
    pub s1: [u32; 256],
    pub s2: [u32; 256],
    pub s3: [u32; 256],
    pub p: [u32; 18],
}
```

state の作り方

- expand key と呼ばれる関数を利用する。

```
Function EksBlowfishSetup
   Input:
      cost:     Number (4..31)                  log2(Iterations)。例えば 12 ==> 2^12 = 4,096
回繰り返す
      salt:     array of Bytes (16 bytes)       ランダムソルト
      password: array of Bytes (1..72 bytes)    UTF-8エンコードされたパスワード
   Output:
      state:    opaque BlowFish state structure

   state ← InitialState()
   state ← ExpandKey(state, salt, password)
   repeat (2^cost)
      state ← ExpandKey(state, 0, password)
      state ← ExpandKey(state, 0, salt)
```

https://ja.wikipedia.org/wiki/Bcrypt

state の作り方

- expand key と呼ばれる関数を利用する。

```
Function EksBlowfishSetup
   Input:
      cost:     Number (4..31)                  log2(Iterations)。例えば 12 ==> 2^12 = 4,096
回繰り返す
      salt:     array of Bytes (16 bytes)       ランダムソルト
      password: array of Bytes (1..72 bytes)    UTF-8エンコードされたパスワード
   Output:
      state:    opaque BlowFish state structure

   state ← InitialState()
   state ← ExpandKey(state, salt, password)
   repeat (2^cost)
      state ← ExpandKey(state, 0, password)
      state ← ExpandKey(state, 0, salt)
```

bcrypt で

ここの実行に一番時間がかかる

https://ja.wikipedia.org/wiki/Bcrypt

expand key?

```
Function ExpandKey(state, salt, password)
   Input:
      state:    Opaque BlowFish state structure     P配列 と S-box のエントリーを含む
      salt:     array of Bytes (16 bytes)           ランダムソルト
      password: array of Bytes (1..72 bytes)        UTF-8エンコードされたパスワード
   Output:
      state:    opaque BlowFish state structure

   //パスワードを状態の内部にあるP-arrayに混ぜていく
   for n ← 1 to 18 do
      Pn ← Pn xor password[32(n-1)..32n-1] //パスワードが循環しているように扱う

   //ソルトの下位8バイトを使いstateの暗号化を行い、8バイトの結果を P1|P2 に格納する
   block ← Encrypt(state, salt[0..63])
   P1 ← block[0..31]  //下位32ビット
   P2 ← block[32..63] //上位32ビット

   //ソルトを用いて繰り返し状態の暗号化を行い、Pの配列の残りの部分に格納していく
   for n ← 2 to 9 do
      block ← Encrypt(state, block xor salt[64(n-1)..64n-1]) //現在のkey scheduleと循環するソ
ルトを用いて暗号化
      P2n-1 ← block[0..31] //下位32ビット
      P2n ← block[32..63]  //上位32ビット

   //暗号化された状態を、状態内部のS-boxに混ぜていく
   for i ← 1 to 4 do
      for n ← 0 to 127 do
         block ← Encrypt(state, block xor salt[64(n-1)..64n-1]) //同上
         Si[2n]   ← block[0..31]  //下位32ビット
         Si[2n+1] ← block[32..63]  //上位32ビット
    return state
```

ざっくりやってることは、暗号化して、暗号化結果を
state に入れていくという処理。
実は、state の途中の状態のものも Blowfish をかけてる

https://ja.wikipedia.org/wiki/Bcrypt

expand key を 実装するときの注意事項

紅の魔術工房
Atelier kurenaif

パスワードと salt は、ループしているものとして利用する。

password password password password ...

salt(16bytes) salt(16bytes) salt(16bytes) ...

