Examples of symmetric primitives

D. J. Bernstein

<table>
<thead>
<tr>
<th></th>
<th>message len</th>
<th>tweak</th>
<th>key</th>
<th>encrypts</th>
<th>authenticates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permutation</td>
<td>fixed</td>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Compression function</td>
<td>fixed</td>
<td>yes</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Block cipher</td>
<td>fixed</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
</tr>
<tr>
<td>Tweakable block cipher</td>
<td>fixed</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
</tr>
<tr>
<td>Hash function</td>
<td>variable</td>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MAC (without nonce)</td>
<td>variable</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>MAC (using nonce)</td>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Stream cipher</td>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Authenticated cipher</td>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
Examples of symmetric primitives

D. J. Bernstein

<table>
<thead>
<tr>
<th></th>
<th>message len</th>
<th>tweak</th>
<th>key</th>
<th>encrypts</th>
<th>authenticates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permutation</td>
<td>fixed</td>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Compression function</td>
<td>fixed</td>
<td>yes</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>Block cipher</td>
<td>fixed</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
</tr>
<tr>
<td>Tweakable block cipher</td>
<td>fixed</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
</tr>
<tr>
<td>Hash function</td>
<td>variable</td>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>MAC (without nonce)</td>
<td>variable</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>MAC (using nonce)</td>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Stream cipher</td>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>Authenticated cipher</td>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

1994 Wheeler–Needham "TEA, a tiny encryption algorithm":

```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 32;r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0] ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2] ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```
## Examples of symmetric primitives

<table>
<thead>
<tr>
<th></th>
<th>message len</th>
<th>tweak</th>
<th>key</th>
<th>encrypts</th>
<th>authenticates</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Permutation</strong></td>
<td>fixed</td>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>Compression function</strong></td>
<td>fixed</td>
<td>yes</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>Block cipher</strong></td>
<td>fixed</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
</tr>
<tr>
<td><strong>Tweakable block cipher</strong></td>
<td>fixed</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
</tr>
<tr>
<td><strong>Hash function</strong></td>
<td>variable</td>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td><strong>MAC (without nonce)</strong></td>
<td>variable</td>
<td>no</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><strong>MAC (using nonce)</strong></td>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td><strong>Stream cipher</strong></td>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td><strong>Authenticated cipher</strong></td>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

### 1994 Wheeler–Needham “TEA, a tiny encryption algorithm”:

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1], r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```
Examples of symmetric primitives

<table>
<thead>
<tr>
<th>Message len</th>
<th>tweak</th>
<th>key</th>
<th>Encrypts</th>
<th>Authenticates</th>
</tr>
</thead>
<tbody>
<tr>
<td>fixed</td>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>fixed</td>
<td>yes</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>fixed</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
</tr>
<tr>
<td>fixed</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
</tr>
<tr>
<td>variable</td>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>variable</td>
<td>yes</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>variable</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>variable</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

1994 Wheeler–Needham “TEA, a tiny encryption algorithm”:

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```
### TEA Implementation

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

### Table: TEA Options

<table>
<thead>
<tr>
<th>tweak</th>
<th>key</th>
<th>encrypts</th>
<th>authenticates</th>
</tr>
</thead>
<tbody>
<tr>
<td>no</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>—</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
<td>—</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
1994 Wheeler–Needham “TEA, a tiny encryption algorithm”:

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x;  b[1] = y;
}
```

uint32: 32 bits (b₀; b₁; ⋯; b₃₁) representing the “unsigned” integer b₀ + 2b₁ + ⋯ + 2³¹b₃₁.

+: addition mod 2³².

c += d: same as c = c + d.

^: xor; ⊕; addition of each bit separately mod 2.

Lower precedence than + in C, so spacing is not misleading.

<<4: multiplication by 16, i.e., (0; 0; 0; 0; b₀; b₁; ⋯; b₂₇).

>>5: division by 32, i.e., (b₅; b₆; ⋯; b₃₁; 0; 0; 0; 0; 0).
1994 Wheeler–Needham “TEA, a tiny encryption algorithm”:

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
             ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
             ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

`uint32`: 32 bits (`b_0`, `b_1`, ..., `b_31`) representing the “unsigned" integer `b_0 + 2 b_1 + ... + 2^31 b_31`.

`+`: addition mod 2^{32}.

`c += d`: same as `c = c + d`.

`^`: xor; \(\oplus\); addition of each bit separately mod 2.

Lower precedence than `+` in C, so spacing is not misleading.

`<<4`: multiplication by 16, \((0, 0, 0, 0, b_0, b_1, \ldots, b_27, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)\).

`>>5`: division by 32, \((b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0)\).
1994 Wheeler–Needham “TEA, a tiny encryption algorithm”:

```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 32;r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0]
            ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2]
            ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

uint32: 32 bits ($b_0, b_1, \ldots$) representing the “unsigned” integer $b_0 + 2b_1 + \cdots + 2^{31}b_{31}$.

+: addition mod $2^{32}$.

c += d: same as $c = c + d$.

^: xor; $\oplus$; addition of each bit separately mod 2. Lower precedence than + in C, so spacing is not misleading.

<<4: multiplication by 16, i.e.,

(0, 0, 0, 0, $b_0, b_1, \ldots, b_{27}$).

>>5: division by 32, i.e.,

($b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0$).
1994 Wheeler–Needham “TEA, a tiny encryption algorithm”:

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

- `uint32`: 32 bits \((b_0, b_1, \ldots, b_{31})\) representing the “unsigned” integer \(b_0 + 2b_1 + \cdots + 2^{31}b_{31}\).
- `+`: addition mod \(2^{32}\).
- `c += d`: same as \(c = c + d\).
- `^`: xor; \(\oplus\); addition of each bit separately mod 2. Lower precedence than + in C, so spacing is not misleading.
- `<<4`: multiplication by 16, i.e., \((0, 0, 0, 0, b_0, b_1, \ldots, b_{27})\).
- `>>5`: division by 32, i.e., \((b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0, 0)\).
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}

uint32: 32 bits \( (b_0, b_1, \ldots, b_{31}) \) representing the “unsigned” integer \( b_0 + 2b_1 + \cdots + 2^{31}b_{31} \).

+ : addition mod \( 2^{32} \).

c += d: same as \( c = c + d \).

\( \wedge \) : xor; \( \oplus \); addition of each bit separately mod 2. Lower precedence than + in C, so spacing is not misleading.

\( \ll 4 \) : multiplication by 16, i.e., \( (0, 0, 0, 0, b_0, b_1, \ldots, b_{27}) \).

\( \gg 5 \) : division by 32, i.e., \( (b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0) \).

Functionality

TEA is a 64-bit block cipher with a 128-bit key.
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
      c += 0x9e3779b9;
      x += y+c ^ (y<<4)+k[0] ^ (y>>5)+k[1];
      y += x+c ^ (x<<4)+k[2] ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}

uint32: 32 bits \( (b_0, b_1, \ldots, b_{31}) \) representing the “unsigned” integer 
\( b_0 + 2b_1 + \cdots + 2^{31}b_{31} \).

\+: addition mod \( 2^{32} \).

c += d: same as \( c = c + d \).

\^: xor; \( \oplus \); addition of each bit separately mod 2.

Lower precedence than \( + \) in C, so spacing is not misleading.

\(<<4\): multiplication by 16, i.e., 
\( (0, 0, 0, 0, b_0, b_1, \ldots, b_{27}) \).

\(<<5\): division by 32, i.e., 
\( (b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0) \).

Functionality
TEA is a 64-bit block cipher with a 128-bit key.
Functionality

TEA is a **64-bit block cipher** with a **128-bit key**.

```c
uint32: 32 bits \( (b_0, b_1, \ldots, b_{31}) \)
representing the “unsigned” integer \( b_0 + 2b_1 + \cdots + 2^{31}b_{31} \).

+: addition mod \( 2^{32} \).

c += d: same as \( c = c + d \).

^: xor; \( \oplus \); addition of each bit separately mod 2.
Lower precedence than + in C, so spacing is not misleading.

<<4: multiplication by 16, i.e.,
\( (0, 0, 0, 0, b_0, b_1, \ldots, b_{27}) \).

>>5: division by 32, i.e.,
\( (b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0) \).
```
uint32: 32 bits \((b_0, b_1, \ldots, b_{31})\) representing the “unsigned” integer \(b_0 + 2b_1 + \cdots + 2^{31}b_{31}\).

+: addition mod \(2^{32}\).

c += d: same as \(c = c + d\).

^: xor; \(\oplus\); addition of each bit separately mod 2. Lower precedence than + in C, so spacing is not misleading.

<<4: multiplication by 16, i.e., \((0, 0, 0, 0, b_0, b_1, \ldots, b_{27})\).

>>5: division by 32, i.e., \((b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0, 0)\).

Functionality

TEA is a 64-bit block cipher with a 128-bit key.
uint32: 32 bits \((b_0, b_1, \ldots, b_{31})\) representing the “unsigned” integer \(b_0 + 2b_1 + \cdots + 2^{31}b_{31}\).

+: addition mod \(2^{32}\).

c += d: same as \(c = c + d\).

^: xor; \(\oplus\); addition of each bit separately mod 2.

Lower precedence than + in C, so spacing is not misleading.

<<4: multiplication by 16, i.e., \((0, 0, 0, 0, b_0, b_1, \ldots, b_{27})\).

>>5: division by 32, i.e., \((b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0)\).

Functionality

TEA is a 64-bit block cipher with a 128-bit key.

Input: 128-bit key (namely \(k[0], k[1], k[2], k[3]\)); 64-bit plaintext \((b[0], b[1])\).

Output: 64-bit ciphertext (final \(b[0], b[1]\)).
**Functionality**

TEA is a **64-bit block cipher** with a **128-bit key**.

Input: 128-bit key (namely $k[0], k[1], k[2], k[3]$);
64-bit plaintext $(b[0], b[1])$.

Output: 64-bit ciphertext $(b[0], b[1])$.

Can efficiently **encrypt**: $(key, plaintext) \mapsto ciphertext$.

Can efficiently **decrypt**: $(key, ciphertext) \mapsto plaintext$.

**uint32**: 32 bits $(b_0, b_1, \ldots, b_{31})$ representing the “unsigned” integer $b_0 + 2b_1 + \cdots + 2^{31}b_{31}$.

$+$: addition mod $2^{32}$.

$c += d$: same as $c = c + d$.

$\wedge$: xor; $\oplus$; addition of each bit separately mod 2.

Lower precedence than $+$ in C, so spacing is not misleading.

$<<4$: multiplication by 16, i.e.,
$(0, 0, 0, 0, b_0, b_1, \ldots, b_{27})$.

$>>5$: division by 32, i.e.,
$(b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0, 0)$. 


32 bits \((b_0, b_1, \ldots, b_{31})\) representing the “unsigned” integer \(b_0 + 2b_1 + \cdots + 2^{31}b_{31}\). Addition mod \(2^{32}\).

\(+=\): same as \(c = c + d\).

\(\oplus\): addition of each bit separately mod 2. Lower precedence than + in C, so spacing is not misleading.

\(<|<\): multiplication by 16, i.e., \(0, 0, b_0, b_1, \ldots, b_{27}\).

\(>|>\): division by 32, i.e., \(b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0, 0\).

---

**Functionality**

TEA is a **64-bit block cipher** with a **128-bit key**.

**Input:** 128-bit key (namely \(k[0], k[1], k[2], k[3]\)); 64-bit **plaintext** \((b[0], b[1])\).

**Output:** 64-bit **ciphertext** (final \(b[0], b[1]\)).

Can efficiently **encrypt**:

\((\text{key}, \text{plaintext}) \mapsto \text{ciphertext}\).

Can efficiently **decrypt**:

\((\text{key}, \text{ciphertext}) \mapsto \text{plaintext}\).

---

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c \oplus (y << 4) + k[0] \oplus (y >> 5) + k[1];
        y += x + c \oplus (x << 4) + k[2] \oplus (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```
$$b_0, b_1, \ldots, b_{31}$$ represents the “unsigned” integer $$b_0 \cdot 2^0 + b_1 \cdot 2^1 + \ldots + 2^{31} b_{31}$$.

$$\oplus$$: xor; addition of each bit separately mod 2. Lower precedence than + in C, so spacing is not misleading.

$$\ll{\text{-}4}$$: multiplication by 16, i.e., $$(0, 0, 0, 0, b_0, b_1, \ldots, b_{27})$$.

$$\gg{\text{-}5}$$: division by 32, i.e., $$(b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0)$$. 

Functionality

TEA is a 64-bit block cipher with a 128-bit key.

Input: 128-bit key (namely $$k[0], k[1], k[2], k[3]$$);
64-bit plaintext $$(b[0], b[1])$$.

Output: 64-bit ciphertext $$(\text{final } b[0], b[1])$$.

Can efficiently encrypt: 
(key, plaintext) $$\mapsto$$ ciphertext.

Can efficiently decrypt: 
(key, ciphertext) $$\mapsto$$ plaintext.

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
             ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
             ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```
Functionality

TEA is a **64-bit block cipher** with a **128-bit key**.

Input: 128-bit key (namely $k[0], k[1], k[2], k[3]$);
64-bit plaintext ($b[0], b[1]$).

Output: 64-bit ciphertext (final $b[0], b[1]$).

Can efficiently **encrypt**: (key, plaintext) $\mapsto$ ciphertext.

Can efficiently **decrypt**: (key, ciphertext) $\mapsto$ plaintext.

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```
Functionality

TEA is a 64-bit block cipher with a 128-bit key.

Input: 128-bit key (namely \(k[0], k[1], k[2], k[3]\)); 64-bit plaintext \((b[0], b[1])\).

Output: 64-bit ciphertext \((final b[0], b[1])\).

Can efficiently encrypt: \((key, plaintext) \mapsto ciphertext\).

Can efficiently decrypt: \((key, ciphertext) \mapsto plaintext\).

Wait, how can we decrypt?

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```
Functionality

TEA is a 64-bit block cipher with a 128-bit key.

Input: 128-bit key (namely $k[0], k[1], k[2], k[3]$);
64-bit plaintext $(b[0], b[1])$.

Output: 64-bit ciphertext $(b[0], b[1])$.

Can efficiently encrypt:
$(key; plaintext) \mapsto ciphertext$.

Can efficiently decrypt:
$(key; ciphertext) \mapsto plaintext$.

```
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Answer: Each step is invertible.

```
void decrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 32 * 0x9e3779b9;
    for (r = 0; r < 32; r += 1) {
        y -= x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
        x -= y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        c -= 0x9e3779b9;
    }
    b[0] = x; b[1] = y;
}
```
Wait, how can we decrypt?

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Answer: Each step is invertible.

```c
void decrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 32 * 0x9e3779b9;
    for (r = 0; r < 32; r += 1) {
        c -= 0x9e3779b9;
        y -= x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
        x -= y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
    }
    b[0] = x; b[1] = y;
}
```
Functionality

TEA is a 64-bit block cipher with a 128-bit key.

Input: 128-bit key (namely $k[0], k[1], k[2], k[3]$);
64-bit plaintext ($b[0], b[1]$).

Output: 64-bit ciphertext (final $b[0], b[1]$).

Can efficiently encrypt: $(key; plaintext) \mapsto ciphertext.$

Can efficiently decrypt: $(key; ciphertext) \mapsto plaintext.$

Wait, how can we decrypt?

Answer: Each step is invertible.

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

```c
void decrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 32 * 0x9e3779b9;
    for (r = 0; r < 32; r += 1) {
        y -= x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
        x -= y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        c -= 0x9e3779b9;
    }
    b[0] = x; b[1] = y;
}
```
Wait, how can we decrypt?

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Answer: Each step is invertible.

```c
void decrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 32 * 0x9e3779b9;
    for (r = 0; r < 32; r += 1) {
        y -= x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
        x -= y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        c -= 0x9e3779b9;
    }
    b[0] = x; b[1] = y;
}
```
void encrypt(uint32 *b,uint32 *k)
{
uint32 x = b[0], y = b[1];
uint32 r, c = 0;
for (r = 0;r < 32;r += 1) {
    c += 0x9e3779b9;
    x += y+c ^ (y<<4)+k[0]
        ^ (y>>5)+k[1];
    y += x+c ^ (x<<4)+k[2]
        ^ (x>>5)+k[3];
}
b[0] = x; b[1] = y;
}

Answer: Each step is invertible.

void decrypt(uint32 *b,uint32 *k)
{
uint32 x = b[0], y = b[1];
uint32 r, c = 32 * 0x9e3779b9;
for (r = 0;r < 32;r += 1) {
    c -= 0x9e3779b9;
    y -= x+c ^ (x<<4)+k[2]
        ^ (x>>5)+k[3];
    x -= y+c ^ (y<<4)+k[0]
        ^ (y>>5)+k[1];
}
b[0] = x; b[1] = y;
}

Generalization, Feistel network (used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):
x += function1(y,k);
y += function2(x,k);
x += function3(y,k);
y += function4(x,k);
...

Decryption, inverting each step:
...
y -= function4(x,k);
x -= function3(y,k);
y -= function2(x,k);
x -= function1(y,k);
Wait, how can we decrypt?

```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 32;r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0] ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2] ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

Answer: Each step is invertible.

```c
void decrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 32 * 0x9e3779b9;
    for (r = 0;r < 32;r += 1) {
        y -= x+c ^ (x<<4)+k[2] ^ (x>>5)+k[3];
        x -= y+c ^ (y<<4)+k[0] ^ (y>>5)+k[1];
        c -= 0x9e3779b9;
    }
    b[0] = x; b[1] = y;
}
```

Generalization, Feistel network (used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):

```
x += function1(y,k);
y += function2(x,k);
x += function3(y,k);
y += function4(x,k);
...
```

Decryption, inverting each step:

```
y -= function4(x,k);
x -= function3(y,k);
y -= function2(x,k);
x -= function1(y,k);
...```

void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}

Answer: Each step is invertible.

void decrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 32 * 0x9e3779b9;
    for (r = 0; r < 32; r += 1) {
        y -= x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
        x -= y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        c -= 0x9e3779b9;
    }
    b[0] = x; b[1] = y;
}

Generalization, Feistel network (used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):

\[
\begin{align*}
    x & \leftarrow x + f_1(y, k) \\
    y & \leftarrow y + f_2(x, k) \\
    x & \leftarrow x + f_3(y, k) \\
    y & \leftarrow y + f_4(x, k)
\end{align*}
\]

Decryption, inverting each step:

\[
\begin{align*}
    x & \leftarrow x - f_1(y, k) \\
    y & \leftarrow y - f_2(x, k) \\
    x & \leftarrow x - f_3(y, k) \\
    y & \leftarrow y - f_4(x, k)
\end{align*}
\]
Answer: Each step is invertible.

void decrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 32 * 0x9e3779b9;
    for (r = 0; r < 32; r += 1) {
        y -= x+c ^ (x<<4)+k[2]
             ^ (x>>5)+k[3];
        x -= y+c ^ (y<<4)+k[0]
             ^ (y>>5)+k[1];
        c -= 0x9e3779b9;
    }
    b[0] = x; b[1] = y;
}

Generalization, Feistel network (used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):

x += function1(y,k);
y += function2(x,k);
x += function3(y,k);
y += function4(x,k);
...

