Examples of symmetric primitives

D. J. Bernstein

	message len	tweak	key	encrypts	authenticates
Permutation	fixed	no	no	—	—
Compression function	fixed	yes	no	—	—
Block cipher	fixed	no	yes	yes	—
Tweakable block cipher	fixed	yes	yes	yes	—
Hash function	variable	no	no	—	—
MAC (without nonce)	variable	no	yes	no	yes
MAC (using nonce)	variable	yes	yes	no	yes
Stream cipher	variable	yes	yes	yes	no
Authenticated cipher	variable	yes	yes	yes	yes

es of symmetric primitives

ernstein

	message len	tweak	key	encrypts	authenticates
ation	fixed	no	no	—	—
ssion function	fixed	yes	no	—	—
ipher	fixed	no	yes	yes	—
ble block cipher	fixed	yes	yes	yes	—
nction	variable	no	no	—	—
without nonce)	variable	no	yes	no	yes
using nonce)	variable	yes	yes	no	yes
cipher	variable	yes	yes	yes	no
ticated cipher	variable	yes	yes	yes	yes

1994 Wh

a tiny en

```
void enc
{
  uint32
  uint32
  for (i
    c +=
    x +=

    y +=

  }
  b[0] =
}
```

netric primitives

	message len	tweak	key	encrypts	authenticates
	fixed	no	no	—	—
tion	fixed	yes	no	—	—
	fixed	no	yes	yes	—
cipher	fixed	yes	yes	yes	—
	variable	no	no	—	—
nce)	variable	no	yes	no	yes
e)	variable	yes	yes	no	yes
	variable	yes	yes	yes	no
her	variable	yes	yes	yes	yes

1994 Wheeler–Nee

a tiny encryption a

```
void encrypt(uin
{
  uint32 x = b[0
  uint32 r, c =
  for (r = 0;r <
    c += 0x9e377
    x += y+c ^ (
             ^ (
    y += x+c ^ (
             ^ (
  }
  b[0] = x; b[1]
}
```

itives

essage len	tweak	key	encrypts	authenticates
ed	no	no	—	—
ed	yes	no	—	—
ed	no	yes	yes	—
ed	yes	yes	yes	—
riable	no	no	—	—
riable	no	yes	no	yes
riable	yes	yes	no	yes
riable	yes	yes	yes	no
riable	yes	yes	yes	yes

1994 Wheeler–Needham "Tl
a tiny encryption algorithm"

```
void encrypt(uint32 *b,ui
{
  uint32 x = b[0], y = b[
  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;
}
```

tweak	key	encrypts	authenticates
no	no	—	—
yes	no	—	—
no	yes	yes	—
yes	yes	yes	—
no	no	—	—
no	yes	no	yes
yes	yes	no	yes
yes	yes	yes	no
yes	yes	yes	yes

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

```
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;
}
```

y	encrypts	authenticates
o	—	—
o	—	—
s	yes	—
s	yes	—
o	—	—
s	no	yes
s	no	yes
s	yes	no
s	yes	yes

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

```
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`:
represen
integer b

+: addit

c += d:

^: xor; ⊕

each bit

Lower pr

so spacin

<<4: mu

$(0, 0, 0, 0$

>>5: div

$(b_5, b_6, \ldots$

s	authenticates
	—
	—
	—
	—
	—
	yes
	yes
	no
	yes

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

```
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 (

representing the "

integer $b_0 + 2b_1 +$

+: addition mod 2

c += d: same as c

^: xor; $\oplus$; additio

each bit separately

Lower precedence

so spacing is not r

<<4: multiplicatio

$(0, 0, 0, 0, b_0, b_1, \ldots$

>>5: division by 3

$(b_5, b_6, \ldots, b_{31}, 0,$

icates

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

```
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 $b_0 + 2b_1 + \cdots + 2^{31}$

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

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

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

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

`>>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":

```
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., $(b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0)$.