```rust
pub fn get_next_u32(from: &[u8], pos: &mut usize) -> Option<u32> {
    if *pos >= from.len() {
        return None
    }

    let mut res: u32 = 0;
    for _ in 0..4 {
        res = res << 8 | (from[*pos] as u32);
        *pos += 1;
        if *pos >= from.len() {
            *pos = 0
        }
    }
    Some(res)
}
```

足りない部分は
ループさせる

Encrypt（暗号化）って何してるの？

紅の魔術工房
Atelier kurenaif

- 実は state 生成時にも Blowfish を利用している。
- そろそろ Blowfish の話をしようか

Blowfish?

- 共通鍵暗号の一種（仲間：AES(Rijndael)）
- 「state」と「平文」で暗号化を行う。
- 1993 年生まれ

Description of a new variable-length key, 64-bit block cipher (Blowfish)

Authors　Authors and affiliations

Bruce Schneier

Conference paper
First Online: 08 June 2005

235 Citations　4 Mentions

Part of the Lecture Notes in Computer Science book series (LNCS, volume 809)

https://link.springer.com/chapter/10.1007/3-540-58108-1_24#:~:text=Blowfish%2C%20a%20new%20secret%2Dkey%20block%20cipher%2C%20is%20proposed.&text=The%20block%20size%20is%2064,very%20efficient%20on%20large%20microprocessors

Blowfish の実装

紅の魔術工房
Atelier kurenaif

実は中身はそんなに難しいことはやってない

Divide x into two 32-bit halves: x_L, x_R
For i = 1 to 16:
$x_L = x_L$ XOR P_i
$x_R = F(x_L)$ XOR x_R
Swap x_L and x_R
Swap x_L and x_R (Undo the last swap.)
$x_R = x_R$ XOR P_{17}
$x_L = x_L$ XOR P_{18}
Recombine x_L and x_R

xr: 下位 32bit

xl: 上位 32bit

Divide x_L into four eight-bit quarters: a, b, c, and d
$F(x_L) = ((S_{1,a} + S_{2,b} \bmod 2^{32})$ XOR $S_{3,c}) + S_{4,d} \bmod 2^{32}$

a: 上位 8bit

d: 下位 8bit

Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)
(Sctmeie, 1993)

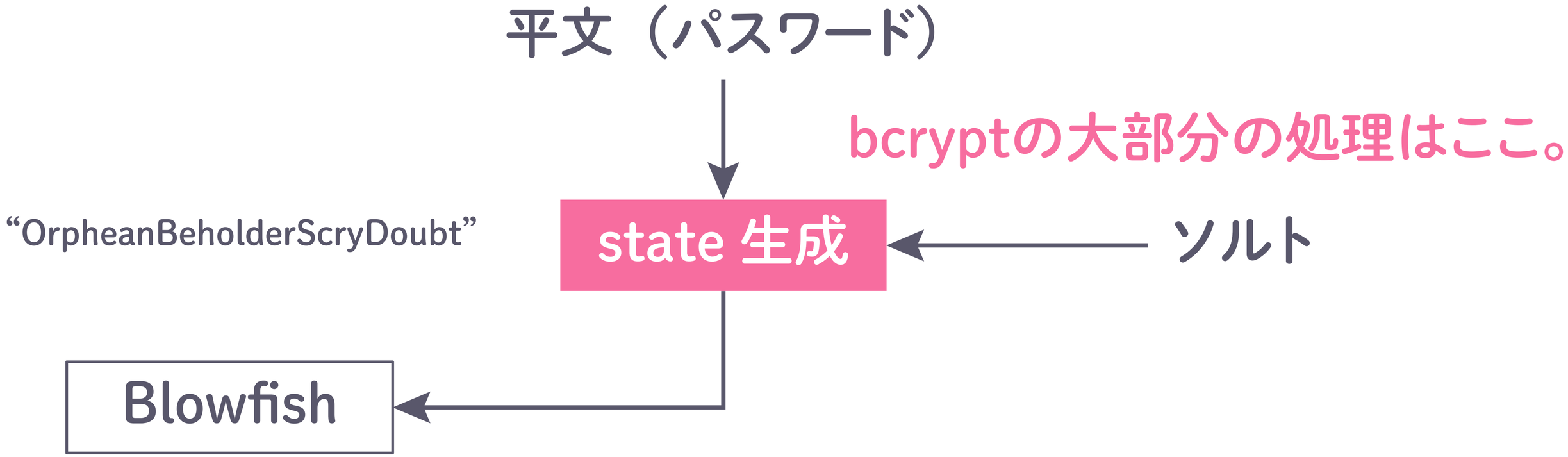
Blowfish の実装

紅の魔術工房
Atelier kurenaif

実は中身はそんなに難しいことはやってない