Decryption, inverting each step:

... y -= function4(x,k);
x -= function3(y,k);
y -= function2(x,k);
x -= function1(y,k);
Each step is invertible.

```c
void decrypt(uint32 *b, uint32 *k) {
  uint32 x = b[0], y = b[1];
  uint32 r, c = 32 * 0x9e3779b9;
  for (r = 0; r < 32; r += 1) {
    y -= x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    x -= y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
    c -= 0x9e3779b9;
  }
  b[0] = x; b[1] = y;
}
```

Generalization, **Feistel network** (used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):

```c
x += function1(y, k);
y += function2(x, k);
x += function3(y, k);
y += function4(x, k);
...
```

Decryption, inverting each step:

```c
...
y -= function4(x, k);
x -= function3(y, k);
y -= function2(x, k);
x -= function1(y, k);
```

Higher-level functionality

User's message is long sequence of 64-bit blocks $m_0; m_1; m_2; \ldots$. 
void decrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 32 * 0x9e3779b9;
    for (r = 0; r < 32; r += 1) {
        y -= x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
        x -= y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        c -= 0x9e3779b9;
    }
    b[0] = x; b[1] = y;
}

Generalization, **Feistel network**
(used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):

\[
\begin{align*}
    x &\leftarrow x + \text{function1}(y, k) \\
    y &\leftarrow y + \text{function2}(x, k) \\
    x &\leftarrow x + \text{function3}(y, k) \\
    y &\leftarrow y + \text{function4}(x, k)
\end{align*}
\]

... 

Decryption, inverting each step:

\[
\begin{align*}
    \cdots \\
    y &\leftarrow y - \text{function4}(x, k) \\
    x &\leftarrow x - \text{function3}(y, k) \\
    y &\leftarrow y - \text{function2}(x, k) \\
    x &\leftarrow x - \text{function1}(y, k)
\end{align*}
\]

Higher-level functionality
User’s message is a long sequence of 64-bit blocks \(m_0; m_1; m_2; \cdots\).
Generalization, **Feistel network** (used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):

```c
uint32 x = b[0], y = b[1];
uint32 r, c = 0x9e3779b9;
for (r = 0; r < 32; r += 1) {
    y -= x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    x -= y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
    c -= 0x9e3779b9;
}
b[0] = x; b[1] = y;
```

Decryption, inverting each step:

```c
...
y -= function4(x, k);
x -= function3(y, k);
y -= function2(x, k);
x -= function1(y, k);
```

Higher-level functionality

User’s message is long sequence of 64-bit blocks $m_0, m_1, m_2, \ldots$. 
Generalization, **Feistel network** (used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):

\[
\begin{align*}
x &+ = \text{function1}(y,k); \\
y &+ = \text{function2}(x,k); \\
x &+ = \text{function3}(y,k); \\
y &+ = \text{function4}(x,k); \\
\ldots
\end{align*}
\]

Decryption, inverting each step:

\[
\begin{align*}
\ldots \\
y &- = \text{function4}(x,k); \\
x &- = \text{function3}(y,k); \\
y &- = \text{function2}(x,k); \\
x &- = \text{function1}(y,k);
\end{align*}
\]

Higher-level functionality

User’s message is long sequence of 64-bit blocks \( m_0, m_1, m_2, \ldots \).
Generalization, **Feistel network** (used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):

\[
x += \text{function1}(y, k);
y += \text{function2}(x, k);
x += \text{function3}(y, k);
y += \text{function4}(x, k);
\cdots
\]

Decryption, inverting each step:

\[
\cdots
y -= \text{function4}(x, k);
x -= \text{function3}(y, k);
y -= \text{function2}(x, k);
x -= \text{function1}(y, k);
\]

**Higher-level functionality**

User’s message is long sequence of 64-bit blocks \( m_0, m_1, m_2, \ldots \).

TEA-CTR produces ciphertext \( c_0 = m_0 \oplus \text{TEA}_k(n, 0) \), \( c_1 = m_1 \oplus \text{TEA}_k(n, 1) \), \( c_2 = m_2 \oplus \text{TEA}_k(n, 2) \), \ldots

using 128-bit key \( k \),

32-bit **nonce** \( n \),

32-bit **block counter** \( 0, 1, 2, \ldots \).
Generalization, **Feistel network** (used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):

\[
\begin{align*}
x &\xleftarrow{} \text{function1}(y, k); \\
y &\xleftarrow{} \text{function2}(x, k); \\
x &\xleftarrow{} \text{function3}(y, k); \\
y &\xleftarrow{} \text{function4}(x, k);
\end{align*}
\]

\[
\ldots
\]

Decryption, inverting each step:

\[
\begin{align*}
y &\xrightarrow{} \text{function4}(x, k); \\
x &\xrightarrow{} \text{function3}(y, k); \\
y &\xrightarrow{} \text{function2}(x, k); \\
x &\xrightarrow{} \text{function1}(y, k);
\end{align*}
\]

Higher-level functionality

User’s message is long sequence of 64-bit blocks \(m_0, m_1, m_2, \ldots\).

TEA-CTR produces ciphertext

\[
\begin{align*}
c_0 &= m_0 \oplus \text{TEA}_k(n, 0), \\
c_1 &= m_1 \oplus \text{TEA}_k(n, 1), \\
c_2 &= m_2 \oplus \text{TEA}_k(n, 2), \ldots
\end{align*}
\]

using 128-bit key \(k\),

32-bit **nonce** \(n\),

32-bit **block counter** \(0, 1, 2, \ldots\).

CTR is a **mode of operation** that converts block cipher TEA into **stream cipher** TEA-CTR.
Higher-level functionality

User’s message is long sequence of 64-bit blocks \( m_0, m_1, m_2, \ldots \).

TEA-CTR produces ciphertext
\[
\begin{align*}
c_0 &= m_0 \oplus \text{TEA}_k(n, 0), \\
c_1 &= m_1 \oplus \text{TEA}_k(n, 1), \\
c_2 &= m_2 \oplus \text{TEA}_k(n, 2), \ldots
\end{align*}
\]
using 128-bit key \( k \),
32-bit nonce \( n \),
32-bit block counter \( 0, 1, 2, \ldots \).

CTR is a mode of operation that converts block cipher TEA into stream cipher TEA-CTR.

User also wants to recognize forged/modified ciphertexts.
Generalization, Feistel network (used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):

\[ x \mathrel{+}= \text{function1}(y, k); \]
\[ y \mathrel{+}= \text{function2}(x, k); \]
\[ x \mathrel{+}= \text{function3}(y, k); \]
\[ y \mathrel{+}= \text{function4}(x, k); \]
\[ ... \]
Decryption, inverting each step:

\[ y \mathrel{-}= \text{function4}(x, k); \]
\[ x \mathrel{-}= \text{function3}(y, k); \]
\[ y \mathrel{-}= \text{function2}(x, k); \]
\[ x \mathrel{-}= \text{function1}(y, k); \]

Higher-level functionality

User’s message is long sequence of 64-bit blocks \( m_0, m_1, m_2, \ldots \).

TEA-CTR produces ciphertext

\[
c_0 = m_0 \oplus \text{TEA}_k(n, 0),
\]
\[
c_1 = m_1 \oplus \text{TEA}_k(n, 1),
\]
\[
c_2 = m_2 \oplus \text{TEA}_k(n, 2), \ldots
\]
using 128-bit key \( k \),
32-bit nonce \( n \),
32-bit block counter \( 0, 1, 2, \ldots \).

CTR is a mode of operation that converts block cipher TEA into stream cipher TEA-CTR.

User also wants to recognize forged/modified ciphertexts.
Higher-level functionality

User’s message is long sequence of 64-bit blocks $m_0, m_1, m_2, \ldots$.

TEA-CTR produces ciphertext

$c_0 = m_0 \oplus \text{TEA}_k(n, 0)$,
$c_1 = m_1 \oplus \text{TEA}_k(n, 1)$,
$c_2 = m_2 \oplus \text{TEA}_k(n, 2)$, \ldots

using 128-bit key $k$,
32-bit nonce $n$,
32-bit block counter 0, 1, 2, \ldots.

CTR is a mode of operation that converts block cipher TEA into stream cipher TEA-CTR.

User also wants to recognize forged/modified ciphertexts.
Higher-level functionality
User’s message is long sequence of 64-bit blocks $m_0, m_1, m_2, \ldots$.

TEA-CTR produces ciphertext
\[c_0 = m_0 \oplus \text{TEA}_k(n, 0),\]
\[c_1 = m_1 \oplus \text{TEA}_k(n, 1),\]
\[c_2 = m_2 \oplus \text{TEA}_k(n, 2), \ldots\]
using 128-bit key $k$,
32-bit nonce $n$,
32-bit block counter 0, 1, 2, \ldots.

CTR is a mode of operation that converts block cipher TEA into stream cipher TEA-CTR.

User also wants to recognize forged/modified ciphertexts.
Higher-level functionality

User’s message is long sequence of 64-bit blocks $m_0, m_1, m_2, \ldots$.

TEA-CTR produces ciphertext

$c_0 = m_0 \oplus \text{TEA}_k(n, 0),$
$c_1 = m_1 \oplus \text{TEA}_k(n, 1),$
$c_2 = m_2 \oplus \text{TEA}_k(n, 2), \ldots$

using 128-bit key $k$,
32-bit nonce $n$,
32-bit block counter 0, 1, 2, \ldots.

CTR is a mode of operation

that converts block cipher TEA into stream cipher TEA-CTR.

User also wants to recognize forged/modified ciphertexts.

Usual strategy:

append **authenticator** to the ciphertext $c = (c_0, c_1, c_2, \ldots)$. 
Higher-level functionality
User’s message is long sequence of 64-bit blocks \( m_0, m_1, m_2, \ldots \).

TEA-CTR produces ciphertext
\[
c_0 = m_0 \oplus \text{TEA}_k(n, 0), \\
c_1 = m_1 \oplus \text{TEA}_k(n, 1), \\
c_2 = m_2 \oplus \text{TEA}_k(n, 2), \ldots
\]
using 128-bit key \( k \),
32-bit nonce \( n \),
32-bit block counter \( 0, 1, 2, \ldots \).

CTR is a mode of operation that converts block cipher TEA into stream cipher TEA-CTR.

User also wants to recognize forged/modified ciphertexts.

Usual strategy:
append authenticator to the ciphertext \( c = (c_0, c_1, c_2, \ldots) \).

TEA-XCBC-MAC computes
\[
a_0 = \text{TEA}_j(c_0), \\
a_1 = \text{TEA}_j(c_1 \oplus a_0), \\
a_2 = \text{TEA}_j(c_2 \oplus a_1), \ldots, \\
a_{\ell-1} = \text{TEA}_j(c_{\ell-1} \oplus a_{\ell-2}), \\
a_{\ell} = \text{TEA}_j(i \oplus c_{\ell} \oplus a_{\ell-1})
\]
using 128-bit key \( j \), 64-bit key \( i \).

Authenticator is \( a_{\ell} \): i.e.,
transmit \( (c_0, c_1, \ldots, c_\ell, a_{\ell}) \).
Higher-level functionality

User's message is long sequence of 64-bit blocks $m_0, m_1, m_2, \ldots$.

CTR produces ciphertext

$\oplus \text{TEA}_k(n, 0),$
$\oplus \text{TEA}_k(n, 1),$
$\oplus \text{TEA}_k(n, 2), \ldots$

using 128-bit key $k$, 32-bit nonce $n$, 32-bit block counter $0, 1, 2, \ldots$.

CTR is a mode of operation that converts block cipher TEA into stream cipher TEA-CTR.

User also wants to recognize forged/modified ciphertexts.

Usual strategy:
append authenticator to
the ciphertext $c = (c_0, c_1, c_2, \ldots)$.

TEA-XCBC-MAC computes

$a_0 = \text{TEA}_j(c_0),$
$a_1 = \text{TEA}_j(c_1 \oplus a_0),$
$a_2 = \text{TEA}_j(c_2 \oplus a_1), \ldots,$
$a_{\ell-1} = \text{TEA}_j(c_{\ell-1} \oplus a_{\ell-2}),$
$a_\ell = \text{TEA}_j(i \oplus c_\ell \oplus a_{\ell-1})$

using 128-bit key $j$, 64-bit key $i$.

Authenticator is $a_\ell$: i.e.,
transmit $(c_0, c_1, \ldots, c_\ell, a_\ell)$.

Specifying TEA-CTR-XCBC-MAC authenticated cipher:

320-bit key $(k, j, i)$.
Specify how this is chosen:
uniform random 320-bit string.
Higher-level functionality
User's message is long sequence
of $64$-bit blocks $m_0, m_1, m_2, \ldots$.

TEA-CTR produces ciphertext $c_0 = m_0 \oplus \text{TEA}_k(n, 0)$, $c_1 = m_1 \oplus \text{TEA}_k(n, 1)$, $c_2 = m_2 \oplus \text{TEA}_k(n, 2)$, $\ldots$ using $128$-bit key $k$, $32$-bit nonce $n$, $32$-bit block counter $0, 1, 2, \ldots$. CTR is a mode of operation that converts block cipher TEA into stream cipher TEA-CTR.

User also wants to recognize forged/modified ciphertexts. Usual strategy: append authenticator to the ciphertext $c = (c_0, c_1, c_2, \ldots)$.

TEA-XCBC-MAC computes $a_0 = \text{TEA}_j(c_0)$, $a_1 = \text{TEA}_j(c_1 \oplus a_0)$, $a_2 = \text{TEA}_j(c_2 \oplus a_1)$, $\ldots$, $a_{\ell - 1} = \text{TEA}_j(c_{\ell - 1} \oplus a_{\ell - 2})$, $a_{\ell} = \text{TEA}_j(i \oplus c_{\ell} \oplus a_{\ell - 1})$ using $128$-bit key $j$, $64$-bit key $i$.

Authenticator is $a_{\ell}$: i.e., transmit $(c_0, c_1, \ldots, c_{\ell}, a_{\ell})$.

Specifying TEA-CTR-XCBC-MAC authenticated cipher: $320$-bit key $(k, j, i)$.
Specify how this is chosen: uniform random $320$-bit string.
User also wants to recognize forged/modified ciphertexts.

Usual strategy:
append authenticator to
the ciphertext $c = (c_0, c_1, c_2, \ldots)$.

TEA-XCBC-MAC computes

$a_0 = \text{TEA}_j(c_0)$,
$a_1 = \text{TEA}_j(c_1 \oplus a_0)$,
$a_2 = \text{TEA}_j(c_2 \oplus a_1)$, \ldots,
$a_{\ell-1} = \text{TEA}_j(c_{\ell-1} \oplus a_{\ell-2})$,
$a_{\ell} = \text{TEA}_j(i \oplus c_\ell \oplus a_{\ell-1})$
using 128-bit key $j$, 64-bit key $i$.
Authenticator is $a_\ell$: i.e.,
transmit $(c_0, c_1, \ldots, c_\ell, a_\ell)$.

Specifying TEA-CTR-XCBC-MAC authenticated cipher:

320-bit key $(k, j, i)$.
Specify how this is chosen:
uniform random 320-bit string.
User also wants to recognize forged/modified ciphertexts.

Usual strategy:
append **authenticator** to the ciphertext \( c = (c_0, c_1, c_2, \ldots) \).

TEA-XCBC-MAC computes
\[
a_0 = \text{TEA}_j(c_0), \\
a_1 = \text{TEA}_j(c_1 \oplus a_0), \\
a_2 = \text{TEA}_j(c_2 \oplus a_1), \ldots, \\
a_{\ell-1} = \text{TEA}_j(c_{\ell-1} \oplus a_{\ell-2}), \\
a_{\ell} = \text{TEA}_j(i \oplus c_{\ell} \oplus a_{\ell-1})
\]
using 128-bit key \( j \), 64-bit key \( i \).

Authenticator is \( a_{\ell} \): i.e.,
transmit \( (c_0, c_1, \ldots, c_{\ell}, a_{\ell}) \).

Specifying TEA-CTR-XCBC-MAC authenticated cipher:
320-bit key \( (k, j, i) \).
Specify how this is chosen: uniform random 320-bit string.
User also wants to recognize forged/modified ciphertexts.

Usual strategy:
append **authenticator** to
the ciphertext \( c = (c_0, c_1, c_2, \ldots) \).

TEA-XCBC-MAC computes
\[ a_0 = \text{TEA}_j(c_0), \]
\[ a_1 = \text{TEA}_j(c_1 \oplus a_0), \]
\[ a_2 = \text{TEA}_j(c_2 \oplus a_1), \ldots, \]
\[ a_{\ell-1} = \text{TEA}_j(c_{\ell-1} \oplus a_{\ell-2}), \]
\[ a_{\ell} = \text{TEA}_j(i \oplus c_{\ell} \oplus a_{\ell-1}) \]
using 128-bit key \( j \), 64-bit key \( i \).

Authenticator is \( a_{\ell} \): i.e.,
transmit \( (c_0, c_1, \ldots, c_{\ell}, a_{\ell}) \).

Specifying TEA-CTR-XCBC-MAC authenticated cipher:

320-bit key \( (k, j, i) \).
Specify how this is chosen:
uniform random 320-bit string.

Specify set of messages:
message is sequence of
at most \( 2^{32} \) 64-bit blocks.
(Can do some extra work
to allow sequences of bytes.)
User also wants to recognize forged/modified ciphertexts.

Usual strategy:
append **authenticator** to
the ciphertext \( c = (c_0, c_1, c_2, \ldots) \).

TEA-XCBC-MAC computes
\[
\begin{align*}
a_0 & = \text{TEA}_j(c_0), \\
a_1 & = \text{TEA}_j(c_1 \oplus a_0), \\
a_2 & = \text{TEA}_j(c_2 \oplus a_1), \ldots, \\
a_{\ell-1} & = \text{TEA}_j(c_{\ell-1} \oplus a_{\ell-2}), \\
a_{\ell} & = \text{TEA}_j(i \oplus c_{\ell} \oplus a_{\ell-1})
\end{align*}
\]
using 128-bit key \( j \), 64-bit key \( i \).

Authenticator is \( a_{\ell} \): i.e.,
transmit \( (c_0, c_1, \ldots, c_{\ell}, a_{\ell}) \).

Specifying TEA-CTR-XCBC-MAC **authenticated cipher**:

320-bit key \( (k, j, i) \).
Specify how this is chosen:
uniform random 320-bit string.

Specify set of messages:
message is sequence of
at most \( 2^{32} \) 64-bit blocks.
(Can do some extra work
to allow sequences of bytes.)