heeler–Needham "TEA,
ncryption algorithm":

```
crypt(uint32 *b,uint32 *k)

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

= 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., $(b_5, b_6, \ldots, b_{31}, 0, 0, 0, 0, 0)$.

Functio

TEA is a
with a **1**

edham "TEA,
algorithm":

```
t32 *b,uint32 *k)

], y = b[1];
0;
 32;r += 1) {
9b9;
y<<4)+k[0]
y>>5)+k[1];
x<<4)+k[2]
x>>5)+k[3];

 = 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 b**
with a **128-bit ke**

EA,
:

```
nt32 *k)

[1];

 1) {

]
];
]
];
```


`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 ciph**
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**.

`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]`).

`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]`).

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

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

32 bits ($b_0, b_1, \ldots, b_{31}$)
ting the "unsigned"
$b_0 + 2b_1 + \cdots + 2^{31} b_{31}$.

ion mod 2^{32}.

same as c = c + d.

$\oplus$; addition of
separately mod 2.
recedence than + in C,
ng is not misleading.

ultiplication by 16, i.e.,
0, $b_0, b_1, \ldots, b_{27}$).

vision by 32, i.e.,
. . , 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]`).

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

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

Wait, ho

```
void enc
{
  uint32
  uint32
  for (i
    c +=
    x +=


    y +=


  }
  b[0] =
}
```

$b_0, b_1, \ldots, b_{31})$

unsigned"

$\cdots + 2^{31} b_{31}$.

^{32}.

= c + d.

n of

mod 2.

than + in C,

misleading.

n by 16, i.e.,

$\ldots, b_{27})$.

2, i.e.,

$0, 0, 0, 0)$.

<u>Functionality</u>

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

```
void encrypt(uin
{
  uint32 x = b[0
  uint32 r, c =
  for (r = 0;r <
    c += 0x9e377
    x += y+c ^ (
             ^ (
    y += x+c ^ (
             ^ (
  }
  b[0] = x; b[1]
}
```

, b_{31})

b_{31}.

C,

.

e.,

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?

```
void encrypt(uint32 *b,ui
{
  uint32 x = b[0], y = b[
  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;
}
```

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?

```
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;
}
```

nality

a **64-bit block cipher**
28-bit key.

28-bit key (namely
[1],k[2],k[3]);
laintext (b[0],b[1]).

64-bit **ciphertext**
[0],b[1]).

ciently **encrypt**:
intext) ↦ ciphertext.

ciently **decrypt**:
hertext) ↦ plaintext.

Wait, how can we decrypt?

```
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:

```
void de
{
  uint3
  uint3
  for (
    y -
    x -
    c -
  }
  b[0] 
}
```

lock cipher
y.

(namely
,k[3]);
b[0],b[1]).

phertext
.

crypt:
ciphertext.

crypt:
→ plaintext.

Wait, how can we decrypt?

```
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

```
void decrypt(uin
{
  uint32 x = b[0
  uint32 r, c =
  for (r = 0;r <
    y -= x+c ^ (
             ^ (
    x -= y+c ^ (
             ^ (
    c -= 0x9e377
  }
  b[0] = x; b[1]
}
```

er

]).

t.

t.

Wait, how can we decrypt?

```
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 inverti

```
void decrypt(uint32 *b,ui
{
  uint32 x = b[0], y = b[
  uint32 r, c = 32 * 0x9e
  for (r = 0;r < 32;r +=
    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?

```
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;
}
```

ow can we decrypt?

```
crypt(uint32 *b,uint32 *k)

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

= 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;
}
```

Generali

(used in

1973 Fei

```
x += fu
y += fu
x += fu
y += fu
...
```

Decrypti

```
...
y -= fu
x -= fu
y -= fu
x -= fu
```

decrypt?

```
t32 *b,uint32 *k)

], y = b[1];
0;
 32;r += 1) {
9b9;
y<<4)+k[0]
y>>5)+k[1];
x<<4)+k[2]
x>>5)+k[3];

 = 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, **Fe**

(used in, e.g., "Lu

1973 Feistel–Copp