```rust
// Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish) pp.195
pub fn encrypt(state: &State, data: u64) -> u64 {
    let mut xr = data as u32;
    let mut xl = (data >> 32) as u32;

    for i in 0..16 {
        xl = xl ^ state.p[i];
        xr = f(state, xl) ^ xr;
        swap(&mut xl, &mut xr);
    }
    swap(&mut xl, &mut xr);
    xr = xr ^ state.p[16];
    xl = xl ^ state.p[17];

    (xl as u64) << 32 | (xr as u64)
}

fn f(state: &State, x: u32) -> u32 {
    let a = (x >> 24) as u8;
    let b = (x >> 16) as u8;
    let c = (x >> 8) as u8;
    let d = x as u8;
    ((state.s0[a as usize].wrapping_add(state.s1[b as usize])) ^ state.s2[c as usize]).w
    usize])
}
```

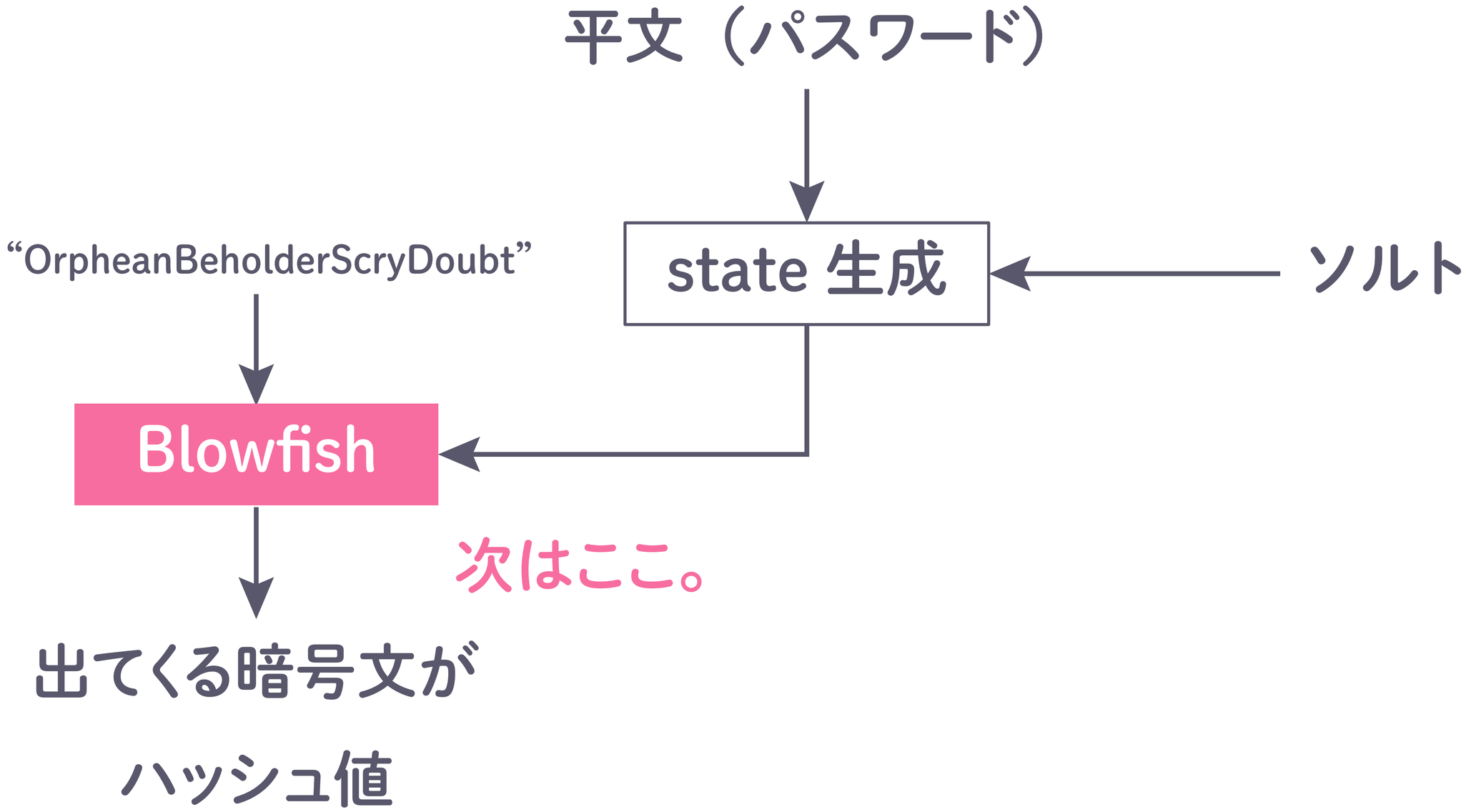
これで state の生成はできた。



出てくる暗号文が

ハッシュ値