Specify how nonce is chosen:
message number. (Stateless
alternative: uniform random.)
User also wants to recognize forged/modified ciphertexts.

Usual strategy:

**authenticator** to ciphertext \( c = (c_0, c_1, c_2, \ldots) \).

TEA-XCBC-MAC computes

\[
\begin{align*}
A_j(c_0), \\
A_j(c_1 \oplus a_0), \\
A_j(c_2 \oplus a_1), \ldots, \\
TEA_j(c_{\ell-1} \oplus a_{\ell-2}), \\
A_j(i \oplus c_{\ell} \oplus a_{\ell-1})
\end{align*}
\]

8-bit key \( j \), 64-bit key \( i \).

Authenticator is \( a_{\ell} \): i.e.,

\( (c_0, c_1, \ldots, c_\ell, a_\ell) \).

Specifying TEA-CTR-XCBC-MAC authenticated cipher:

320-bit key \((k, j, i)\).

Specify how this is chosen:

uniform random 320-bit string.

Specify set of messages:

message is sequence of at most \(2^{32}\) 64-bit blocks.

(Can do some extra work to allow sequences of bytes.)

Specify how nonce is chosen:

message number. (Stateless alternative: uniform random.)

Is this secure?

Step 1: Define security for authenticated ciphers.
User also wants to recognize forged/modified ciphertexts.

Usual strategy: append authenticator to the ciphertext $c = (c_0, c_1, c_2, \ldots)$. TEA-XCBC-MAC computes:

- $a_0 = \text{TEA}_j(c_0)$,
- $a_1 = \text{TEA}_j(c_1 \oplus a_0)$,
- $a_2 = \text{TEA}_j(c_2 \oplus a_1)$,
- $\vdots$
- $a_{l-1} = \text{TEA}_j(c_{l-1} \oplus a_{l-2})$,
- $a_l = \text{TEA}_j(i \oplus c_0 \oplus a_{l-1})$

using 128-bit key $j$, 64-bit key $i$.

Authenticator is $a_l$: i.e., transmit $(c_0, c_1, \ldots, c_l, a_l)$.

Specifying TEA-CTR-XCBC-MAC authenticated cipher:

320-bit key $(k, j, i)$.
Specify how this is chosen: uniform random 320-bit string.

Specify set of messages: message is sequence of at most $2^{32}$ 64-bit blocks.
(Can do some extra work to allow sequences of bytes.)

Specify how nonce is chosen: message number. (Stateless alternative: uniform random.)

Is this secure?
Step 1: Define security for authenticated ciphers.
User also wants to recognize forged/modified ciphertexts.
Usual strategy:
append authenticator to the ciphertext $c = (c_0; c_1; c_2; \ldots)$. 

TEA-XCBC-MAC computes $a_0 = \text{TEA}_j(c_0)$, $a_1 = \text{TEA}_j(c_1 \oplus a_0)$, $a_2 = \text{TEA}_j(c_2 \oplus a_1)$, $\ldots$, $a_{i-1} = \text{TEA}_j(c_i \oplus a_{i-1})$, $a_i = \text{TEA}_j(i \oplus c_i \oplus a_{i-1})$ using 128-bit key $j$, 64-bit key $i$.

Authenticator is $a_i$: i.e., transmit $(c_0; c_1; \ldots; c_i; a_i)$.

Specifying TEA-CTR-XCBC-MAC authenticated cipher:
320-bit key $(k, j, i)$. Specify how this is chosen: uniform random 320-bit string.

Specify set of messages: message is sequence of at most $2^{32}$ 64-bit blocks. (Can do some extra work to allow sequences of bytes.)

Specify how nonce is chosen: message number. (Stateless alternative: uniform random.)

Is this secure?
Step 1: Define security for authenticated ciphers.
Specifying TEA-CTR-XCBC-MAC authenticated cipher:
320-bit key \((k, j, i)\).
Specify how this is chosen:
uniform random 320-bit string.

Specify set of messages:
message is sequence of
at most \(2^{32}\) 64-bit blocks.
(Can do some extra work
to allow sequences of bytes.)

Specify how nonce is chosen:
message number. (Stateless
alternative: uniform random.)

Is this secure?
Step 1: Define security
for authenticated ciphers.
Specifying TEA-CTR-XCBC-MAC authenticated cipher:
320-bit key \((k, j, i)\).
Specify how this is chosen:
uniform random 320-bit string.

Specify set of messages:
message is sequence of
at most \(2^{32}\) 64-bit blocks.
(Can do some extra work
to allow sequences of bytes.)

Specify how nonce is chosen:
message number. (Stateless
alternative: uniform random.)

Is this secure?
Step 1: Define security
for authenticated ciphers.
This is not easy to do!
Specifying TEA-CTR-XCBC-MAC authenticated cipher:
320-bit key \((k, j, i)\).
Specify how this is chosen:
uniform random 320-bit string.

Specify set of messages:
message is sequence of at most \(2^{32}\) 64-bit blocks.
(Can do some extra work to allow sequences of bytes.)

Specify how nonce is chosen:
message number. (Stateless alternative: uniform random.)

Is this secure?
Step 1: Define security for authenticated ciphers.
This is not easy to do!
Useless extreme: “It’s secure unless you show me the key.”
Too weak. Many ciphers leak plaintext or allow forgeries without leaking key.
Specifying TEA-CTR-XCBC-MAC authenticated cipher:

320-bit key \((k, j, i)\).

Specify how this is chosen: uniform random 320-bit string.

Specify set of messages: message is sequence of at most \(2^{32}\) 64-bit blocks. (Can do some extra work to allow sequences of bytes.)

Specify how nonce is chosen: message number. (Stateless alternative: uniform random.)

Is this secure?

Step 1: Define security for authenticated ciphers.

This is not easy to do!

Useless extreme: “It’s secure unless you show me the key.”

Too weak. Many ciphers leak plaintext or allow forgeries without leaking key.

Another useless extreme: “Any structure is an attack.”

Hard to define clearly.

Everything seems “attackable”.

Specifying TEA-CTR-XCBC-MAC authenticated cipher:
key \( (k, j, i) \).
how this is chosen:
random 320-bit string.
set of messages:
is sequence of \( 2^{32} \) 64-bit blocks.
some extra work
sequences of bytes.)
how nonce is chosen:
number. (Stateless
ve: uniform random.)

Is this secure?
Step 1: Define security for authenticated ciphers.
This is not easy to do!
Useless extreme: “It’s secure unless you show me the key.”
Too weak. Many ciphers leak plaintext or allow forgeries without leaking key.
Another useless extreme:
“All structure is an attack.”
Hard to define clearly.
Everything seems “attackable.”
Specifying TEA-CTR-XCBC-MAC authenticated cipher:
320-bit key ($k; j; i$).
Specify how this is chosen:
uniform random 320-bit string.
Specify set of messages:
message is sequence of at most $2^{32}$ 64-bit blocks.
(Can do some extra work to allow sequences of bytes.)
Specify how nonce is chosen:
message number. (Stateless alternative: uniform random.)

Is this secure?
Step 1: Define security for authenticated ciphers.
This is not easy to do!
Useless extreme: “It’s secure unless you show me the key.” Too weak. Many ciphers leak plaintext or allow forgeries without leaking key.
Another useless extreme: “Any structure is an attack.”
Hard to define clearly.
Everything seems “attackable”.

Step 2: After settling on target security definition, prove that security follows from simpler properties.
Specifying TEA-CTR-XCBC-MAC authenticated cipher:

- 320-bit key ($k;j;i$).
- Specify how this is chosen: uniform random 320-bit string.
- Specify set of messages: message is sequence of at most 2 $32^{64}$-bit blocks. (Can do some extra work to allow sequences of bytes.)
- Specify how nonce is chosen: message number. (Stateless alternative: uniform random.)

Is this secure?

Step 1: Define security for authenticated ciphers.

This is not easy to do!

Useless extreme: “It’s secure unless you show me the key.”

Too weak. Many ciphers leak plaintext or allow forgeries without leaking key.

Another useless extreme: “Any structure is an attack.”

Hard to define clearly.

Everything seems “attackable”.

Step 2: After settling on target security definition, prove that security follows from simpler properties.
Is this secure?

Step 1: Define security for authenticated ciphers.
This is not easy to do!

Useless extreme: “It’s secure unless you show me the key.”
Too weak. Many ciphers leak plaintext or allow forgeries without leaking key.

Another useless extreme: “Any structure is an attack.”
Hard to define clearly.
Everything seems “attackable”.

Step 2: After settling on target security definition, prove that security follows from simpler properties.
Is this secure?

Step 1: Define security for authenticated ciphers.

This is not easy to do!

Useless extreme: “It’s secure unless you show me the key.”
Too weak. Many ciphers leak plaintext or allow forgeries without leaking key.

Another useless extreme:
“Any structure is an attack.”
Hard to define clearly.
Everything seems “attackable”.

Step 2: After settling on target security definition, prove that security follows from simpler properties.

e.g. Prove PRF security of 
\[ n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots \]
assuming PRF security of 
\[ b \mapsto \text{TEA}_k(b). \]
Is this secure?

Step 1: Define security for authenticated ciphers.

This is not easy to do!

Useless extreme: “It’s secure unless you show me the key.”

Too weak. Many ciphers leak plaintext or allow forgeries without leaking key.

Another useless extreme: “Any structure is an attack.”

Hard to define clearly.

Everything seems “attackable”.

Step 2: After settling on target security definition, prove that security follows from simpler properties.

e.g. Prove PRF security of $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ assuming PRF security of $b \mapsto \text{TEA}_k(b)$.

i.e. Prove that any PRF attack against $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ implies PRF attack against $b \mapsto \text{TEA}_k(b)$.
Is this secure?

Step 1: Define security for authenticated ciphers. This is not easy to do!

Useless extreme: “It’s secure unless you show me the key.”

Too weak. Many ciphers leak plaintext or allow forgeries without leaking key.

Another useless extreme: “Any structure is an attack.”

Hard to define clearly. Everything seems “attackable”.

Step 2: After settling on target security definition, prove that security follows from simpler properties.

E.g. Prove PRF security of $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ assuming PRF security of $b \mapsto \text{TEA}_k(b)$.

I.e. Prove that any PRF attack against $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ implies PRF attack against $b \mapsto \text{TEA}_k(b)$.
Step 2: After settling on target security definition, prove that security follows from simpler properties.

e.g. Prove PRF security of $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ assuming PRF security of $b \mapsto \text{TEA}_k(b)$.

i.e. Prove that any PRF attack against $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ implies PRF attack against $b \mapsto \text{TEA}_k(b)$. 

PRF security of $n \mapsto \text{TEA}_k(n, 0), n \mapsto \text{TEA}_k(n, 1), \ldots$ implies PRF security of $b \mapsto \text{TEA}_k(b)$. 

Step 2: After settling on target security definition, prove that security follows from simpler properties.

e.g. Prove PRF security of $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ assuming PRF security of $b \mapsto \text{TEA}_k(b)$.

i.e. Prove that any PRF attack against $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ implies PRF attack against $b \mapsto \text{TEA}_k(b)$. 

PRF security of $n \mapsto \text{TEA}_k(n, 0), n \mapsto \text{TEA}_k(n, 1), \ldots$ implies PRF security of $b \mapsto \text{TEA}_k(b)$.
Step 2: After settling on target security definition, prove that security follows from simpler properties.

e.g. Prove PRF security of $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ assuming PRF security of $b \mapsto \text{TEA}_k(b)$.

i.e. Prove that any PRF attack against $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ implies PRF attack against $b \mapsto \text{TEA}_k(b)$. 
Step 2: After settling on target security definition, prove that security follows from simpler properties.

e.g. Prove PRF security of $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ assuming PRF security of $b \mapsto \text{TEA}_k(b)$.

i.e. Prove that any PRF attack against $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ implies PRF attack against $b \mapsto \text{TEA}_k(b)$.
After settling on a target security definition, prove that security follows from simpler properties.

Given PRF security of $A_k(n, 0), \text{TEA}_k(n, 1), \ldots$ and PRF security of $A_k(b)$.

Prove that a PRF attack against $A_k(n, 0), \text{TEA}_k(n, 1), \ldots$ implies PRF attack against $A_k(b)$.
Step 2: After settling on target security definition, prove that security follows from simpler properties. e.g. Prove PRF security of $n \mapsto \text{TEA}_k(n;0)$, $\text{TEA}_k(n;1)$, ... assuming PRF security of $b \mapsto \text{TEA}_k(b)$.

i.e. Prove that any PRF attack against $n \mapsto \text{TEA}_k(n;0)$, $\text{TEA}_k(n;1)$, ... implies PRF attack against $b \mapsto \text{TEA}_k(b)$. 

Privacy of TEA-CTR-XCBC-MAC

Authenticity of TEA-CTR-XCBC-MAC

Privacy of TEA-CTR

↑ ↑

Authenticity of TEA-XCBC-MAC

↑ ↑

PRF security of $n \mapsto \text{TEA}_k(n;0)$, $\text{TEA}_k(n;1)$, ... 

↑ ↑

PRF security of TEA

↑ ↖ ↗ ↑

PRP security of TEA

↑ ↑
Step 2: After settling on target security definition, prove that security follows from simpler properties. e.g. Prove PRF security of $n \mapsto \text{TEA}_k(n, 0); \text{TEA}_k(n, 1); \ldots$ assuming PRF security of $b \mapsto \text{TEA}_k(b)$. i.e. Prove that any PRF attack against $n \mapsto \text{TEA}_k(n, 0); \text{TEA}_k(n, 1); \ldots$ implies PRF attack against $b \mapsto \text{TEA}_k(b)$. 

Privacy of TEA-CTR-XCBC-MAC 

PRF security of $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ 

PRF security of TEA 

PRP security of TEA 

Privacy of TEA-CTR 

Authenticity of TEA-CTR-XCBC-MAC 

PRF security of TEA-XCBC-MAC 

Privacy of TEA-CTR 

Authenticity of TEA-XCBC-MAC 

PRF security of TEA-XCBC-MAC 

PRF security of TEA
privacy of TEA-CTR-XCBC-MAC

privacy of TEA-CTR

PRF security of $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$

PRF security of TEA

PRP security of TEA

authenticity of TEA-CTR-XCBC-MAC

authenticity of TEA-XCBC-MAC

PRF security of TEA-XCBC-MAC
Many things can go wrong here:
1. Security definition too weak.
PRP security of TEA

PRF security of TEA

PRF security of TEA-XCBC-MAC

PRF security of TEA-CTR-XCBC-MAC

Privacy of TEA-CTR

Privacy of TEA-CTR-XCBC-MAC

Authenticity of TEA-CTR

Authenticity of TEA-CTR-XCBC-MAC

Many things can go wrong here:
1. Security definition too weak.
Many things can go wrong here:

1. Security definition too weak.
Many things can go wrong here:
1. Security definition too weak.
Many things can go wrong here:
1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
Many things can go wrong here:
1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
3. Errors in proofs.
Did anyone write full proofs?
Did anyone check all details?
Many things can go wrong here:

1. Security definition too weak.

2. Internal mismatch between hypotheses and conclusions.

3. Errors in proofs.
   Did anyone write full proofs?
   Did anyone check all details?

4. Quantitative problems.
   e.g. 2016 Bhargavan–Leurent sweet32.info: Triple-DES broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.
Many things can go wrong here:
1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
3. Errors in proofs. Did anyone write full proofs? Did anyone check all details?
4. Quantitative problems. e.g. 2016 Bhargavan–Leurent sweet32.info: Triple-DES broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.
5. Is TEA PRP-secure?
Many things can go wrong here:

1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
3. Errors in proofs.
   Did anyone write full proofs?
   Did anyone check all details?
4. Quantitative problems.
   e.g. 2016 Bhargavan–Leurent
   sweet32.info: Triple-DES
   broken in TLS; PRP-PRF switch
too weak for 64-bit block ciphers.
5. Is TEA PRP-secure?

One-time pad has complete proof of privacy, but key must be
as long as total of all messages.
Many things can go wrong here:
1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
3. Errors in proofs.
   Did anyone write full proofs?
   Did anyone check all details?
4. Quantitative problems.
   e.g. 2016 Bhargavan–Leurent
   sweet32.info: Triple-DES
   broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.
5. Is TEA PRP-secure?
Many things can go wrong here:
1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
3. Errors in proofs. Did anyone write full proofs? Did anyone check all details?
4. Quantitative problems. e.g. 2016 Bhargavan–Leurent sweet32.info: Triple-DES broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.
5. Is TEA PRP-secure?

One-time pad has complete proof of privacy, but key must be as long as total of all messages.
Many things can go wrong here:
1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
3. Errors in proofs. Did anyone write full proofs? Did anyone check all details?
4. Quantitative problems. e.g. 2016 Bhargavan–Leurent sweet32.info: Triple-DES broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.
5. **Is TEA PRP-secure?**

One-time pad has complete proof of privacy, but key must be as long as total of all messages.
Many things can go wrong here:

1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
3. Errors in proofs. Did anyone write full proofs? Did anyone check all details?
4. Quantitative problems. e.g. 2016 Bhargavan–Leurent sweet32.info: Triple-DES broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.
5. Is TEA PRP-secure?

One-time pad has complete proof of privacy, but key must be as long as total of all messages.

Wegman–Carter authenticator has complete proof of authenticity, but key length is proportional to number of messages.
Many things can go wrong here:

1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
3. Errors in proofs. Did anyone write full proofs? Did anyone check all details?
4. Quantitative problems. e.g. 2016 Bhargavan–Leurent sweet32.info: Triple-DES broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.
5. Is TEA PRP-secure?

One-time pad has complete proof of privacy, but key must be as long as total of all messages.

Wegman–Carter authenticator has complete proof of authenticity, but key length is proportional to number of messages.

Short-key cipher handling many messages: **no complete proofs.**
Many things can go wrong here:

1. Security definition too weak.

2. Internal mismatch between hypotheses and conclusions.

3. Errors in proofs.
   Did anyone write full proofs?
   Did anyone check all details?

4. Quantitative problems.
   e.g. 2016 Bhargavan–Leurent sweet32.info: Triple-DES broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.

5. Is TEA PRP-secure?

One-time pad has complete proof of privacy, but key must be as long as total of all messages.

Wegman–Carter authenticator has complete proof of authenticity, but key length is proportional to number of messages.

Short-key cipher handling many messages: no complete proofs. We conjecture security after enough failed attack efforts.

“All of these attacks fail and we don’t have better attack ideas.”
Many things can go wrong here:

1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
3. Errors in proofs. Did anyone write full proofs? Did anyone check all details?
4. Quantitative problems. e.g. 2016 Bhargavan–Leurent sweet32.info: Triple-DES broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.
5. Is TEA PRP-secure?

One-time pad has complete proof of privacy, but key must be as long as total of all messages.

Wegman–Carter authenticator has complete proof of authenticity, but key length is proportional to number of messages.