```
x += function1(y
y += function2(x
x += function3(y
y += function4(x
...
```

Decryption, invert

```
...
y -= function4(x
x -= function3(y
y -= function2(x
x -= function1(y
```

```
nt32 *k)

1];

1) {

]

];

]

];
```


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 netw**

(used in, e.g., "Lucifer" fron

1973 Feistel–Coppersmith):

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

Decryption, inverting each s

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

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.

```
crypt(uint32 *b,uint32 *k)

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

= 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);
```

Higher-l

User's m

of 64-bit

o is invertible.

```
t32 *b,uint32 *k)

], y = b[1];
32 * 0x9e3779b9;
 32;r += 1) {
x<<4)+k[2]
x>>5)+k[3];
y<<4)+k[0]
y>>5)+k[1];
9b9;

 = 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);
```

Higher-level functi

User's message is

of 64-bit blocks *m*

ble.

```
nt32 *k)

[1];
3779b9;
1) {
]
];
]
];
```


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);
```

Higher-level functionality

User's message is long seque

of 64-bit blocks $m_0, m_1, m_2,$

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);
```


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 += 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);
```


Higher-level functionality

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

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

```
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);
```


Higher-level functionality

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

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

zation, **Feistel network**
, e.g., “Lucifer” from
istel–Coppersmith):

```
nction1(y,k);
nction2(x,k);
nction3(y,k);
nction4(x,k);
```

ion, inverting each step:

```
nction4(x,k);
nction3(y,k);
nction2(x,k);
nction1(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
$c_0 = m_0 \oplus \mathrm{TEA}_k(n, 0)$,
$c_1 = m_1 \oplus \mathrm{TEA}_k(n, 1)$,
$c_2 = m_2 \oplus \mathrm{TEA}_k(n, 2)$, ...
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 als
forged/r

istel network

cifer" from

ersmith):

```
,k);
,k);
,k);
,k);
```

ing each step:

```
,k);
,k);
,k);
,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
$c_0 = m_0 \oplus \text{TEA}_k(n, 0)$,
$c_1 = m_1 \oplus \text{TEA}_k(n, 1)$,
$c_2 = m_2 \oplus \text{TEA}_k(n, 2)$, ...
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
forged/modified ci

work

m

tep:

Higher-level functionality

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

TEA-CTR produces ciphertext
$c_0 = m_0 \oplus \text{TEA}_k(n, 0)$,
$c_1 = m_1 \oplus \text{TEA}_k(n, 1)$,
$c_2 = m_2 \oplus \text{TEA}_k(n, 2)$, ...
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
$c_0 = m_0 \oplus \text{TEA}_k(n, 0)$,
$c_1 = m_1 \oplus \text{TEA}_k(n, 1)$,
$c_2 = m_2 \oplus \text{TEA}_k(n, 2)$, ...
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
$c_0 = m_0 \oplus \text{TEA}_k(n, 0)$,
$c_1 = m_1 \oplus \text{TEA}_k(n, 1)$,
$c_2 = m_2 \oplus \text{TEA}_k(n, 2)$, ...
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
$c_0 = m_0 \oplus \mathrm{TEA}_k(n, 0)$,
$c_1 = m_1 \oplus \mathrm{TEA}_k(n, 1)$,
$c_2 = m_2 \oplus \mathrm{TEA}_k(n, 2)$, ...
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
$a_0 = \mathrm{TEA}_j(c_0)$,
$a_1 = \mathrm{TEA}_j(c_1 \oplus a_0)$,
$a_2 = \mathrm{TEA}_j(c_2 \oplus a_1)$, ...,
$a_{\ell-1} = \mathrm{TEA}_j(c_{\ell-1} \oplus a_{\ell-2})$,
$a_\ell = \mathrm{TEA}_j(i \oplus c_\ell \oplus a_{\ell-1})$
using 128-bit key j, 64-bit key i.
Authenticator is a_ℓ: i.e., transmit $(c_0, c_1, \ldots, c_\ell, a_\ell)$.

evel functionality

essage is long sequence
t blocks $m_0, m_1, m_2, \ldots$.