“OrpheanBeholderScryDoubt”の暗号化



“OrpheanBeholderScryDoubt”の暗号化

ここでは 64 回暗号化処理を行う。

さっき紹介した Encrypt をそのまま使う。

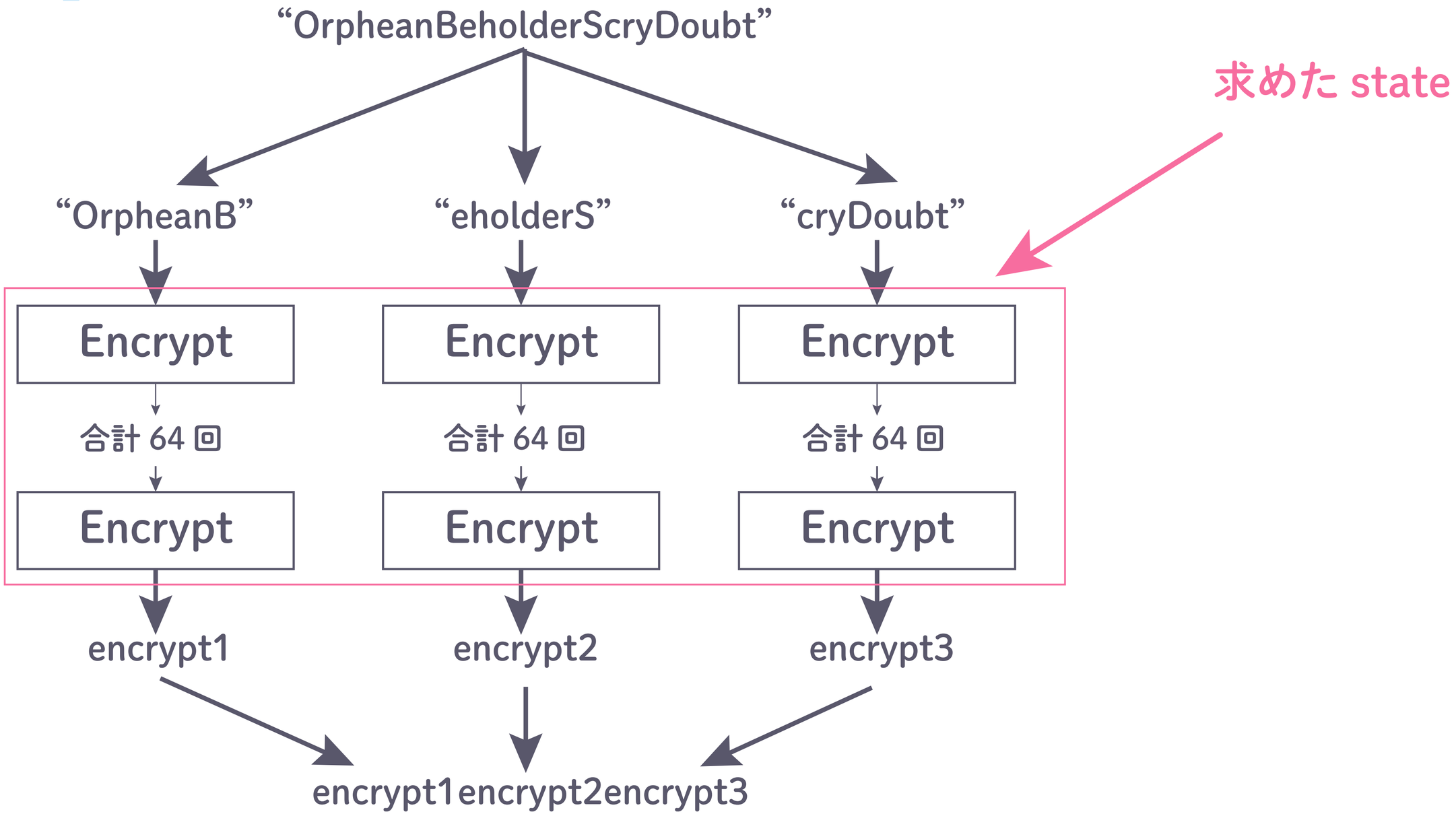
けど、Blowfish は 64bit = 8 文字分しか暗号化できない。

なので、文字列を 3 分割し、

8 文字ずつ暗号化する。

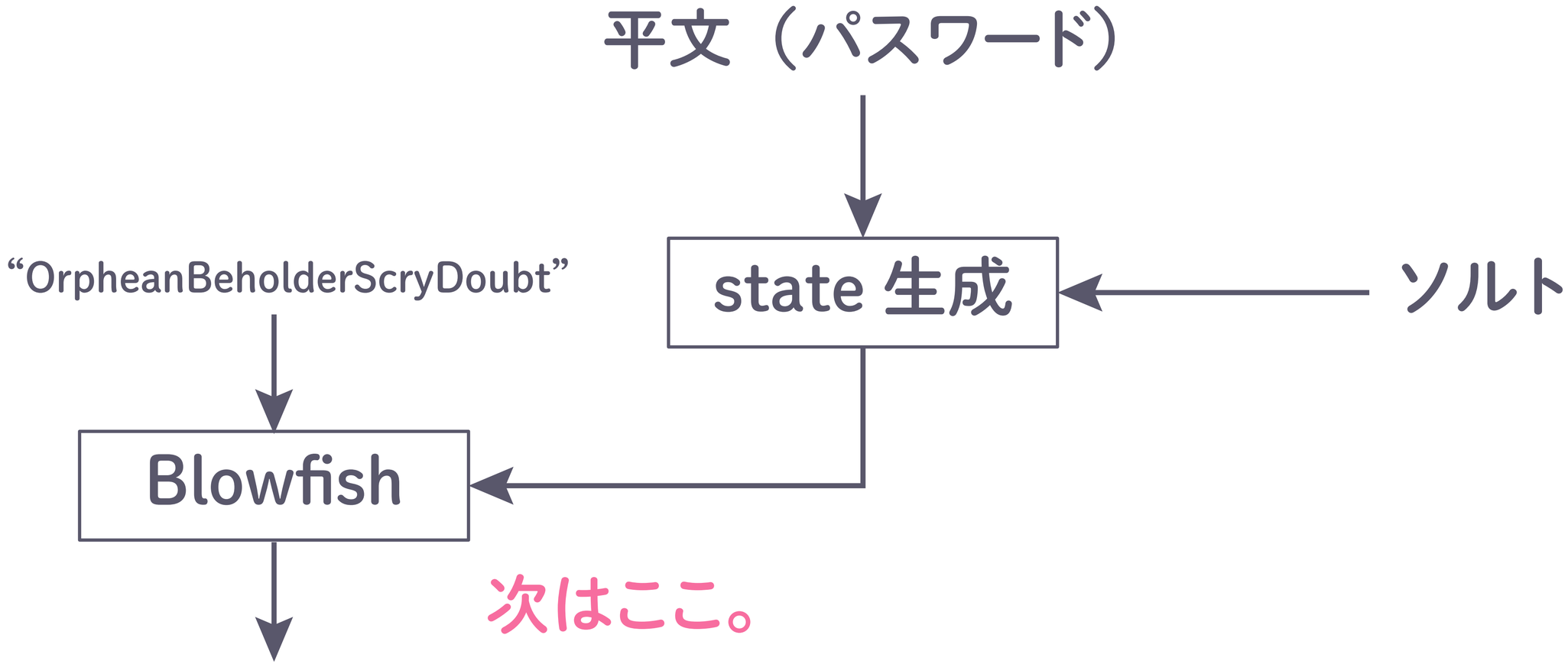
紅の魔術工房
Atelier kurenaif

“OrpheanBeholderScryDoubt” の暗号化



紅の魔術工房
Atelier kurenaif

最後の出力


平文（パスワード）
“OrpheanBeholderScryDoubt”
state 生成
ソルト
Blowfish
次はここ。
出てくる暗号文が
ハッシュ値

最後の出力

紅の魔術工房
Atelier kurenaif

$2a$10$N9qo8uLOickgx2ZMRZoMyeIjZAgcfl7p92ldGxad68LJZdL17lhWy

- $2a : バージョン
- 10 : 計算負荷（cost）
- N9qo8uLOickgx2ZMRZoMye : ソルト（同じ文字列でも違うハッシュ値にしてくれる）
- IjZAgcfl7p92ldGxad68LJZdL17lhWy : ハッシュ値

あくまでも出てくるハッシュ値は バイト列。

どうやってこの文字列を作るの？

base64 の復習

1. 元データ
    - 文字列: "ABCDEFG"
    - 16進表現: 41, 42, 43, 44, 45, 46, 47