Short-key cipher handling many messages: no complete proofs.

We conjecture security after enough failed attack efforts. “All of these attacks fail and we don’t have better attack ideas.”

XORTEA:

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r++) {
        c += 0x9e3779b9;
        x ^= y ^ c ^ (y << 4) ^ k[0] ^ (y >> 5) ^ k[1];
        y ^= x ^ c ^ (x << 4) ^ k[2] ^ (x >> 5) ^ k[3];
    }
    b[0] = x; b[1] = y;
}
```
Many things can go wrong here:
1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
3. Errors in proofs. Did anyone write full proofs? Did anyone check all details?
4. Quantitative problems. e.g. 2016 Bhargavan–Leurent sweet32.info: Triple-DES broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.
5. Is TEA PRP-secure?

One-time pad has complete proof of privacy, but key must be as long as total of all messages.

Wegman–Carter authenticator has complete proof of authenticity, but key length is proportional to number of messages.

Short-key cipher handling many messages: no complete proofs.

We conjecture security after enough failed attack efforts. “All of these attacks fail and we don’t have better attack ideas.”

XORTEA: a bad cipher

```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 32;r += 1) {
        c += 0x9e3779b9;
        x ^= y^c ^ (y<<4)^k[0]
            ^ (y>>5)^k[1];
        y ^= x^c ^ (x<<4)^k[2]
            ^ (x>>5)^k[3];
    }
    b[0] = x; b[1] = y;
}
```
Many things can go wrong here:

1. Security definition too weak.
2. Internal mismatch between hypotheses and conclusions.
3. Errors in proofs. Did anyone write full proofs? Did anyone check all details?
4. Quantitative problems. e.g. 2016 Bhargavan–Leurent sweet32.info: Triple-DES broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.
5. Is TEA PRP-secure?

One-time pad has complete proof of privacy, but key must be as long as total of all messages.

Wegman–Carter authenticator has complete proof of authenticity, but key length is proportional to number of messages.

Short-key cipher handling many messages: no complete proofs.

We conjecture security after enough failed attack efforts. “All of these attacks fail and we don’t have better attack ideas.”

XORTEA: a bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x ^= y^c ^ (y<<4)^k[0]^ (y>>5)^k[1];
        y ^= x^c ^ (x<<4)^k[2]^ (x>>5)^k[3];
    }
    b[0] = x; b[1] = y;
}
```
One-time pad has complete proof of privacy, but key must be as long as total of all messages.

Wegman–Carter authenticator has complete proof of authenticity, but key length is proportional to number of messages.

Short-key cipher handling many messages: **no complete proofs**.

We conjecture security after enough failed attack efforts. “All of these attacks fail and we don’t have better attack ideas.”

---

**XORTEA: a bad cipher**

```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 32;r += 1) {
        c += 0x9e3779b9;
        x ^= y^c ^ (y<<4)^k[0]
            ^ (y>>5)^k[1];
        y ^= x^c ^ (x<<4)^k[2]
            ^ (x>>5)^k[3];
    }
    b[0] = x; b[1] = y;
}
```
One-time pad has complete proof of privacy, but key must be as long as total of all messages.

Wegman–Carter authenticator has complete proof of authenticity, but key length is proportional to number of messages.

Any cipher handling many messages: **no complete proofs**.

Lecture security through failed attack efforts. These attacks fail and we have better attack ideas.”

---

**XORTEA: a bad cipher**

```c
void encrypt(uint32 *b,uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 32;r += 1) {
        c += 0x9e3779b9;
        x ^= y^c ^ (y<<4)^k[0]
            ^ (y>>5)^k[1];
        y ^= x^c ^ (x<<4)^k[2]
            ^ (x>>5)^k[3];
    }
    b[0] = x; b[1] = y;
}
```

“Hardware-friendlier” cipher, since xor circuit is cheaper than add.
One-time pad has complete proof of privacy, but key must be as long as total of all messages.

Wegman–Carter authenticator has complete proof of authenticity, but key length is proportional to number of messages.

Short-key cipher handling many messages: no complete proofs.

We conjecture security after enough failed attack efforts.

"All of these attacks fail and we don’t have better attack ideas."

XORTEA: a bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x ^= y ^ c ^ (y << 4) ^ k[0] ^ (y >> 5) ^ k[1];
        y ^= x ^ c ^ (x << 4) ^ k[2] ^ (x >> 5) ^ k[3];
    }
    b[0] = x; b[1] = y;
}
```

“Hardware-friendlier cipher, since xor circuit is cheaper than add.”
One-time pad has complete proof of privacy, but key must be as long as total of all messages.

Wegman–Carter authenticator has complete proof of authenticity, but key length is proportional to number of messages.

Short-key cipher handling many messages: no complete proofs.

We conjecture security after enough failed attack efforts.

"All of these attacks fail and we don't have better attack ideas."

XORTEA: a bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1)
    {
        c += 0x9e3779b9;
        x ^= y^c ^ (y<<4)^k[0] ^ (y>>5)^k[1];
        y ^= x^c ^ (x<<4)^k[2] ^ (x>>5)^k[3];
    }
    b[0] = x; b[1] = y;
}
```

"Hardware-friendlier" cipher, since xor circuit is cheaper than add.
XORTEA: a bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x ^= y ^= c ^ (y << 4) ^ k[0] ^ (y >> 5) ^ k[1];
        y ^= x ^= c ^ (x << 4) ^ k[2] ^ (x >> 5) ^ k[3];
    }
    b[0] = x; b[1] = y;
}
```

“Hardware-friendlier” cipher, since xor circuit is cheaper than add.
XORTEA: a bad cipher

void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x ^= y ^ c ^ (y << 4) ^ k[0] ^ (y >> 5) ^ k[1];
        y ^= x ^ c ^ (x << 4) ^ k[2] ^ (x >> 5) ^ k[3];
    }
    b[0] = x; b[1] = y;
}

“Hardware-friendlier” cipher, since xor circuit is cheaper than add.
But output bits are linear functions of input bits!
XORTEA: a bad cipher

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x ^= y^c ^ (y<<4)^k[0]
            ^ (y>>5)^k[1];
        y ^= x^c ^ (x<<4)^k[2]
            ^ (x>>5)^k[3];
    }
    b[0] = x; b[1] = y;
}
```

“Hardware-friendlier” cipher, since xor circuit is cheaper than add.

But output bits are linear functions of input bits!

e.g. First output bit is

\[
1 \oplus k_0 \oplus k_1 \oplus k_3 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{20} \oplus k_{21} \oplus k_{30} \oplus k_{32} \oplus k_{33} \oplus k_{35} \oplus k_{42} \oplus k_{43} \oplus k_{44} \oplus k_{52} \oplus k_{53} \oplus k_{62} \oplus k_{64} \oplus k_{67} \oplus k_{69} \oplus k_{76} \oplus k_{85} \oplus k_{94} \oplus k_{96} \oplus k_{99} \oplus k_{101} \oplus k_{108} \oplus k_{117} \oplus k_{126} \oplus b_1 \oplus b_3 \oplus b_{10} \oplus b_{12} \oplus b_{21} \oplus b_{23} \oplus b_{32} \oplus b_{33} \oplus b_{35} \oplus b_{37} \oplus b_{39} \oplus b_{42} \oplus b_{43} \oplus b_{44} \oplus b_{47} \oplus b_{52} \oplus b_{53} \oplus b_{57} \oplus b_{62}.
\]
A: a bad cipher

crypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x ^= y ^ c ^ (y << 4) ^ k[0] ^ (y >> 5) ^ k[1];
        y ^= x ^ c ^ (x << 4) ^ k[2] ^ (x >> 5) ^ k[3];
    }
    b[0] = x; b[1] = y;
}

"Hardware-friendlier" cipher, since xor circuit is cheaper than add.

But output bits are linear functions of input bits!

e.g. First output bit is

1 ⊕ k_0 ⊕ k_1 ⊕ k_3 ⊕ k_{10} ⊕ k_{11} ⊕ k_{12} ⊕ k_{20} ⊕ k_{21} ⊕ k_{30} ⊕ k_{32} ⊕ k_{33} ⊕ k_{35} ⊕ k_{42} ⊕ k_{43} ⊕ k_{44} ⊕ k_{52} ⊕ k_{53} ⊕ k_{62} ⊕ k_{64} ⊕ k_{67} ⊕ k_{69} ⊕ k_{76} ⊕ k_{85} ⊕ k_{94} ⊕ k_{96} ⊕ k_{99} ⊕ k_{101} ⊕ k_{108} ⊕ k_{117} ⊕ k_{126} ⊕ b_1 ⊕ b_3 ⊕ b_{10} ⊕ b_{12} ⊕ b_{21} ⊕ b_{30} ⊕ b_{32} ⊕ b_{33} ⊕ b_{35} ⊕ b_{37} ⊕ b_{39} ⊕ b_{42} ⊕ b_{43} ⊕ b_{44} ⊕ b_{47} ⊕ b_{52} ⊕ b_{53} ⊕ b_{57} ⊕ b_{62}.

There is a matrix $M$ with coefficients in $F_2$ such that

$\text{XORTEA}_k(b) = (1 ; k ; b) M$. 
XORTEA: a bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x ^= y ^ c ^ (y << 4) ^ k[0] ^ (y >> 5) ^ k[1];
        y ^= x ^ c ^ (x << 4) ^ k[2] ^ (x >> 5) ^ k[3];
    }
    b[0] = x; b[1] = y;
}
```

“Hardware-friendlier” cipher, since xor circuit is cheaper than add.

But output bits are linear functions of input bits!

e.g. First output bit is

\[
1 \oplus k_0 \oplus k_1 \oplus k_3 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{20} \oplus k_{21} \oplus k_{30} \oplus k_{32} \oplus k_{33} \oplus k_{35} \oplus k_{42} \oplus k_{43} \oplus k_{44} \oplus k_{52} \oplus k_{53} \oplus k_{62} \oplus k_{64} \oplus k_{67} \oplus k_{69} \oplus k_{76} \oplus k_{85} \oplus k_{94} \oplus k_{96} \oplus k_{99} \oplus k_{101} \oplus k_{108} \oplus k_{117} \oplus k_{126} \oplus b_1 \oplus b_3 \oplus b_{10} \oplus b_{12} \oplus b_{21} \oplus b_{30} \oplus b_{32} \oplus b_{33} \oplus b_{35} \oplus b_{37} \oplus b_{39} \oplus b_{42} \oplus b_{43} \oplus b_{44} \oplus b_{47} \oplus b_{52} \oplus b_{53} \oplus b_{57} \oplus b_{62}.
\]

There is a matrix $M$ with coefficients in $\mathbb{F}_2$ such that, for all $(k; b)$,

\[
\text{XORTEA}_k(b) = (1; k; b) M.
\]
“Hardware-friendlier” cipher, since xor circuit is cheaper than add.

But output bits are linear functions of input bits!

e.g. First output bit is

\[ 1 \oplus k_0 \oplus k_1 \oplus k_3 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{20} \oplus k_{21} \oplus k_{30} \oplus k_{32} \oplus k_{33} \oplus k_{35} \oplus k_{42} \oplus k_{43} \oplus k_{44} \oplus k_{52} \oplus k_{53} \oplus k_{62} \oplus k_{64} \oplus k_{67} \oplus k_{69} \oplus k_{76} \oplus k_{85} \oplus k_{94} \oplus k_{96} \oplus k_{99} \oplus k_{101} \oplus k_{108} \oplus k_{117} \oplus k_{126} \oplus b_1 \oplus b_3 \oplus b_{10} \oplus b_{12} \oplus b_{21} \oplus b_{30} \oplus b_{32} \oplus b_{33} \oplus b_{35} \oplus b_{37} \oplus b_{39} \oplus b_{42} \oplus b_{43} \oplus b_{44} \oplus b_{47} \oplus b_{52} \oplus b_{53} \oplus b_{57} \oplus b_{62}. \]

There is a matrix \( M \) with coefficients in \( \mathbb{F}_2 \) such that, for all \((k, b)\),

\( \text{XORTEA}_k(b) = (1, k, b)M. \)
“Hardware-friendlier” cipher, since xor circuit is cheaper than add.

But output bits are linear functions of input bits!

e.g. First output bit is

$$1 \oplus k_0 \oplus k_1 \oplus k_3 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{20} \oplus k_{21} \oplus k_{30} \oplus k_{32} \oplus k_{33} \oplus k_{35} \oplus k_{42} \oplus k_{43} \oplus k_{44} \oplus k_{52} \oplus k_{53} \oplus k_{62} \oplus k_{64} \oplus k_{67} \oplus k_{69} \oplus k_{76} \oplus k_{85} \oplus k_{94} \oplus k_{96} \oplus k_{99} \oplus k_{101} \oplus k_{108} \oplus k_{117} \oplus k_{126} \oplus b_1 \oplus b_3 \oplus b_{10} \oplus b_{12} \oplus b_{21} \oplus b_{30} \oplus b_{32} \oplus b_{33} \oplus b_{35} \oplus b_{37} \oplus b_{39} \oplus b_{42} \oplus b_{43} \oplus b_{44} \oplus b_{47} \oplus b_{52} \oplus b_{53} \oplus b_{57} \oplus b_{62}.$$  

There is a matrix $M$ with coefficients in $\mathbf{F}_2$ such that, for all $(k, b)$, $\text{XORTEA}_k(b) = (1, k, b) M$. 
“Hardware-friendlier” cipher, since xor circuit is cheaper than add.

But output bits are linear functions of input bits!

e.g. First output bit is

\[ 1 \oplus k_0 \oplus k_1 \oplus k_3 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{20} \oplus k_{21} \oplus k_{30} \oplus k_{32} \oplus k_{33} \oplus k_{35} \oplus k_{42} \oplus k_{43} \oplus k_{44} \oplus k_{52} \oplus k_{53} \oplus k_{62} \oplus k_{64} \oplus k_{67} \oplus k_{69} \oplus k_{76} \oplus k_{85} \oplus k_{94} \oplus k_{96} \oplus k_{99} \oplus k_{101} \oplus k_{108} \oplus k_{117} \oplus k_{126} \oplus b_1 \oplus b_3 \oplus b_{10} \oplus b_{12} \oplus b_{21} \oplus b_{30} \oplus b_{32} \oplus b_{33} \oplus b_{35} \oplus b_{37} \oplus b_{39} \oplus b_{42} \oplus b_{43} \oplus b_{44} \oplus b_{47} \oplus b_{52} \oplus b_{53} \oplus b_{57} \oplus b_{62}. \]

There is a matrix \( M \) with coefficients in \( F_2 \) such that, for all \((k, b)\),

\[ \text{XORTEA}_k(b) = (1, k, b)M. \]

\[ \text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = (0, 0, b_1 \oplus b_2)M. \]
"Hardware-friendlier" cipher, since xor circuit is cheaper than add.

But output bits are linear functions of input bits!

e.g. First output bit is

\[ \begin{align*}
1 & \oplus k_0 \oplus k_1 \oplus k_3 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus \\
k_20 & \oplus k_{21} \oplus k_{30} \oplus k_{32} \oplus k_{33} \oplus k_{35} \oplus \\
k_{42} & \oplus k_{43} \oplus k_{44} \oplus k_{52} \oplus k_{53} \oplus k_{62} \oplus \\
k_{64} & \oplus k_{67} \oplus k_{69} \oplus k_{76} \oplus k_{85} \oplus k_{94} \oplus \\
k_{96} & \oplus k_{99} \oplus k_{101} \oplus k_{108} \oplus k_{117} \oplus k_{126} \oplus \\
b_1 & \oplus b_3 \oplus b_{10} \oplus b_{12} \oplus b_{21} \oplus b_{30} \oplus b_{32} \oplus \\
b_{33} & \oplus b_{35} \oplus b_{37} \oplus b_{39} \oplus b_{42} \oplus b_{43} \oplus \\
b_{44} & \oplus b_{47} \oplus b_{52} \oplus b_{53} \oplus b_{57} \oplus b_{62}.
\end{align*} \]

There is a matrix \( M \) with coefficients in \( \mathbb{F}_2 \) such that, for all \((k, b)\),

\[ \text{XORTEA}_k(b) = (1, k, b)M. \]

\[ \text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = (0, 0, b_1 \oplus b_2)M. \]

Very fast attack:

if \( b_4 = b_1 \oplus b_2 \oplus b_3 \) then

\[ \text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = \text{XORTEA}_k(b_3) \oplus \text{XORTEA}_k(b_4). \]
“Hardware-friendlier” cipher, since xor circuit is cheaper than add.
But output bits are linear functions of input bits!

e.g. First output bit is
\[1 \oplus k_0 \oplus k_1 \oplus k_3 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus \]
\[k_{20} \oplus k_{21} \oplus k_{30} \oplus k_{32} \oplus k_{33} \oplus k_{35} \oplus \]
\[k_{42} \oplus k_{43} \oplus k_{44} \oplus k_{52} \oplus k_{53} \oplus k_{62} \oplus \]
\[k_{64} \oplus k_{67} \oplus k_{69} \oplus k_{76} \oplus k_{85} \oplus k_{94} \oplus \]
\[k_{96} \oplus k_{99} \oplus k_{101} \oplus k_{108} \oplus k_{117} \oplus k_{126} \oplus \]
\[b_1 \oplus b_3 \oplus b_{10} \oplus b_{12} \oplus b_{21} \oplus b_{30} \oplus b_{32} \oplus \]
\[b_{33} \oplus b_{35} \oplus b_{37} \oplus b_{39} \oplus b_{42} \oplus b_{43} \oplus \]
\[b_{44} \oplus b_{47} \oplus b_{52} \oplus b_{53} \oplus b_{57} \oplus b_{62}.\]

There is a matrix \( M \) with coefficients in \( F_2 \) such that, for all \((k, b)\),
\[\text{XORTEA}_k(b) = (1, k, b)M.\]

\[\text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = (0, 0, b_1 \oplus b_2)M.\]

Very fast attack:
if \(b_4 = b_1 \oplus b_2 \oplus b_3\) then
\[\text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = \text{XORTEA}_k(b_3) \oplus \text{XORTEA}_k(b_4).\]

This breaks PRP (and PRF):
uniform random permutation (or function) \(F\) almost never has
\[F(b_1) \oplus F(b_2) = F(b_3) \oplus F(b_4).\]
“Hardware-friendlier” cipher, since xor circuit is cheaper than add.

But output bits are linear functions of input bits!

But output bit is
\[ k_1 \oplus k_3 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus \]
\[ k_1 \oplus k_{30} \oplus k_{32} \oplus k_{33} \oplus k_{35} \oplus \]
\[ k_3 \oplus k_{44} \oplus k_{52} \oplus k_{53} \oplus k_{62} \oplus \]
\[ k_7 \oplus k_{69} \oplus k_{76} \oplus k_{85} \oplus k_{94} \oplus \]
\[ k_{101} \oplus k_{108} \oplus k_{117} \oplus k_{126} \oplus \]
\[ b_{10} \oplus b_{12} \oplus b_{21} \oplus b_{30} \oplus b_{32} \oplus \]
\[ b_{35} \oplus b_{37} \oplus b_{39} \oplus b_{42} \oplus b_{43} \oplus \]
\[ b_{57} \oplus b_{52} \oplus b_{53} \oplus b_{57} \oplus b_{62}. \]

There is a matrix \( M \) with coefficients in \( F_2 \) such that, for all \((k, b)\),
\[ \text{XORTEA}_k(b) = (1, k, b)M. \]

\[ \text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = (0, 0, b_1 \oplus b_2)M. \]

Very fast attack:
if \( b_4 = b_1 \oplus b_2 \oplus b_3 \) then
\[ \text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = \text{XORTEA}_k(b_3) \oplus \text{XORTEA}_k(b_4). \]

This breaks PRP (and PRF):
uniform random permutation (or function) \( F \) almost never has
\[ F(b_1) \oplus F(b_2) = F(b_3) \oplus F(b_4). \]

LEFTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
             ^ (y << 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
             ^ (x << 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```
There is a matrix $M$ with coefficients in $\mathbb{F}_2$ such that, for all $(k, b)$, 
\[
\text{XORTEA}_k(b) = (1, k, b)M.
\]

\[
\text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = (0, 0, b_1 \oplus b_2)M.
\]

Very fast attack:
if $b_4 = b_1 \oplus b_2 \oplus b_3$ then
\[
\text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = \text{XORTEA}_k(b_3) \oplus \text{XORTEA}_k(b_4).
\]

This breaks PRP (and PRF):
uniform random permutation (or function) $F$ almost never has
\[
F(b_1) \oplus F(b_2) = F(b_3) \oplus F(b_4).
\]
There is a matrix $M$ with coefficients in $\mathbb{F}_2$ such that, for all $(k, b)$,
\[ \text{XORTEA}_k(b) = (1, k, b)M. \]

\[ \text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = (0, 0, b_1 \oplus b_2)M. \]

Very fast attack:
if $b_4 = b_1 \oplus b_2 \oplus b_3$ then
\[ \text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = \text{XORTEA}_k(b_3) \oplus \text{XORTEA}_k(b_4). \]

This breaks PRP (and PRF):
uniform random permutation (or function) $F$ almost never has
\[ F(b_1) \oplus F(b_2) = F(b_3) \oplus F(b_4). \]
There is a matrix $M$ with coefficients in $\mathbb{F}_2$ such that, for all $(k, b)$,

$$\text{XORTEA}_k(b) = (1, k, b)M.$$ 

$$\text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = (0, 0, b_1 \oplus b_2)M.$$ 

Very fast attack: if $b_4 = b_1 \oplus b_2 \oplus b_3$ then

$$\text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = \text{XORTEA}_k(b_3) \oplus \text{XORTEA}_k(b_4).$$

This breaks PRP (and PRF): uniform random permutation (or function) $F$ almost never has

$$F(b_1) \oplus F(b_2) = F(b_3) \oplus F(b_4).$$

---

LEFTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0] ^ (y<<5)+k[1];
        y += x+c ^ (x<<4)+k[2] ^ (x<<5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```
There is a matrix $M$ with coefficients in $\mathbb{F}_2$ such that, for all $(k, b)$,

$$A_k(b) = (1, k, b)M.$$  

$$A_k(b_1) \oplus \text{XORTEA}_k(b_2) = (b_1 \oplus b_2)M.$$  

Very fast attack:

If $b_4 = b_1 \oplus b_2 \oplus b_3$ then

$$A_k(b_1) \oplus \text{XORTEA}_k(b_2) = A_k(b_3) \oplus \text{XORTEA}_k(b_4).$$

Freaks PRP (and PRF): uniform random permutation (or function) $F$ almost never has

$$F(b_2) = F(b_3) \oplus F(b_4).$$

---

**LEFTEA: another bad cipher**

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y+c \oplus (y<<4)+k[0] \oplus (y<<5)+k[1];
        y += x+c \oplus (x<<4)+k[2] \oplus (x<<5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

Addition is not $\mathbb{F}_2$-linear, but addition mod 2 is $\mathbb{F}_2$-linear.

First output bit is $1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}$. 

There is a matrix $M$ with coefficients in $\mathbb{F}_2$ such that, for all $(k, b)$,

$\text{XORTEA}_k(b) = (1^k; b) M$.

$\text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = (0; 0; b_1 \oplus b_2) M$.

Very fast attack: if $b_4 = b_1 \oplus b_2 \oplus b_3$ then

$\text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = \text{XORTEA}_k(b_3)$.

(And PRF):

Uniform random permutation (or function) $F$ almost never has

$F(b_1) \oplus F(b_2) = F(b_3) \oplus F(b_4)$.

Addition is not $\mathbb{F}_2$-linear,
but addition mod 2 is $\mathbb{F}_2$-linear.

First output bit is

$1 \oplus k_0 \oplus k_32 \oplus k_64 \oplus b_3$.

---

**LEFTEA: another bad cipher**

```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 32;r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0]
             ^ (y<<5)+k[1];
        y += x+c ^ (x<<4)+k[2]
             ^ (x<<5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

---
There is a matrix $M$ with coefficients in $F_2$ such that, for all $(k; b)$,

$\text{XORTEA}_k(b) = (1; k; b) M$...

Very fast attack: if $b_4 = b_1 \oplus b_2 \oplus b_3$ then

$\text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = \text{XORTEA}_k(b_3) \oplus \text{XORTEA}_k(b_4)$.

Addition is not $F_2$-linear, but addition mod 2 is $F_2$-linear.

First output bit is

$1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}$.

---

**LEFTEA: another bad cipher**

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0]
            ^ (y<<5)+k[1];
        y += x+c ^ (x<<4)+k[2]
            ^ (x<<5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```
LEFTEA: another bad cipher

void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y << 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x << 5) + k[3];
    }
    b[0] = x; b[1] = y;
}

Addition is not $\mathbb{F}_2$-linear, but addition mod 2 is $\mathbb{F}_2$-linear. First output bit is $1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}$. 

LEFTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y << 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x << 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Addition is not $F_2$-linear, but addition mod 2 is $F_2$-linear.

First output bit is

$$1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}.$$  

Higher output bits are increasingly nonlinear but they never affect first bit.
LEFTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y << 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x << 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Addition is not $F_2$-linear, but addition mod 2 is $F_2$-linear.

First output bit is

$$1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}.$$ 

Higher output bits are increasingly nonlinear but they never affect first bit.

How TEA avoids this problem: $\gg 5$ **diffuses** nonlinear changes from high bits to low bits.
LEFTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y << 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x << 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Addition is not $\mathbb{F}_2$-linear, but addition mod 2 is $\mathbb{F}_2$-linear.

First output bit is
\[ 1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}. \]

Higher output bits are increasingly nonlinear but they never affect first bit.

How TEA avoids this problem: $\gg 5$ diffuses nonlinear changes from high bits to low bits.

(Diffusion from low bits to high bits: $\ll 4$; carries in addition.)
LEFTEA: another bad cipher

crypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0]
            ^ (y<<5)+k[1];
        y += x+c ^ (x<<4)+k[2]
            ^ (x<<5)+k[3];
    }
    b[0] = x; b[1] = y;
}

Addition is not $\mathbb{F}_2$-linear, but addition mod 2 is $\mathbb{F}_2$-linear.

First output bit is
\[
1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}.
\]

Higher output bits
are increasingly nonlinear
but they never affect first bit.

How TEA avoids this problem:
$\ll 5$ **diffuses** nonlinear changes
from high bits to low bits.

(Diffusion from low bits to high bits: $\ll 4$; carries in addition.)

TEA4: another bad cipher

void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 4; r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<5)+k[1]
            ^ (y>>5)+k[0];
        y += x+c ^ (x<<5)+k[3]
            ^ (x<<4)+k[2];
    }
    b[0] = x; b[1] = y;
}
Addition is not $\mathbb{F}_2$-linear, but addition mod 2 is $\mathbb{F}_2$-linear.

First output bit is
$$1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}.$$  

Higher output bits are increasingly nonlinear but they never affect first bit.

How TEA avoids this problem: $\gg 5$ diffuses nonlinear changes from high bits to low bits.  

(Diffusion from low bits to high bits: $\ll 4$; carries in addition.)
Addition is not $\mathbf{F}_2$-linear, but addition mod 2 is $\mathbf{F}_2$-linear.

First output bit is
$$1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}.$$ 

Higher output bits are increasingly nonlinear but they never affect first bit.

How TEA avoids this problem: $\gg 5$ **diffuses** nonlinear changes from high bits to low bits.

(Diffusion from low bits to high bits: $\ll 4$; carries in addition.)
Addition is not $\mathbb{F}_2$-linear, but addition mod 2 is $\mathbb{F}_2$-linear.

First output bit is

$$1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}.$$  

Higher output bits are increasingly nonlinear but they never affect first bit.

How TEA avoids this problem: $\gg 5$ **diffuses** nonlinear changes from high bits to low bits.

(Diffusion from low bits to high bits: $<<4$; carries in addition.)

---

**TEA4: another bad cipher**

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 4; r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0]
            ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2]
            ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```
Addition is not $\mathbb{F}_2$-linear, but addition mod 2 is $\mathbb{F}_2$-linear.

First output bit is $1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}$.

Higher output bits are increasingly nonlinear but they never affect first bit.

TEA avoids this problem: $\ll 5$ diffuses nonlinear changes from high bits to low bits.

(Diffusion from low bits to high bits: $\gg 4$; carries in addition.)

---

TEA4: another bad cipher

```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 4;r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0] ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2] ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

---

Fast attack:

$\text{TEA}_k(x+2^{31};y)$ and $\text{TEA}_k(x;y)$ have same first bit.
Addition is not \( F_2 \)-linear, but addition mod 2 is \( F_2 \)-linear.

First output bit is:
\[ 1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus \ldots \]

Higher output bits are increasingly nonlinear but they never affect first bit.

How TEA avoids this problem:
-\( \gg 5 \) diffuses nonlinear changes from high bits to low bits.
-\( \ll 4 \) carries in addition from low bits to high bits.

Fast attack:
\( \text{TEA}_k(x + 2^{31}, y) \) and \( \text{TEA}_k(x, y) \) have same first bit.

---

**TEA4: another bad cipher**

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 4; r += 1) {
        c += 0x9e3779b9;
        x += y+c \( \oplus \) (y<<4)+k[0] \( \oplus \) (y>>5)+k[1];
        y += x+c \( \oplus \) (x<<4)+k[2] \( \oplus \) (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```
Addition is not $F_2$-linear, but addition mod 2 is $F_2$-linear. First output bit is $1 \oplus k_0 \oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus \ldots$ nonlinear changes from high bits to low bits. (Diffusion from low bits to high bits: $\ll 4$; carries in addition.)

---

**TEA4: another bad cipher**

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 4; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Fast attack:

$TEA4_k(x + 2^{31}, y)$ and $TEA4_k(x, y)$ have same first bit.
TEA4: another bad cipher

void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 4; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
             ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
             ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}

Fast attack:
TEA4_k(x + 2^{31}, y) and
TEA4_k(x, y) have same first bit.
TEA4: another bad cipher

void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 4; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}

Fast attack:
TEA4_k(x + 2^{31}, y) and
TEA4_k(x, y) have same first bit.

Trace x, y differences
through steps in computation.

\( r = 0 \): multiples of 2^{31}, 2^{26}.
\( r = 1 \): multiples of 2^{21}, 2^{16}.
\( r = 2 \): multiples of 2^{11}, 2^{6}.
\( r = 3 \): multiples of 2^{1}, 2^{0}. 
TEA4: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 4; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Fast attack:
TEA4k(x + 2^{31}, y) and TEA4k(x, y) have same first bit.

Trace x, y differences through steps in computation.
r = 0: multiples of 2^{31}, 2^{26}.
r = 1: multiples of 2^{21}, 2^{16}.
r = 2: multiples of 2^{11}, 2^{6}.
r = 3: multiples of 2^{1}, 2^{0}.

Uniform random function F:
F(x + 2^{31}, y) and F(x, y) have same first bit with probability 1/2.
TEA4: another bad cipher

void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 4;r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0]
            ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2]
            ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 4; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Fast attack:
TEA4_k(x + 2^{31}, y) and
TEA4_k(x, y) have same first bit.

Trace x, y differences
differences through steps in computation.

- \( r = 0 \): multiples of \( 2^{31}, 2^{26} \).
- \( r = 1 \): multiples of \( 2^{21}, 2^{16} \).
- \( r = 2 \): multiples of \( 2^{11}, 2^{6} \).
- \( r = 3 \): multiples of \( 2^{1}, 2^{0} \).

Uniform random function \( F \):
\( F(x + 2^{31}, y) \) and \( F(x, y) \) have
same first bit with probability 1/2.

PRF advantage 1/2.
Two pairs \((x, y)\): advantage 3/4.

More sophisticated attacks:
trace probabilities of differences;
probabilities of linear equations;
probabilities of higher-order
\( C(x + \delta) - C(x) \).
Use algebra+statistics to exploit
non-randomness in probabilities.
Fast attack:

TEA4_k(x + 2^{31}, y) and
TEA4_k(x, y) have same first bit.

Trace x, y differences
through steps in computation.

- r = 0: multiples of 2^{31}, 2^{26}.
- r = 1: multiples of 2^{21}, 2^{16}.
- r = 2: multiples of 2^{11}, 2^6.
- r = 3: multiples of 2^1, 2^0.

Uniform random function F:

F(x + 2^{31}, y) and F(x, y) have
same first bit with probability 1/2.

PRF advantage 1/2.
Two pairs (x, y): advantage 3/4.

More sophisticated attacks:
trace probabilities of differences;
probabilities of linear equations;
probabilities of higher-order differences C(x + ‹) − C(x + ›) + C(x);
Use algebra+statistics to exploit
non-randomness in probabilities.
Fast attack:
\( \text{TEA4}_k(x + 2^{31}, y) \) and \( \text{TEA4}_k(x, y) \) have same first bit.

Trace \( x, y \) differences through steps in computation.
\( r = 0 \): multiples of \( 2^{31}, 2^{26} \).
\( r = 1 \): multiples of \( 2^{21}, 2^{16} \).
\( r = 2 \): multiples of \( 2^{11}, 2^{6} \).
\( r = 3 \): multiples of \( 2^{1}, 2^{0} \).

Uniform random function \( F \):
\( F(x + 2^{31}, y) \) and \( F(x, y) \) have same first bit with probability \( 1/2 \).

PRF advantage \( 1/2 \).
Two pairs \((x, y)\): advantage \( 3/4 \).

More sophisticated attacks:
trace probabilities of differences; probabilities of linear equations; probabilities of higher-order differences \( C(x + \delta + \epsilon) - C(x + \delta) - C(x + \epsilon) + C(x) \); etc.
Use algebra+statistics to exploit non-randomness in probability.
Fast attack:
$\text{TEA}_4^k(x + 2^{31}, y)$ and $\text{TEA}_4^k(x, y)$ have same first bit.

Trace $x, y$ differences through steps in computation.
- $r = 0$: multiples of $2^{31}, 2^{26}$.
- $r = 1$: multiples of $2^{21}, 2^{16}$.
- $r = 2$: multiples of $2^{11}, 2^6$.
- $r = 3$: multiples of $2^1, 2^0$.

Uniform random function $F$:
$F(x + 2^{31}, y)$ and $F(x, y)$ have same first bit with probability $1/2$.

PRF advantage $1/2$.
Two pairs $(x, y)$: advantage $3/4$.

More sophisticated attacks:
trace \textit{probabilities} of differences; \textit{probabilities} of linear equations; \textit{probabilities} of higher-order differences $C(x + \delta + \epsilon) - C(x + \delta) - C(x + \epsilon) + C(x)$; etc.
Use algebra+statistics to exploit non-randomness in probabilities.
Fast attack:

$TEA_4^k(x + 2^{31}, y)$ and $TEA_4^k(x, y)$ have same first bit.

Trace $x, y$ differences through steps in computation.

$r = 0$: multiples of $2^{31}, 2^{26}$.

$r = 1$: multiples of $2^{21}, 2^{16}$.

$r = 2$: multiples of $2^{11}, 2^6$.

$r = 3$: multiples of $2^1, 2^0$.

Uniform random function $F$:

$F(x + 2^{31}, y)$ and $F(x, y)$ have same first bit with probability $1/2$.

PRF advantage $1/2$.

Two pairs $(x, y)$: advantage $3/4$.

More sophisticated attacks:

trace *probabilities* of differences; probabilities of linear equations; probabilities of higher-order differences $C(x + \delta + \varepsilon) - C(x + \delta) - C(x + \varepsilon) + C(x)$; etc.

Use algebra + statistics to exploit non-randomness in probabilities.

Attacks get beyond $r = 4$ but rapidly lose effectiveness.

Very far from full TEA.
Fast attack:
TEA4_k(x + 2^{31}, y) and
TEA4_k(x, y) have same first bit.

Trace x, y differences
through steps in computation.
\( r = 0 \): multiples of 2^{31}, 2^{26}.
\( r = 1 \): multiples of 2^{21}, 2^{16}.
\( r = 2 \): multiples of 2^{11}, 2^{6}.
\( r = 3 \): multiples of 2^{1}, 2^{0}.

Uniform random function \( F \):
\( F(x + 2^{31}, y) \) and \( F(x, y) \) have
same first bit with probability 1/2.

PRF advantage 1/2.
Two pairs \((x, y)\): advantage 3/4.

More sophisticated attacks:
trace *probabilities* of differences;
probabilities of linear equations;
probabilities of higher-order
differences \( C(x + \delta + \epsilon) - C(x + \delta) - C(x + \epsilon) + C(x) \); etc.
Use algebra+statistics to exploit
non-randomness in probabilities.

Attacks get beyond \( r = 4 \)
but rapidly lose effectiveness.
Very far from full TEA.

Hard question in cipher design:
How many “rounds” are
really needed for security?
Fast attack: 
\((x + 2^{31}, y)\) and 
\((x, y)\) have same first bit.

x and y differences 
trace through steps in computation.

Multiple of 2^{31}, 2^{26}.

Multiple of 2^{21}, 2^{16}.

Multiple of 2^{11}, 2^{6}.

Multiple of 2^{1}, 2^{0}.

Uniform random function \(F\):

\((x + 2^{31}, y)\) and \(F(x, y)\) have 
same first bit with probability 1/2.

Hard question in cipher design: 
How many “rounds” are 
really needed for security?

More sophisticated attacks:

trace probabilities of differences;
probabilities of linear equations;
probabilities of higher-order differences 
\(C(x + \delta + \epsilon) - C(x + \delta) - C(x + \epsilon) + C(x)\); etc.

Use algebra+statistics to exploit 
non-randomness in probabilities.

Attacks get beyond \(r = 4\) 
but rapidly lose effectiveness.

Very far from full TEA.
Fast attack: 

TEA4k(x + 2^{31}; y) and TEA4k(x; y) have same first bit.

Trace x;y differences through steps in computation.

r = 0: multiples of 2^{31}, 2^{26}.

r = 1: multiples of 2^{21}, 2^{16}.

r = 2: multiples of 2^{11}, 2^{6}.

r = 3: multiples of 2^{1}, 2^{0}.

Uniform random function F:

F(x + 2^{31}; y) and F(x; y) have same first bit with probability \( 1/2 \).

PRF advantage \( 1 = 2 \).

Two pairs (x;y): advantage \( 3 = 4 \).

More sophisticated attacks:

trace probabilities of differences;
probabilities of linear equations;
probabilities of higher-order differences \( C(x + \delta + \epsilon) - C(x + \delta) - C(x + \epsilon) + C(x) \); etc.

Use algebra+statistics to exploit non-randomness in probabilities.

Attacks get beyond \( r = 4 \) but rapidly lose effectiveness.

Very far from full TEA.

Hard question in cipher design:

How many “rounds” are really needed for security?

REPTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0; r < 1000; r += 1) {
        x += y+c ^ (y<<4)+k[0]
            ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2]
            ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

Fast attack: TEA4 \( k \) (\( x + 2^{31};y \)) and TEA4 \( k \) (\( x;y \)) have same first bit.

Trace \( x;y \) differences through steps in computation.

\( r = 0: \) multiples of 2 \( 31;2^{26} \).

\( r = 1: \) multiples of 2 \( 21;2^{16} \).

\( r = 2: \) multiples of 2 \( 11;2^6 \).

\( r = 3: \) multiples of 2 \( 1;2^0 \).

Uniform random function \( F \):

\[ F(x + 2^{31};y) \text{ and } F(x;y) \] have same first bit with probability 1/2.

PRF advantage 1 = 2.

Two pairs (\( x;y \)): advantage 3 = 4.

More sophisticated attacks:

trace \textit{probabilities} of differences; probabilities of linear equations; probabilities of higher-order differences \( C(x + \delta + \epsilon) - C(x + \delta) - C(x + \epsilon) + C(x) \); etc.

Use algebra+statistics to exploit non-randomness in probabilities.

Attacks get beyond \( r = 4 \) but rapidly lose effectiveness.

Very far from full TEA.

Hard question in cipher design:

How many “rounds” are really needed for security?

REPTEA: another bad cipher