    - 2進表現: 0100 0001, 0100 0010, 0100 0011, 0100 0100, 0100 0101, 0100 0110, 0100 0111
2. 6ビットずつに分割
    - 010000 010100 001001 000011 010001 000100 010101 000110 010001 11
3. 2ビット余るので、4ビット分0を追加して6ビットにする
    - 010000 010100 001001 000011 010001 000100 010101 000110 010001 110000

10進	2進	文字	10進	2進	文字	10進	2進	文字	10進	2進	文字
0	000000	A	16	010000	Q	32	100000	g	48	110000	w
1	000001	B	17	010001	R	33	100001	h	49	110001	x
2	000010	C	18	010010	S	34	100010	i	50	110010	y
3	000011	D	19	010011	T	35	100011	j	51	110011	z
4	000100	E	20	010100	U	36	100100	k	52	110100	0
5	000101	F	21	010101	V	37	100101	l	53	110101	1
6	000110	G	22	010110	W	38	100110	m	54	110110	2
7	000111	H	23	010111	X	39	100111	n	55	110111	3
8	001000	I	24	011000	Y	40	101000	o	56	111000	4
9	001001	J	25	011001	Z	41	101001	p	57	111001	5
10	001010	K	26	011010	a	42	101010	q	58	111010	6
11	001011	L	27	011011	b	43	101011	r	59	111011	7
12	001100	M	28	011100	c	44	101100	s	60	111100	8
13	001101	N	29	011101	d	45	101101	t	61	111101	9
14	001110	O	30	011110	e	46	101110	u	62	111110	+
15	001111	P	31	011111	f	47	101111	v	63	111111	/

https://ja.wikipedia.org/wiki/Base64

base64 の復習

rfc に使用文字が定義されている。

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

https://tools.ietf.org/html/rfc4648

base64 の復習

rfc に使用文字が定義されている。

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	- (minus)
12	M	29	d	46	u	63	_ (underline)
13	N	30	e	47	v		
14	O	31	f	48	w		
15	P	32	g	49	x		
16	Q	33	h	50	y	(pad)	=

https://tools.ietf.org/html/rfc4648

bcrypt の符号化に使用する文字の対応

紅の魔術工房
Atelier kurenaif