```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0;r < 1000;r += 1) {
        x += y+c ^ (y<<4)+k[0]^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2]^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```
More sophisticated attacks: trace *probabilities* of differences; probabilities of linear equations; probabilities of higher-order differences $C(x + \delta + \epsilon) - C(x + \delta) - C(x + \epsilon) + C(x)$; etc. Use algebra+statistics to exploit non-randomness in probabilities.

Attacks get beyond $r = 4$ but rapidly lose effectiveness.

Very far from full TEA.

Hard question in cipher design: How many “rounds” are really needed for security?

REPTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0; r < 1000; r += 1) {
        x += y+c ^ (y<<4)+k[0] ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2] ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```
More sophisticated attacks: 

- Probabilities of differences; 
- Probabilities of linear equations; 
- Probabilities of higher-order differences $C(x + \delta + \epsilon) - C(x + \epsilon) + C(x)$; etc. 
- Use algebra + statistics to exploit non-randomness in probabilities.

Attacks get beyond $r = 4$ and rapidly lose effectiveness. Very far from full TEA.

Hard question in cipher design: How many “rounds” are needed for security?

REPTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0; r < 1000; r += 1) {
        x += y+c ^ (y<<4)+k[0] ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2] ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

REPTEA $k(b) = I_{1000} k(b)$ where $I_k$ does $x+=...; y+=...$. 

More sophisticated attacks:
trace probabilities of differences;
probabilities of linear equations;
probabilities of higher-order differences
\[ C(x) + \epsilon \] - 
\[ C(x) + C(x) \]; etc.
Use algebra+statistics to exploit non-randomness in probabilities.

Attacks get beyond \( r = 4 \) but rapidly lose effectiveness.

Very far from full TEA.

Hard question in cipher design:
How many "rounds" are really needed for security?

REPTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0; r < 1000; r += 1) {
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

\[ REPTEA_k (b) = I_{1000}^k (b) \]
where \( I_k \) does \( x += \ldots; y += \ldots \).
More sophisticated attacks:
trace probabilities of differences;
probabilities of linear equations;
probabilities of higher-order differences $C(x + \langle \rangle + C(x + \langle \rangle - C(x + \langle \rangle)) + C(x);$ etc.
Use algebra+statistics to exploit non-randomness in probabilities.
Attacks get beyond $r = 4$ but rapidly lose effectiveness.
Very far from full TEA.

Hard question in cipher design:
How many "rounds" are really needed for security?

REPTEA: another bad cipher

```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0;r < 1000;r += 1) {
        x += y+c ^ (y<<4)+k[0] ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2] ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

REPTEA$_k(b) = I_k^{1000}(b)$ where $I_k$ does $x+=\ldots; y+=\ldots$. 
REPTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0; r < 1000; r += 1) {
        x += y+c ^ (y<<4)+k[0]
             ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2]
             ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

REPTEA\_k(b) = I\_k^{1000}(b)
where I\_k does x+=...; y+=....
REPTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0; r < 1000; r += 1) {
        x += y + c ^ (y << 4) + k[0]
             ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
             ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

REPTEA<sub>k</sub>(b) = \( I_{k}^{1000}(b) \)
where \( I_{k} \) does \( x += \ldots; y += \ldots \).
Try list of \( 2^{32} \) inputs \( b \).
Collect outputs REPTEA<sub>k</sub>(b).
REPTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0; r < 1000; r += 1) {
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

REPTEA_k(b) = I_k^{1000}(b)
where I_k does x+=...; y+=... .
Try list of 2^{32} inputs b.
Collect outputs REPTEA_k(b).
Good chance that some b in list also has a = I_k(b) in list. Then
REPTEA_k(a) = I_k(REPTEA_k(b)).
REPTEA: another bad cipher

void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0; r < 1000; r += 1) {
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}

REPTEA_k(b) = I_k^{1000}(b)
where I_k does x+=...;y+=... .

Try list of 2^{32} inputs b.
Collect outputs REPTEA_k(b).
Good chance that some b in list also has a = I_k(b) in list. Then
REPTEA_k(a) = I_k(REPTEA_k(b)).

For each (b, a) from list:
Try solving equations a = I_k(b),
REPTEA_k(a) = I_k(REPTEA_k(b))
to figure out k. (More equations:
try re-encrypting these outputs.)
REPTEA: another bad cipher

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0; r < 1000; r += 1) {
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Where $I_k$ does $x += \ldots; y += \ldots$.

Try list of $2^{32}$ inputs $b$.
Collect outputs $\text{REPTEA}_k(b)$.
Good chance that some $b$ in list also has $a = I_k(b)$ in list. Then
$\text{REPTEA}_k(a) = I_k(\text{REPTEA}_k(b))$.

For each $(b, a)$ from list:
Try solving equations $a = I_k(b)$, $\text{REPTEA}_k(a) = I_k(\text{REPTEA}_k(b))$ to figure out $k$. (More equations: try re-encrypting these outputs.)

This is a **slide attack**.
TEA avoids this by varying $c$. 
REPTEA: another bad cipher

code snippet:
```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0;r < 1000;r += 1) {
        x += y+c ^ (y<<4)+k[0]
            ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2]
            ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

REPTEA<sub>k</sub>(b) = \(I_k^{1000}(b)\)
where \(I_k\) does \(x+=\ldots;y+=\ldots\).

Try list of \(2^{32}\) inputs \(b\).
Collect outputs REPTEA<sub>k</sub>(b).
Good chance that some \(b\) in list also has \(a = I_k(b)\) in list. Then
REPTEA<sub>k</sub>(a) = \(I_k(\text{REPTEA}_k(b))\).

For each \((b, a)\) from list:
Try solving equations \(a = I_k(b)\),
REPTEA<sub>k</sub>(a) = \(I_k(\text{REPTEA}_k(b))\) to figure out \(k\).
(More equations: try re-encrypting these outputs.)

This is a **slide attack**.
TEA avoids this by varying \(c\).

---

What about original TEA?

code snippet:
```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 32;r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0]
            ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2]
            ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```
REPTEA\(_k(b) = I^{1000}_k(b)\)
where \(I_k\) does \(x+=\ldots;y+=\ldots\).

Try list of \(2^{32}\) inputs \(b\).
Collect outputs REPTEA\(_k(b)\).
Good chance that some \(b\) in list also has \(a = I_k(b)\) in list. Then
REPTEA\(_k(a) = I_k(\text{REPTEA}_k(b))\).

For each \((b, a)\) from list:
Try solving equations \(a = I_k(b)\),
REPTEA\(_k(a) = I_k(\text{REPTEA}_k(b))\)
to figure out \(k\). (More equations:
try re-encrypting these outputs.)

This is a **slide attack**.
TEA avoids this by varying \(c\).
REPTEA<sub>k</sub>(b) = I<sub>k</sub><sup>1000</sup>(b)
where \( I_k \) does \( x+=\ldots;y+=\ldots \).

Try list of \( 2^{32} \) inputs \( b \).
Collect outputs REPTEA<sub>k</sub>(b).
Good chance that some \( b \) in list also has \( a = I_k(b) \) in list. Then
REPTEA<sub>k</sub>(a) = I<sub>k</sub>(REPTEA<sub>k</sub>(b)).

For each \((b, a)\) from list:
Try solving equations \( a = I_k(b) \),
REPTEA<sub>k</sub>(a) = I<sub>k</sub>(REPTEA<sub>k</sub>(b)) to figure out \( k \). (More equations: try re-encrypting these outputs.)

This is a slide attack.
TEA avoids this by varying \( c \).

What about original TEA?

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0]
            ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2]
            ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```
REPTEA_k(b) = I_k^{1000}(b)
where I_k does x+=...; y+=... .

Try list of 2^{32} inputs b.
Collect outputs REPTEA_k(b).
Good chance that some b in list also has a = I_k(b) in list. Then
REPTEA_k(a) = I_k(REPTEA_k(b)).

For each (b,a) from list:
Try solving equations a = I_k(b),
REPTEA_k(a) = I_k(REPTEA_k(b)) to figure out k. (More equations: try re-encrypting these outputs.)

This is a **slide attack**.

TEA avoids this by varying c.

---

What about original TEA?

```c
void encrypt(uint32 *b,uint32 *k) {
uint32 x = b[0], y = b[1];
uint32 r, c = 0;
for (r = 0; r < 32; r += 1) {
    c += 0x9e3779b9;
    x += y+c ^ (y<<4)+k[0]
        ^ (y>>5)+k[1];
    y += x+c ^ (x<<4)+k[2]
        ^ (x>>5)+k[3];
}
b[0] = x; b[1] = y;
}
```
$A_k(b) = I_k^{1000}(b)$

$k$ does $x+=...;y+=...$.

of $2^{32}$ inputs $b$.

outputs $\text{REPTEA}_k(b)$.

chance that some $b$ in list

$a = I_k(b)$ in list. Then

$A_k(a)=I_k(\text{REPTEA}_k(b))$.

in $(b,a)$ from list:

solving equations $a = I_k(b)$,

$A_k(a)=I_k(\text{REPTEA}_k(b))$

out $k$. (More equations:

encrypting these outputs.)

a slide attack.

avoids this by varying $c$.

What about original TEA?

```c
void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 32;r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0]
        ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2]
        ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

Related keys: e.g.,

$\text{TEA}_k'(b) = \text{TEA}_k(b)$

where $(k'[0] ;k'[1] ;k'[2] ;k'[3]) =

$(k[0] + 2^{31} ;k[1] + 2^{31} ;k[2] ;k[3])$. 
What about original TEA?

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0] ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2] ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

Related keys: e.g.,

\[ \text{TEA}_{k'}(b) = \text{TEA}_k(b) \]

where \((k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])\).
What about original TEA?

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Related keys: e.g.,

$$TEA_{k'}(b) = TEA_k(b)$$

where $$(k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])$$. 

What about original TEA?

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Related keys: e.g.,

\[
\text{TEA}_{k'}(b) = \text{TEA}_k(b)
\]

where \((k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])\).
What about original TEA?

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Related keys: e.g.,

$$\text{TEA}_{k'}(b) = \text{TEA}_k(b)$$

where $$(k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])$$.

Is this an attack?
What about original TEA?

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0]
            ^ (y>>5)+k[1];
        y += x+c ^ (x<<4)+k[2]
            ^ (x>>5)+k[3];
    }
    b[0] = x; b[1] = y;
}
```

Related keys: e.g.,

\[ TEA_{k'}(b) = TEA_k(b) \]

where \((k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3]). \]

Is this an attack?

PRP attack goal: distinguish TEA\(_k\), for one secret key \(k\), from uniform random permutation.
What about original TEA?

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Related keys: e.g.,

$$\text{TEA}_{k'}(b) = \text{TEA}_k(b)$$

where $$(k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3]).$$

Is this an attack?

PRP attack goal: distinguish $\text{TEA}_k$, for one secret key $k$, from uniform random permutation.

Brute-force attack:
Guess key $g$, see if $\text{TEA}_g$ matches $\text{TEA}_k$ on some outputs.
What about original TEA?

```c
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
              ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
              ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Related keys: e.g.,

$$\text{TEA}_k'(b) = \text{TEA}_k(b)$$

where

$$(k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3]).$$

Is this an attack?

PRP attack goal: distinguish

$\text{TEA}_k$, for one secret key $k$, from uniform random permutation.

Brute-force attack:

Guess key $g$, see if $\text{TEA}_g$
matches $\text{TEA}_k$ on some outputs.

Related keys $\Rightarrow g$ succeeds with

chance $2^{-126}$. Still very small.
What about original TEA?

```c
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
```

Related keys: e.g.,

\[
\text{TEA}_{k'}(b) = \text{TEA}_k(b)
\]

where \((k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])\).

Is this an attack?

PRP attack goal: distinguish \(\text{TEA}_k\), for one secret key \(k\), from uniform random permutation.

Brute-force attack:
Guess key \(g\), see if \(\text{TEA}_g\) matches \(\text{TEA}_k\) on some outputs.

Related keys \(\Rightarrow g\) succeeds with chance \(2^{-126}\). Still very small.

1997 Kelsey–Schneier–Wagner:
Fancier relationship between \(k; k'\) has chance \(2^{-11}\) of producing a particular output equation.
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0] ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2] ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}

Related keys: e.g.,
TEA_{k'}(b) = TEA_k(b)
where (k'[0], k'[1], k'[2], k'[3]) =
(k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3]).

Is this an attack?

PRP attack goal: distinguish
TEA_k, for one secret key k, from
uniform random permutation.

Brute-force attack:
Guess key g, see if TEA_g
matches TEA_k on some outputs.

Related keys ⇒ g succeeds with
chance 2^{-126}. Still very small.

1997 Kelsey–Schneier–Wagner:
Fancier relationship between k; k' has chance 2^{-11} of producing
a particular output equation.
Related keys: e.g.,
\[ \text{TEA}_k'(b) = \text{TEA}_k(b) \]
where \((k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])\).

Is this an attack?

PRP attack goal: distinguish \(\text{TEA}_k\), for one secret key \(k\), from uniform random permutation.

Brute-force attack:
Guess key \(g\), see if \(\text{TEA}_g\) matches \(\text{TEA}_k\) on some outputs.

Related keys \(\Rightarrow g\) succeeds with chance \(2^{-126}\). Still very small.

1997 Kelsey–Schneier–Wagner:
Fancier relationship between \(k; k'\) has chance \(2^{-11}\) of producing a particular output equation.
Related keys: e.g.,
\[ \text{TEA}_{k'}(b) = \text{TEA}_k(b) \]
where \((k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])\).

Is this an attack?

PRP attack goal: distinguish \(\text{TEA}_k\), for one secret key \(k\), from uniform random permutation.

Brute-force attack:
Guess key \(g\), see if \(\text{TEA}_g\) matches \(\text{TEA}_k\) on some outputs.

Related keys \(\Rightarrow g\) succeeds with chance \(2^{-126}\). Still very small.

1997 Kelsey–Schneier–Wagner:
Fancier relationship between \(k, k'\) has chance \(2^{-11}\) of producing a particular output equation.
Related keys: e.g., $\text{TEA}_{k'}(b) = \text{TEA}_k(b)$
where $(k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])$.

Is this an attack?

PRP attack goal: distinguish $\text{TEA}_k$, for one secret key $k$, from uniform random permutation.

Brute-force attack:
Guess key $g$, see if $\text{TEA}_g$ matches $\text{TEA}_k$ on some outputs.

Related keys ⇒ $g$ succeeds with chance $2^{-126}$. Still very small.

1997 Kelsey–Schneier–Wagner:
Fancier relationship between $k, k'$ has chance $2^{-11}$ of producing a particular output equation.

No evidence in literature that this helps brute-force attack, or otherwise affects PRP security.
No challenge to security analysis of TEA-CTR-XCBC-MAC.
Related keys: e.g.,
$$\text{TEA}_{k'}(b) = \text{TEA}_k(b)$$
where $$(k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])$$.

Is this an attack?

PRP attack goal: distinguish $\text{TEA}_k$, for one secret key $k$, from uniform random permutation.

Brute-force attack:
Guess key $g$, see if $\text{TEA}_g$ matches $\text{TEA}_k$ on some outputs.

Related keys $\Rightarrow g$ succeeds with chance $2^{-126}$. Still very small.

1997 Kelsey–Schneier–Wagner: Fancier relationship between $k, k'$ has chance $2^{-11}$ of producing a particular output equation.

No evidence in literature that this helps brute-force attack, or otherwise affects PRP security.
No challenge to security analysis of TEA-CTR-XCBC-MAC.

But advertised as “related-key cryptanalysis” and claimed to justify recommendations for designers regarding key scheduling.
Related keys: e.g.,
\[ \text{TEA}_{k'}(b) = \text{TEA}_k(b) \]
where \((k'[0], k'[1], k'[2], k'[3]) = (2^{31}, k[1] + 2^{31}, k[2], k[3])\).

Is this an attack?

PRP attack goal: distinguish \(\text{TEA}_k\), for one secret key \(k\), from uniform random permutation.

Brute-force attack:
Guess key \(g\), see if \(\text{TEA}_g\) matches \(\text{TEA}_k\) on some outputs.

Related keys \(\Rightarrow g\) succeeds with chance \(2^{-126}\). Still very small.

1997 Kelsey–Schneier–Wagner: Fancier relationship between \(k, k'\) has chance \(2^{-11}\) of producing a particular output equation.

No evidence in literature that this helps brute-force attack, or otherwise affects PRP security.