```
276  static const u_int8_t Base64Code[] =
277  "./ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
278
```

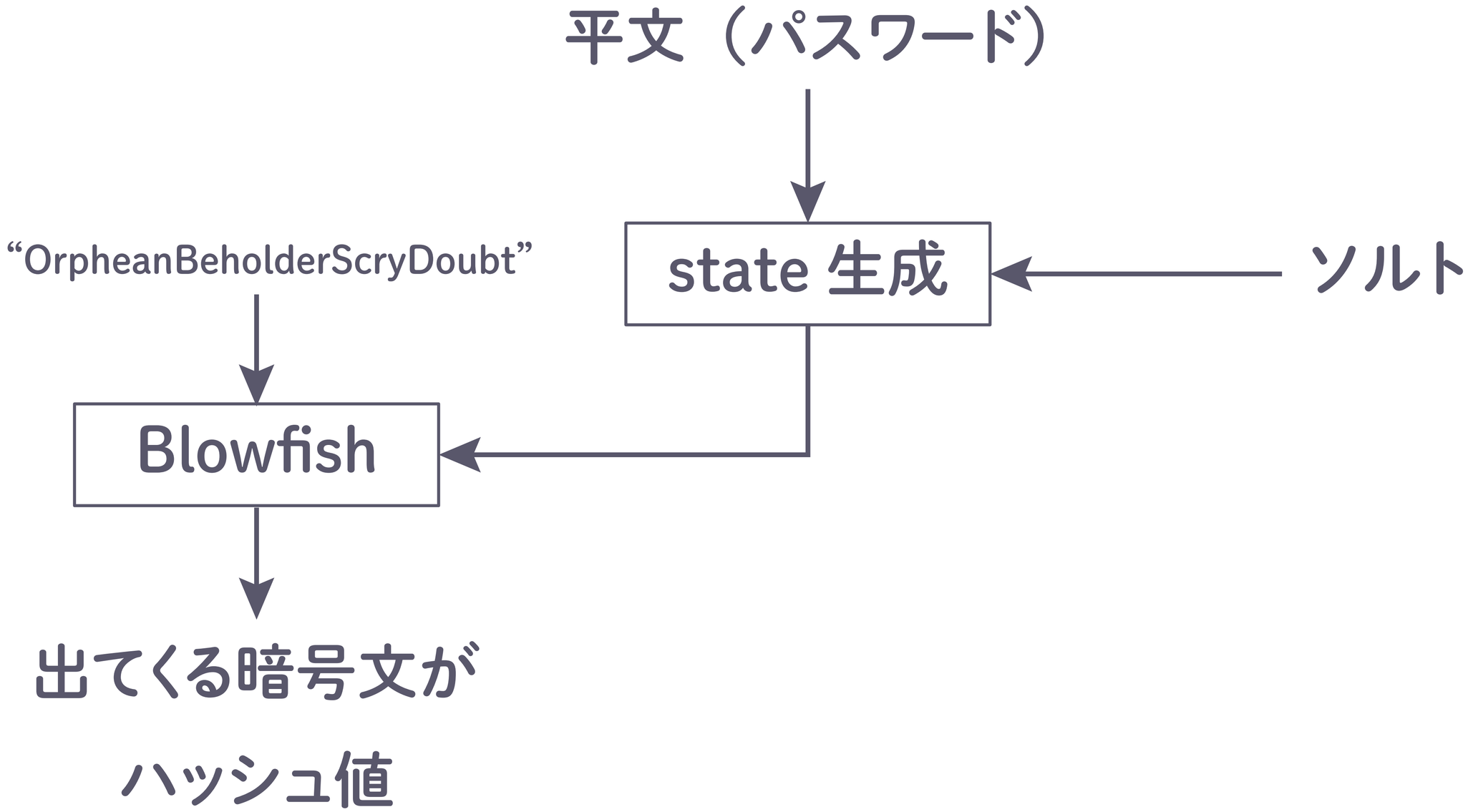
実は RFC4648 の base64 の定義に違反している。

が、みんな base64 を主張しているので実装する際には

注意が必要。

https://github.com/libressl-portable/openbsd/blob/master/src/lib/libc/crypt/bcrypt.c#L276-L277

bcrypt が完成！


平文（パスワード）
state 生成
ソルト
“OrpheanBeholderScryDoubt”
Blowfish
出てくる暗号文が
ハッシュ値

次なる疑問

- bcrypt ってどこが重いの？
- 高速化って本当に難しいの？

自作実装を計測してみよう！

計測してみた。flamegraph

紅の魔術工房
Atelier kurenaif

・ほとんどが eks_blowfish_setup
（つまり state の生成に使ってる）
・ほとんど encrypt の処理に使ってる
・encrypt の中で使ってる f() が重そう

計測してみた：VisualStudio のプロファイラ

紅の魔術工房
Atelier kurenaif

◢ kurebcrypt::bcrypt	33291 (99.08%)	201 (0.60%)	kurebcrypt.exe	IO | カーネル
◢ kurebcrypt::eks_blowfish_setup	33061 (98.39%)	0 (0.00%)	kurebcrypt.exe	IO | カーネル
▷ kurebcrypt::expand_key_without_salt	32516 (96.77%)	32513 (96.76%)	kurebcrypt.exe	カーネル
▷ kurebcrypt::expand_key	539 (1.60%)	511 (1.52%)	kurebcrypt.exe	
[外部コード]	5 (0.01%)	5 (0.01%)	複数のモジュール	カーネル
▷ std::alloc::__default_lib_allocator::__rdl_d...	1 (0.00%)	0 (0.00%)	kurebcrypt.exe	IO

C:¥Users¥kurenaif¥Desktop¥hoge¥bcrypt_training¥src¥lib.rs:226