No challenge to security analysis of \(\text{TEA-CTR-XCBC-MAC}\).

But advertised as “related-key cryptanalysis” and claimed to justify recommendations for designers regarding key scheduling.

Some ways to learn more about cipher attacks, hash-function attacks, etc.:
Take upcoming course “Selected areas in cryptology”.
Includes symmetric attacks.
Read attack papers, especially from FSE conference.
Try to break ciphers yourself: e.g., find attacks on FEAL.
Reasonable starting point: 2000 Schneier “Self-study course in block-cipher cryptanalysis”.
\[ \text{Related keys: e.g., } TEA_k' \left( b \right) = TEA_k \left( b \right) \]

where \((k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1], k[2], k[3])\).

Is this an attack?

PRP attack goal: distinguish secret key \(k\), from uniform random permutation.

Brute-force attack:

Guess key \(g\), see if \(\text{TEA}_g\) matches \(\text{TEA}_k\) on some outputs.

Related keys \(\Rightarrow g\) succeeds with chance \(2^{-126}\). Still very small.

1997 Kelsey–Schneier–Wagner: Fancier relationship between \(k, k'\) has chance \(2^{-11}\) of producing a particular output equation.

No evidence in literature that this helps brute-force attack, or otherwise affects PRP security.

No challenge to security analysis of \(\text{TEA-CTR-XCBC-MAC}\).

But advertised as “related-key cryptanalysis” and claimed to justify recommendations for designers regarding key scheduling.

Some ways to learn more about cipher attacks, hash-function attacks, etc.:

Take upcoming course “Selected areas in cryptology.” Includes symmetric attacks.

Read attack papers, especially from FSE conference.

Try to break ciphers yourself: e.g., find attacks on \(\text{TEA}\).

Reasonable starting point:

2000 Schneier “Self-study course in block-cipher cryptanalysis.”
1997 Kelsey–Schneier–Wagner: Fancier relationship between $k, k'$ has chance $2^{-11}$ of producing a particular output equation.

No evidence in literature that this helps brute-force attack, or otherwise affects PRP security. No challenge to security analysis of TEA-CTR-XCBC-MAC.

But advertised as “related-key cryptanalysis” and claimed to justify recommendations for designers regarding key scheduling.

Some ways to learn more about cipher attacks, hash-function attacks, etc.:

Take upcoming course “Selected areas in cryptology”. Includes symmetric attacks.

Read attack papers, especially from FSE conference.

Try to break ciphers yourself, e.g., find attacks on FEAL.

Reasonable starting point: 2000 Schneier “Self-study course in block-cipher cryptanalysis.”
1997 Kelsey–Schneier–Wagner: Fancier relationship between $k, k'$ has chance $2^{-11}$ of producing a particular output equation.

No evidence in literature that this helps brute-force attack, or otherwise affects PRP security. No challenge to security analysis of TEA-CTR-XCBC-MAC.

But advertised as “related-key cryptanalysis” and claimed to justify recommendations for designers regarding key scheduling.

Some ways to learn more about cipher attacks, hash-function attacks, etc.:

Take upcoming course “Selected areas in cryptology”. Includes symmetric attacks.

Read attack papers, especially from FSE conference. Try to break ciphers yourself: e.g., find attacks on FEAL. Reasonable starting point: 2000 Schneier “Self-study course in block-cipher cryptanalysis”.
Kelsey–Schneier–Wagner: relationship between $k, k'$ has chance $2^{-11}$ of producing a particular output equation. No evidence in literature that this helps brute-force attack, or otherwise affects PRP security. No challenge to security analysis of TEA-CTR-XCBC-MAC.

Advertised as "related-key cryptanalysis" aimed to justify recommendations for designers regarding key scheduling.

Some ways to learn more about cipher attacks, hash-function attacks, etc.:

- Take upcoming course "Selected areas in cryptology".
- Includes symmetric attacks.
- Read attack papers, especially from FSE conference.
- Try to break ciphers yourself: e.g., find attacks on FEAL.
- Reasonable starting point: 2000 Schneier "Self-study course in block-cipher cryptanalysis".

Some cipher history:
Kelsey–Schneier–Wagner: Fancier relationship between $k, k'$ has chance $2^{-11}$ of producing a particular output equation.

No evidence in literature that this helps brute-force attack, or otherwise affects PRP security.

No challenge to security analysis of TEA-CTR-XCBC-MAC.

But advertised as "related-key cryptanalysis" and claimed to justify recommendations for designers regarding key scheduling.

Some ways to learn more about cipher attacks, hash-function attacks, etc.:

Take upcoming course "Selected areas in cryptology".
Includes symmetric attacks.

Read attack papers, especially from FSE conference.
Try to break ciphers yourself: e.g., find attacks on FEAL.
Reasonable starting point: 2000 Schneier "Self-study course in block-cipher cryptanalysis".

1973, and again in 1974:
U.S. National Bureau of Standards solicits proposals for a Data Encryption Standard.
Fancier relationship between $k, k'$ has chance $2^{-11}$ of producing a particular output equation. No evidence in literature that this helps brute-force attack, or otherwise affects PRP security. No challenge to security analysis of TEA-CTR-XCBC-MAC. But advertised as "related-key cryptanalysis" and claimed to justify recommendations for designers regarding key scheduling.

Some ways to learn more about cipher attacks, hash-function attacks, etc.:

Take upcoming course "Selected areas in cryptology". Includes symmetric attacks.

Read attack papers, especially from FSE conference.

Try to break ciphers yourself: e.g., find attacks on FEAL.

Reasonable starting point:
2000 Schneier “Self-study course in block-cipher cryptanalysis".

Some cipher history
Some ways to learn more about cipher attacks, hash-function attacks, etc.:

Take upcoming course “Selected areas in cryptology”. Includes symmetric attacks.

Read attack papers, especially from FSE conference. Try to break ciphers yourself: e.g., find attacks on FEAL. Reasonable starting point: 2000 Schneier “Self-study course in block-cipher cryptanalysis”.

Some cipher history
Some ways to learn more about cipher attacks, hash-function attacks, etc.:

Take upcoming course “Selected areas in cryptology”. Includes symmetric attacks.

Read attack papers, especially from FSE conference.

Try to break ciphers yourself: e.g., find attacks on FEAL.

Reasonable starting point: 2000 Schneier “Self-study course in block-cipher cryptanalysis”.

---

Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.
Some ways to learn more about cipher attacks, hash-function attacks, etc.:

Take upcoming course “Selected areas in cryptology”. Includes symmetric attacks.

Read attack papers, especially from FSE conference. Try to break ciphers yourself: e.g., find attacks on FEAL. Reasonable starting point: 2000 Schneier “Self-study course in block-cipher cryptanalysis”.

Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”
Some ways to learn more about cipher attacks, hash-function attacks, etc.:

- Take upcoming course "Selected areas in cryptology".
- Read attack papers, especially from FSE conference.
- Try to break ciphers yourself: e.g., find attacks on FEAL.
- Reasonable starting point: 2000 Schneier "Self-study course in block-cipher cryptanalysis".

**Some cipher history**


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.
Some ways to learn more about cipher attacks, hash-function attacks, etc.:
Take upcoming course “Selected areas in cryptology”. Includes symmetric attacks.
Read attack papers, especially from FSE conference.
Try to break ciphers yourself: e.g., find attacks on FEAL.
Reasonable starting point: 2000 Schneier “Self-study course in block-cipher cryptanalysis”.

Some cipher history
1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.
1976: NSA meets Diffie and Hellman to discuss criticism.
Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.
1977: Diffie and Hellman publish detailed design of a $20000000 machine to break hundreds of DES keys per year.
Some ways to learn more about cipher attacks, hash-function attacks, etc.:

- Take upcoming course “Selected areas in cryptology”.
- Includes symmetric attacks.
- Read attack papers, especially from FSE conference.
- Try to break ciphers yourself: e.g., find attacks on FEAL.
- Reasonable starting point: 2000 Schneier “Self-study course in block-cipher cryptanalysis”.

Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20,000,000 machine to break hundreds of DES keys per year.
Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20000000 machine to break hundreds of DES keys per year.
Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20000000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.

Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $200,000,000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.

Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20000000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.
Some cipher history


NBS publishes IBM DES. 64-bit block, 56-bit key.

NSA meets Diffie and Hellman to discuss criticism.

“somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20000000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims "somewhere over $400,000,000 to break a DES key; "I don't think you can tell any Congressman what's going to be secure 25 years from now."

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20000000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes "NSA convinced IBM that a reduced key size was sufficient".


Researchers publish new cipher proposals and security analysis.

Some cipher history


1975: NBS publishes IBM DES proposal. 64-bit block, 56-bit key.

1976: NSA meets Diffie and Hellman to discuss criticism. Claims "somewhere over $400,000,000" to break a DES key; "I don't think you can tell any Congressman what's going to be secure 25 years from now."

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20000000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes "NSA convinced IBM that a reduced key size was sufficient".


Researchers publish new cipher proposals and security analysis.

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20000000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.

1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $2000000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.


1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20000000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.


1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20000000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.
1977: DES is standardized.

1977: Diffie and Hellman publish detailed design of $20000000 machine to break hundreds of DES keys per year.

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”.


Researchers publish new cipher proposals and security analysis.


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?
1977: DES is standardized.


2000: NIST, advised by NSA, selects Rijndael as AES. “Security was the most important factor in the evaluation”—Really?
1977: DES is standardized.
1977: Diffie and Hellman publish detailed design of $20000000 machine to break hundreds of DES keys per year.
1978: Congressional investigation into NSA influence concludes "NSA convinced IBM that a reduced key size was sufficient".

1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES. “Security was the most important factor in the evaluation”—Really?


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. ... Serpent appears to offer a high security margin.”


2013–now: CAESAR competition.
A.S. National Institute of Standards and Technology (formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit 28/192/256-bit key.

5 AES proposals.

EFF builds “Deep Crack” for $250000 to break hundreds of DES keys per year.

NIST selects five finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


2013–now: CAESAR competition.

Main operations in AES:
add round key to block;
apply substitution box
$x \mapsto x^{254}$ in $\mathbb{F}_{256}$ to each byte in block;
linearly mix bits across block.
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . .
Serpent appears to offer a high security margin.”

2004–2008: eSTREAM
competition for stream ciphers.


2013–now: CAESAR competition.

Main operations in AES:
add round key to block;
apply substitution $x \mapsto x^{254}$ in $\mathbb{F}_{256}$ to each byte in block;
linearly mix bits across block.
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . .
Serpent appears to offer a high security margin.”


2013–now: CAESAR competition.

Main operations in AES:
add round key to block;
apply substitution box $x \mapsto x^{254}$ in $\mathbb{F}_{256}$ to each byte in block;
linearly mix bits across block.
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


2013–now: CAESAR competition.

Main operations in AES:
add round key to block;
apply substitution box $x \mapsto x^{254}$ in $\mathbb{F}_{256}$
to each byte in block;
linearly mix bits across block.
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


2013–now: CAESAR competition.

Main operations in AES:
add round key to block;
apply substitution box $x \mapsto x^{254}$ in $\mathbb{F}_{256}$ to each byte in block;
linearly mix bits across block.

Extensive security analysis.
No serious threats to AES-256 multi-target SPRP security (which implies PRP security), even in a post-quantum world.
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


2013–now: CAESAR competition.

Main operations in AES: add round key to block; apply substitution box $x \mapsto x^{254}$ in $\mathbb{F}_{256}$ to each byte in block; linearly mix bits across block.

Extensive security analysis. No serious threats to AES-256 multi-target SPRP security (which implies PRP security), even in a post-quantum world.

So why isn’t AES-256 the end of the symmetric-crypto story?
Main operations in AES:
add round key to block;
apply substitution box
\( x \mapsto x^{254} \) in \( \mathbb{F}_{256} \)
to each byte in block;
linearly mix bits across block.

Extensive security analysis.
No serious threats to AES-256
multi-target SPRP security
(which implies PRP security),
even in a post-quantum world.

So why isn’t AES-256 the end
of the symmetric-crypto story?
Main operations in AES:
add round key to block;
apply substitution box
\( x \mapsto x^{254} \) in \( \mathbb{F}_{256} \)
to each byte in block;
linearly mix bits across block.

Extensive security analysis.
No serious threats to AES-256 multi-target SPRP security
(which implies PRP security),
even in a post-quantum world.

So why isn’t AES-256 the end
of the symmetric-crypto story?
Main operations in AES:
add round key to block;
apply substitution box
\( x \mapsto x^{254} \) in \( \mathbb{F}_{256} \)
to each byte in block;
linearly mix bits across block.

Extensive security analysis.
No serious threats to AES-256 multi-target SPRP security (which implies PRP security), even in a post-quantum world.

So why isn’t AES-256 the end of the symmetric-crypto story?
Main operations in AES:
add round key to block;
apply \textbf{substitution box} \( x \mapsto x^{254} \) in \( \mathbb{F}_{256} \) to each byte in block;
linearly mix bits across block.

Extensive security analysis.
No serious threats to AES-256 multi-target SPRP security (which implies PRP security), even in a post-quantum world.

So why isn’t AES-256 the end of the symmetric-crypto story?

---

Speeding up and strengthening HTTPS connections for Chrome on Android
April 24, 2014

Posted by Elie Bursztein, Anti-Abuse Research Lead

Earlier this year, we deployed a new TLS cipher suite in Chrome that operates three times faster than AES-GCM on devices that don’t have AES hardware acceleration, including most Android phones, wearable devices such as Google Glass and older computers.

This improves user experience, reducing latency and saving battery life by cutting down the amount of time spent encrypting and decrypting data.

To make this happen, Adam Langley, Wan-Teh Chang, Ben Laurie and I began implementing new algorithms – ChaCha 20 for symmetric encryption and Poly1305
Main operations in AES:
- Add round key to block;
- Apply substitution box $x \mapsto x^{254}$ in $\mathbb{F}_{256}$ to each byte in block;
- Linearly mix bits across block.

Extensive security analysis.
No serious threats to AES-256 target SPRP security (which implies PRP security), even in a post-quantum world.

So why isn't AES-256 the end of the symmetric-crypto story?
Main operations in AES:
- Add round key to block;
- Apply substitution box $x \mapsto x^{254}$ in $F_{256}$ to each byte in block;
- Linearly mix bits across block.

Extensive security analysis.

No serious threats to AES-256 multi-target SPRP security (which implies PRP security),
even in a post-quantum world.

So why isn't AES-256 the end of the symmetric-crypto story?
Speeding up and strengthening HTTPS connections for Chrome on Android
April 24, 2014

Posted by Elie Bursztein, Anti-Abuse Research Lead

Earlier this year, we deployed a new TLS cipher suite in Chrome that operates three times faster than AES-GCM on devices that don’t have AES hardware acceleration, including most Android phones, wearable devices such as Google Glass and older computers. This improves user experience, reducing latency and saving battery life by cutting down the amount of time spent encrypting and decrypting data.

To make this happen, Adam Langley, Wan-Teh Chang, Ben Laurie and I began implementing new algorithms – ChaCha 20 for symmetric encryption and Poly1305...
The latest news and insights from Google on security and safety on the Internet

Speeding up and strengthening HTTPS connections for Chrome on Android
April 24, 2014

Posted by Elie Bursztein, Anti-Abuse Research Lead

Earlier this year, we deployed a new TLS cipher suite in Chrome that operates three times faster than AES-GCM on devices that don’t have AES hardware acceleration, including most Android phones, wearable devices such as Google Glass and older computers. This improves user experience, reducing latency and saving battery life by cutting down the amount of time spent encrypting and decrypting data.

To make this happen, Adam Langley, Wan-Teh Chang, Ben Laurie and I began implementing new algorithms – ChaCha 20 for symmetric encryption and Poly1305 encryption.

It was officially decided to *not* allow disk encryption [1]. We’ve been working to integrate storage encryption to entry-level Android "Android Go" devices sold in developing countries, so these devices still ship with no encryption. We have to use older CPUs like ARM Cortex A13. Cryptography Extensions, making AES-XTS-256.

As we explained in detail earlier, e.g. here, finding a challenging problem due to the lack of SHA-256 and the very strict performance requirements, made suitable for practical use in dm-crypt is even more challenging. Speck, in this day and age the choice of the latter has a large political element, restricting our options.

Therefore, we (well, Paul Crowley did the initial implementation) worked on a new encryption mode, HPolyC. In essence, it’s a ChaCha stream cipher for disk encryption. More details in this blog post and the full paper here: https://eprint.iacr.org/2018/189.pdf.

Date: 2018-08-06 22:32:51
Message-ID: 20180806223300.11389-6063879884@w57757.read01.s040900.cust1-0-0
[Download message RAW]
Date: 2018-08-06 22:32:51
Message-ID: 20180806223300.113891-1-ebiggers

[Download message RAW]

From: Eric Biggers <ebiggers@google.com>

Hi all,

(Please note that this patchset is a true RFC, i.e. it is being submitted for review and we would like it to be merged quite yet!)

It was officially decided to *not* allow Android to use disk encryption [1]. We've been working to find an alternative to full disk encryption (e.g. dm-crypt) to entry-level Android devices sold in developing countries, where some devices still ship with no encryption, since they would have to use older CPUs like ARM Cortex-A7; and the native Android Cryptography Extensions, making AES-XTS much too slow.

As we explained in detail earlier, e.g. in [2], this presents a challenging problem due to the lack of encryption suite, the very strict performance requirements, while still being suitable for practical use in dm-crypt and fscrypt. To overcome this, I worked with Paul Crowley and wrote a variant of the SPECK block cipher suitable for disk encryption. The implementation works well with the native Android Cryptography Extensions, making AES-XTS much too slow.

Therefore, we (well, Paul Crowley did the real work) designed a new encryption mode, HPolyC. In essence, HPolyC makes it possible to use the ChaCha stream cipher for disk encryption. HPolyC is described in detail in the paper here: https://eprint.iacr.org/2018/720.pdf
Date: 2018-08-06 22:32:51
Message-ID: 20180806223300.113891-1-ebiggers ()

[Download message RAW]

From: Eric Biggers <ebiggers@google.com>

Hi all,

(Please note that this patchset is a true RFC, i.e. we're making it to be merged quite yet!)

It was officially decided to *not* allow Android devices to use hardware encryption [1]. We've been working to find an alternative to full disk encryption to entry-level Android devices like the "Android Go" devices sold in developing countries. Unfortunately, these devices still ship with no encryption, since for cost reasons, we have to use older CPUs like ARM Cortex-A7; and these CPUs lack hardware-based cryptography Extensions, making AES-XTS much too slow.

As we explained in detail earlier, e.g. in [2], this is a very challenging problem due to the lack of encryption algorithms that meet the very strict performance requirements, while still being suitable for practical use in dm-crypt and fscrypt. And as we mentioned, Speck, in this day and age the choice of cryptographic primitives is a large political element, restricting the options even further.

Therefore, we (well, Paul Crowley did the real work) designed a new encryption mode, HPolyC. In essence, HPolyC makes it secure to use ChaCha stream cipher for disk encryption. HPolyC is specified in the paper here: https://eprint.iacr.org/2018/720.pdf ("HPolyC:..."
From: Eric Biggers <ebiggers@google.com>

Hi all,

(Please note that this patchset is a true RFC, i.e. we're not ready for it to be merged quite yet!)

It was officially decided to *not* allow Android devices to use Speck encryption [1]. We've been working to find an alternative way to bring storage encryption to entry-level Android devices like the inexpensive "Android Go" devices sold in developing countries. Unfortunately, of these devices still ship with no encryption, since for cost reasons they have to use older CPUs like ARM Cortex-A7; and these CPUs lack the ARM Cryptography Extensions, making AES-XTS much too slow.

As we explained in detail earlier, e.g. in [2], this is a very challenging problem due to the lack of encryption algorithms that meet the very strict performance requirements, while still being secure and suitable for practical use in dm-crypt and fscrypt. And as we saw with Speck, in this day and age the choice of cryptographic primitives also has a large political element, restricting the options even further.

Therefore, we (well, Paul Crowley did the real work) designed a new encryption mode, HPolyC. In essence, HPolyC makes it secure to use the ChaCha stream cipher for disk encryption. HPolyC is specified by our paper here: https://eprint.iacr.org/2018/720.pdf ("HPolyC:
Hi all,

(Please note that this patchset is a true RFC, i.e. we're not ready for it to be merged quite yet!)

It was officially decided to *not* allow Android devices to use Speck encryption [1]. We've been working to find an alternative way to bring storage encryption to entry-level Android devices like the inexpensive "Android Go" devices sold in developing countries. Unfortunately, often these devices still ship with no encryption, since for cost reasons they have to use older CPUs like ARM Cortex-A7; and these CPUs lack the ARMv8 Cryptography Extensions, making AES-XTS much too slow.

As we explained in detail earlier, e.g. in [2], this is a very challenging problem due to the lack of encryption algorithms that meet the very strict performance requirements, while still being secure and suitable for practical use in dm-crypt and fscrypt. And as we saw with Speck, in this day and age the choice of cryptographic primitives also has a large political element, restricting the options even further.

Therefore, we (well, Paul Crowley did the real work) designed a new encryption mode, HPolyC. In essence, HPolyC makes it secure to use the ChaCha stream cipher for disk encryption. HPolyC is specified by our paper here: https://eprint.iacr.org/2018/720.pdf ("HPolvC:
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

Note that this patchset is a true RFC, i.e. we're not ready for it to be merged quite yet!

We officially decided to *not* allow Android devices to use Speck encryption [1]. We've been working to find an alternative way to bring encryption to entry-level Android devices like the inexpensive "Go" devices sold in developing countries. Unfortunately, these devices still ship with no encryption, since for cost reasons they still use older CPUs like ARM Cortex-A7; and these CPUs lack the ARMv8 cryptography Extensions, making AES-XTS much too slow.

As explained in detail earlier, e.g. in [2], this is a very pressing problem due to the lack of encryption algorithms that meet very strict performance requirements, while still being secure and suitable for practical use in dm-crypt and fscrypt. And as we saw with dm-tools in this day and age the choice of cryptographic primitives also has a large political element, restricting the options even further.

Here, we (well, Paul Crowley did the real work) designed a new encryption mode, HPolyC. In essence, HPolyC makes it secure to use the stream cipher for disk encryption. HPolyC is specified by our paper: https://eprint.iacr.org/2018/720.pdf ("HPolvC:
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

I am not yet convinced that this patchset is a true RFC, i.e. we're not ready for prime time yet!

We've decided to *not* allow Android devices to use Speck. We've been working to find an alternative way to bring dm-crypt to entry-level Android devices like the inexpensive phones sold in developing countries. Unfortunately, often these devices do not ship with no encryption, since for cost reasons they use CPUs like ARM Cortex-A7; and these CPUs lack the ARMv8 AES extensions, making AES-XTS much too slow.

In more detail earlier, e.g. in [2], this is a very common issue due to the lack of encryption algorithms that meet performance requirements, while still being secure and practical use in dm-crypt and fscrypt. And as we saw with Speck and age the choice of cryptographic primitives also becomes an element, restricting the options even further.

Overall, Paul Crowley did the real work) designed a new polyC. In essence, HPolyC makes it secure to use the dm-crypt library for disk encryption. HPolyC is specified by our eprint.iacr.org/2018/720.pdf ("HPolyC:
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

*not* allow Android devices to use Speck working to find an alternative way to bring level Android devices like the inexpensive developing countries. Unfortunately, often no encryption, since for cost reasons they ARM Cortex-A7; and these CPUs lack the ARMv8 AES-XTS much too slow.

Earlier, e.g. in [2], this is a very he lack of encryption algorithms that meet requirements, while still being secure and dm-crypt and fscrypt. And as we saw with he choice of cryptographic primitives also t, restricting the options even further.

Lowley did the real work) designed a new essence, HPolyc makes it secure to use the k encryption. HPolyc is specified by our acr.org/2018/720.pdf ("HPolyc:
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

true RFC, i.e. we're not ready for

how Android devices to use Speck
find an alternative way to bring
oid devices like the inexpensive
g countries. Unfortunately, often
ption, since for cost reasons they
-A7; and these CPUs lack the ARMv8
S much too slow.

in [2], this is a very
encryption algorithms that meet
ts, while still being secure and
and fscrypt. And as we saw with
of cryptographic primitives also
iting the options even further.

the real work) designed a new
HPolyC makes it secure to use the
on. HPolyC is specified by our
18/720.pdf ("HPolvC:
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy. AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

In [2], this is a very encryption algorithms that meet ts, while still being secure and and fscrypt. And as we saw with of cryptographic primitives also ting the options even further.

the real work) designed a new HPolyC makes it secure to use the on. HPolyC is specified by our 18/720.pdf ("HPolyC:
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits PRF security. Workarounds are hard to audit.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy. AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits PRF security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits PRF security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits PRF security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits PRF security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits PRF security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.
Skinny is better than Simon and Speck.
Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy. Software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

It is worse for high-security authenticated ciphers. 128-bit block size limits PRF security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.

Skinny is better than Simon and Speck.

Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.


Gimli permutes $\{0, 1\}^{384}$. 
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy. System is dangerous. Implementations can leak timing.

For high-security authenticated ciphers. 128-bit block size limits PRF security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.

Skinny is better than Simon and Speck.

Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.

AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy. AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits PRF security. Workarounds are hard to audit.

ChaCha creates safe systems with much less work than AES. More examples of how symmetric primitives have been improving speed, simplicity, security: PRESENT is better than DES. Skinny is better than Simon and Speck. Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.

ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.

Skinny is better than Simon and Speck.

Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.


Gimli permutes $\{0, 1\}^{384}$. 
ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.

Skinny is better than Simon and Speck.

Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.


Gimli permutes $\{0, 1\}^{384}$.

“Wait, where’s the key?”
ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.

Skinny is better than Simon and Speck.

Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.


Gimli permutes $\{0, 1\}^{384}$.

“Wait, where’s the key?”

Even–Mansour SPRP mode: $E_k(m) = k \oplus \text{Gimli}(k \oplus m)$.

Salsa/ChaCha PRF mode: $S_k(m) = (k, m) \oplus \text{Gimli}(k, m)$.

Or: $(k, 0) \oplus \text{Gimli}(k, m)$.
ChaCha creates safe systems with much less work than AES.

Examples of how symmetric primitives have been improving:

- PRESENT is better than DES.
- Skinny is better than Simon and Speck.
- BLAKE2, Ascon are better than MD5, SHA-0, SHA-256, SHA-512.


Gimli permutes \{0, 1\}^{384}.

“Wait, where’s the key?”

Even–Mansour SPRP mode:
\[ E_k(m) = k \oplus \text{Gimli}(k \oplus m). \]

Salsa/ChaCha PRF mode:
\[ S_k(m) = (k, m) \oplus \text{Gimli}(k, m). \]

Or: \((k, 0) \oplus \text{Gimli}(k, m)\).
ChaCha creates safe systems with much less work than AES. More examples of how symmetric primitives have been improving speed, simplicity, security:

- PRESENT is better than DES.
- Skinny is better than Simon and Speck.
- Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.


Gimli permutes \{0, 1\}^{384}.

“Wait, where’s the key?”

Even–Mansour SPRP mode:

\[ E_k(m) = k \oplus \text{Gimli}(k \oplus m). \]

Salsa/ChaCha PRF mode:

\[ S_k(m) = (k, m) \oplus \text{Gimli}(k, m). \]

Or: \((k, 0) \oplus \text{Gimli}(k, m)\).

```c
void gimli(uint32 *b)
{
    int r,c;
    uint32 x,y,z;
    for (r = 24;r > 0;--r) {
        for (c = 0;c < 4;++c) {
            x = rotate(b[c], 24);
            y = rotate(b[4+c], 9);
            z = b[8+c];
            b[8+c]=x^(z<<1)^((y&z)<<2);
            b[4+c]=y^x ^((x|z)<<1);
            b[ c]=z^y ^((x&y)<<3);
        }
    }
}
```
ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.

Skinny is better than Simon and Speck.

Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.


Gimli permutes $\{0, 1\}^{384}$.

“Wait, where’s the key?”

Even–Mansour SPRP mode:

$$E_k(m) = k \oplus \text{Gimli}(k \oplus m).$$

Salsa/ChaCha PRF mode:

$$S_k(m) = (k, m) \oplus \text{Gimli}(k, m).$$

Or: $(k, 0) \oplus \text{Gimli}(k, m)$.

```c
void gimli(uint32 *b)
{
    int r,c;
    uint32 x,y,z;
    for (r = 24;r > 0;--r) {
        for (c = 0;c < 4;++c) {
            x = rotate(b[ c], 24);
            y = rotate(b[4+c], 9);
            z = b[8+c];
            b[8+c]=x^(z<<1)^((y&z)<<2);
            b[4+c]=y^x ^((x|z)<<1);
            b[ c]=z^y ^((x&y)<<3);
        }
    }
}
```

Gimli permutes \{0, 1\}^{384}.

“Wait, where’s the key?”

Even–Mansour SPRP mode:
\[ E_k(m) = k \oplus \text{Gimli}(k \oplus m). \]

Salsa/ChaCha PRF mode:
\[ S_k(m) = (k, m) \oplus \text{Gimli}(k, m). \]

Or: \((k, 0) \oplus \text{Gimli}(k, m).\)

```c
void gimli(uint32 *b)
{
    int r,c;
    uint32 x,y,z;

    for (r = 24; r > 0; --r) {
        for (c = 0; c < 4; ++c) {
            x = rotate(b[c], 24);
            y = rotate(b[4+c], 9);
            z = b[8+c];
            b[8+c]=x^(z<<1)^((y&z)<<2);
            b[4+c]=y^x ^((x|z)<<1);
            b[c]=z^y ^((x&y)<<3);
        }
    }
}
```
Gimli permutes $\{0, 1\}^{384}$.

"Wait, where's the key?"

Even–Mansour SPRP mode:
$$E_k(m) = k \oplus \text{Gimli}(k \oplus m).$$

Salsa/ChaCha PRF mode:
$$S_k(m) = (k; m) \oplus \text{Gimli}(k; m).$$

Or:
$$(k; 0) \oplus \text{Gimli}(k; m).$$

```c
void gimli(uint32 *b)
{
    int r, c;
    uint32 x, y, z;

    for (r = 24; r > 0; --r) {
        for (c = 0; c < 4; ++c) {
            x = rotate(b[c], 24);
            y = rotate(b[4+c], 9);
            z = b[8+c];
            b[8+c] = x^{(z<<1)^((y&z)<<2)};
            b[4+c] = y^x ^((x|z)<<1);
            b[c] = z^y ^((x&y)<<3);
        }
    }
}
```
“Gimli: a cross-platform permutation”.

Gimli permutes \{0, 1\}^{384}.

“Wait, where’s the key?”

Even–Mansour SPRP mode:
\[ E_k(m) = k \oplus \text{Gimli}(k \oplus m). \]

Salsa/ChaCha PRF mode:
\[ S_k(m) = (k; m) \oplus \text{Gimli}(k; m). \]

Or:
\[ (k; 0) \oplus \text{Gimli}(k; m). \]

```c
void gimli(uint32 *b)
{
    int r, c;
    uint32 x, y, z;
    for (r = 24; r > 0; --r) {
        for (c = 0; c < 4; ++c) {
            x = rotate(b[c], 24);
            y = rotate(b[4+c], 9);
            z = b[8+c];
            b[8+c] = x^(z<<1)^((y&z)<<2);
            b[4+c] = y^x ^((x|z)<<1);
            b[c] = z^y ^((x&y)<<3);
        }
        if ((r & 3) == 0) {
            x = b[0]; b[0] = b[1]; b[1] = x;
        }
        if ((r & 3) == 2) {
            x = b[0]; b[0] = b[2]; b[2] = x;
        }
        if ((r & 3) == 0)
            b[0] ^= (0x9e377900 | r);
    }
}
```

Gimli permutes \{0, 1\} 384.

"Wait, where's the key?"

Even–Mansour SPRP mode:

\[ E_k(m) = k \oplus \text{Gimli}(k \oplus m). \]

Salsa/ChaCha PRF mode:

\[ S_k(m) = (k;m) \oplus \text{Gimli}(k;m). \]

Or: \((k; 0) \oplus \text{Gimli}(k;m).\)

```c
void gimli(uint32 *b)
{
    int r,c;
    uint32 x,y,z;
    for (r = 24;r > 0;--r) {
        for (c = 0;c < 4;++c) {
            x = rotate(b[c], 24);
            y = rotate(b[4+c], 9);
            z = b[8+c];
            b[8+c]=x^(z<<1)^((y&z)<<2);
            b[4+c]=y^x ^((x|z)<<1);
            b[c]=z^y ^((x&y)<<3);
        }
    }
    if ((r & 3) == 0) {
        x=b[0]; b[0]=b[1]; b[1]=x;
    }
    if ((r & 3) == 2) {
        x=b[0]; b[0]=b[2]; b[2]=x;
    }
    if ((r & 3) == 0)
        b[0] ^= (0x9e377900 | r);
}
```
void gimli(uint32 *b) {
    int r, c;
    uint32 x, y, z;

    for (r = 24; r > 0; --r) {
        for (c = 0; c < 4; ++c) {
            x = rotate(b[c], 24);
            y = rotate(b[4+c], 9);
            z = b[8+c];
            b[8+c] = x^(z<<1)^((y&z)<<2);
            b[4+c] = y^x ^((x|z)<<1);
            b[c] = z^y ^((x&y)<<3);
        }
    }

    if ((r & 3) == 0) {
        x = b[0]; b[0] = b[1]; b[1] = x;
    }

    if ((r & 3) == 2) {
        x = b[0]; b[0] = b[2]; b[2] = x;
    }

    if ((r & 3) == 0)
        b[0] ^= (0x9e377900 | r);
}
```c
void gimli(uint32 *b)
{
    int r, c;
    uint32 x, y, z;

    if ((r & 3) == 0) {
        x = b[0]; b[0] = b[1]; b[1] = x;
    }

    if ((r & 3) == 2) {
        x = b[0]; b[0] = b[2]; b[2] = x;
    }

    if ((r & 3) == 0)
        b[0] ^= (0x9e377900 | r);
}
```

No additions. Nonlinear carries are replaced by shifts of & , | .
(Idea stolen from NORX cipher.)

Big rotations diffuse changes quickly across bit positions.

\( x, y, z \) interaction diffuses changes quickly through columns \((0, 4, 8; 1, 5, 9; 2, 6, 10; 3, 7, 11)\).

Other swaps diffuse changes through rows. Deliberately limited swaps per round ⇒ faster rounds on a wide range of platforms.
void gimli(uint32 *b) {
    int r,c;
    uint32 x,y,z;
    for (r = 24;r > 0;--r) {
        for (c = 0;c < 4;++c) {
            x = rotate(b[c], 24);
            y = rotate(b[4+c], 9);
            z = b[8+c];
            b[8+c]=x^(z<<1)^((y&z)<<2);
            b[4+c]=y^x ^((x|z)<<1);
            b[c]=z^y ^((x&y)<<3);
        }
        if ((r & 3) == 0) {
            x=b[0]; b[0]=b[1]; b[1]=x;
        }
        if ((r & 3) == 2) {
            x=b[0]; b[0]=b[2]; b[2]=x;
        }
        if ((r & 3) == 0)
            b[0] ^= (0x9e377900 | r);
    }

    No additions. Nonlinear carries are replaced by shifts of & , | .
    (Idea stolen from NORX cipher.)

    Big rotations diffuse changes quickly across bit positions.
    x, y, z interaction diffuses changes quickly through columns
    (0, 4, 8; 1, 5, 9; 2, 6, 10; 3, 7, 11).

    Other swaps diffuse changes through rows. Deliberately limited
    swaps per round ⇒ faster rounds on a wide range of platforms.
}
void gimli(uint32 *b)
{
    int r,c;
    uint32 x,y,z;
    for (r = 24;r > 0;--r) {
        for (c = 0;c < 4;++c) {
            x = rotate(b[c], 24);
            y = rotate(b[4+c], 9);
            z = b[8+c];
            b[8+c]=x^(z<<1)^((y&z)<<2);
            b[4+c]=y^x ^((x|z)<<1);
            b[c]=z^y ^((x&y)<<3);
        }
        if ((r & 3) == 0) {
            x=b[0]; b[0]=b[1]; b[1]=x;
        }
        if ((r & 3) == 2) {
            x=b[0]; b[0]=b[2]; b[2]=x;
        }
        if ((r & 3) == 0)
            b[0] ^= (0x9e377900 | r);
    }
}

No additions. Nonlinear carries are replaced by shifts of & , |.
(Idea stolen from NORX cipher.)
Big rotations diffuse changes quickly across bit positions.

x, y, z interaction diffuses changes quickly through columns
(0, 4, 8; 1, 5, 9; 2, 6, 10; 3, 7, 11).

Other swaps diffuse changes through rows. Deliberately limited
swaps per round ⇒ faster rounds
on a wide range of platforms.
if ((r & 3) == 0) {
    x=b[0]; b[0]=b[1]; b[1]=x;
}
if ((r & 3) == 2) {
    x=b[0]; b[0]=b[2]; b[2]=x;
}
if ((r & 3) == 0)
    b[0] ^= (0x9e377900 | r);
}

No additions. Nonlinear carries are replaced by shifts of & , | . (Idea stolen from NORX cipher.)

Big rotations diffuse changes quickly across bit positions.

x, y, z interaction diffuses changes quickly through columns (0, 4, 8; 1, 5, 9; 2, 6, 10; 3, 7, 11).

Other swaps diffuse changes through rows. Deliberately limited swaps per round ⇒ faster rounds on a wide range of platforms.