```
                255     }
                256
     55 (0.16%) 257     for n in 0..128 {
   7985 (23.76%) 258         block = encrypt(&state, block);
                259         let (block_lo, block_hi) = split_u64_to_u32(block);
                260         state.s0[n*2] = block_hi;
                261         state.s0[n*2+1] = block_lo;
                262     }
                263
     50 (0.15%) 264     for n in 0..128 {
   7942 (23.64%) 265         block = encrypt(&state, block);
                266         let (block_lo, block_hi) = split_u64_to_u32(block);
                267         state.s1[n*2] = block_hi;
```

f() を改善してみよう

紅の魔術工房
Atelier kurenaif

f() 周りの実装を golang の実装を参考に変更

```rust
// Description of a New Variable-Length Key, 64-Bit
pub fn encrypt(state: &State, data: u64) -> u64 {
    let mut xr = data as u32;
    let mut xl = (data >> 32) as u32;

    for i in 0..16 {
        xl = xl ^ state.p[i];
        xr = f(state, xl) ^ xr;
        swap(&mut xl, &mut xr);
    }
    swap(&mut xl, &mut xr);
    xr = xr ^ state.p[16];
    xl = xl ^ state.p[17];

    (xl as u64) << 32 | (xr as u64)
}
```

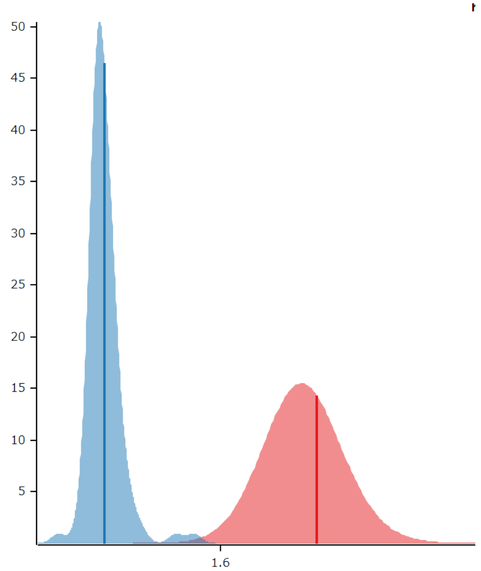
```rust
pub fn encrypt(state: &State, data: u64) -> u64 {
    let mut xr = data as u32;
    let mut xl = (data >> 32) as u32;

    xl ^= state.p[0];
    for i in 1..9 {
        xr ^= (((state.s0[(xl>>24) as u8 as usize].wrapping_a
        (state.s3[(xl) as u8 as usize]) ^ state.p[i*2-1];
        xl ^= (((state.s0[(xr>>24) as u8 as usize].wrapping_a
        (state.s3[(xr) as u8 as usize]) ^ state.p[i*2];
    }
    xr ^= state.p[17];

    (xr as u64) << 32 | (xl as u64)
}
```

f() を呼び出さない実装に変更

計測

紅の魔術工房
Atelier kurenaif

10% ほど改善できた




```
or reduce sample count to 50.
     time:   [1.5250 ms 1.5267 ms 1.5288 ms]
     change: [-8.7845% -8.1074% -7.6022%] (p = 0.00 < 0.05)
     Performance has improved.
ong 100 measurements (5.00%)
d
```

まとめ

bcrypt を実装してみた。

Blowfish の state の生成が重い。

cost で 2^31 まで負荷をあげれる。

重い処理は encrypt()

この子は単純な xor と足し算しか

してなくて、高速化が困難。

(しかも非線形の処理で式変形も困難)

紅の魔術工房
Atelier kurenaif