TR produces ciphertext
$\oplus$ TEA$_k(n, 0)$,
$\oplus$ TEA$_k(n, 1)$,
$\oplus$ TEA$_k(n, 2)$, ...
28-bit key k,
once n,
lock counter $0, 1, 2, \ldots$.

a **mode of operation**
verts block cipher TEA
eam 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
$a_0 = \text{TEA}_j(c_0)$,
$a_1 = \text{TEA}_j(c_1 \oplus a_0)$,
$a_2 = \text{TEA}_j(c_2 \oplus a_1)$, ...,
$a_{\ell-1} = \text{TEA}_j(c_{\ell-1} \oplus a_{\ell-2})$,
$a_\ell = \text{TEA}_j(i \oplus c_\ell \oplus a_{\ell-1})$
using 128-bit key j, 64-bit key i.
Authenticator is a_ℓ: i.e., transmit $(c_0, c_1, \ldots, c_\ell, a_\ell)$.

Specifyi
authent

320-bit
Specify
uniform

onality

long sequence

$_0, m_1, m_2, \ldots$.

es ciphertext

$n, 0)$,

$n, 1)$,

$n, 2)$, ...

k,

ter $0, 1, 2, \ldots$.

f operation

k cipher TEA

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

Specifying TEA-C

authenticated ci

320-bit key $(k, j, i$
Specify how this is
uniform random 3

ence
. . . .
ext

, ...

on
TEA
TR.

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

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

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

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

Specifying TEA-CTR-XCBC
authenticated cipher:

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

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

o wants to recognize
modified ciphertexts.

rategy:
authenticator to
ertext $c = (c_0, c_1, c_2, \ldots)$.

CBC-MAC computes
$\mathrm{EA}_j(c_0)$,
$\mathrm{EA}_j(c_1 \oplus a_0)$,
$\mathrm{EA}_j(c_2 \oplus a_1)$, . . . ,
$\mathrm{TEA}_j(c_{\ell-1} \oplus a_{\ell-2})$,
$\mathrm{A}_j(i \oplus c_\ell \oplus a_{\ell-1})$
28-bit key j, 64-bit key i.
icator is a_ℓ: 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 se

Step 1:
for auth

o recognize
iphertexts.

ator to
$= (c_0, c_1, c_2, \ldots)$.

computes

$_0)$,
$_1)$, . . . ,
$\oplus a_{\ell-2})$,
$\oplus a_{\ell-1})$
j, 64-bit key i.
$_\ell$: i.e.,
$\ldots, c_\ell, a_\ell)$.

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 sec
for authenticated c

e

$_2, \ldots)$.

ey 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".

ng TEA-CTR-XCBC-MAC
ticated 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:
"Any structure is an attack."
Hard to define clearly.
Everything seems "attackable".

Step 2:
target se
prove th
from sim

TR-XCBC-MAC

pher:

).

s chosen:

20-bit string.

sages:

ce of

t blocks.

ra work

s of bytes.)

e is chosen:

(Stateless

m 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 sett

target security def

prove that security

from simpler prope

-MAC

ng.

)

n:

.)

<u>Is this secure?</u>

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

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 $b \mapsto \text{TEA}_k(b)$.

ecure?

Define security
enticated ciphers.

not easy to do!

extreme: "It's secure
ou show me the key."
nk. Many ciphers
ntext or allow forgeries
leaking key.

useless extreme:
ructure is an attack."
define clearly.
ng seems "attackable".

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

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

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

TE

$n \mapsto$ TE

curity
ciphers.

o do!

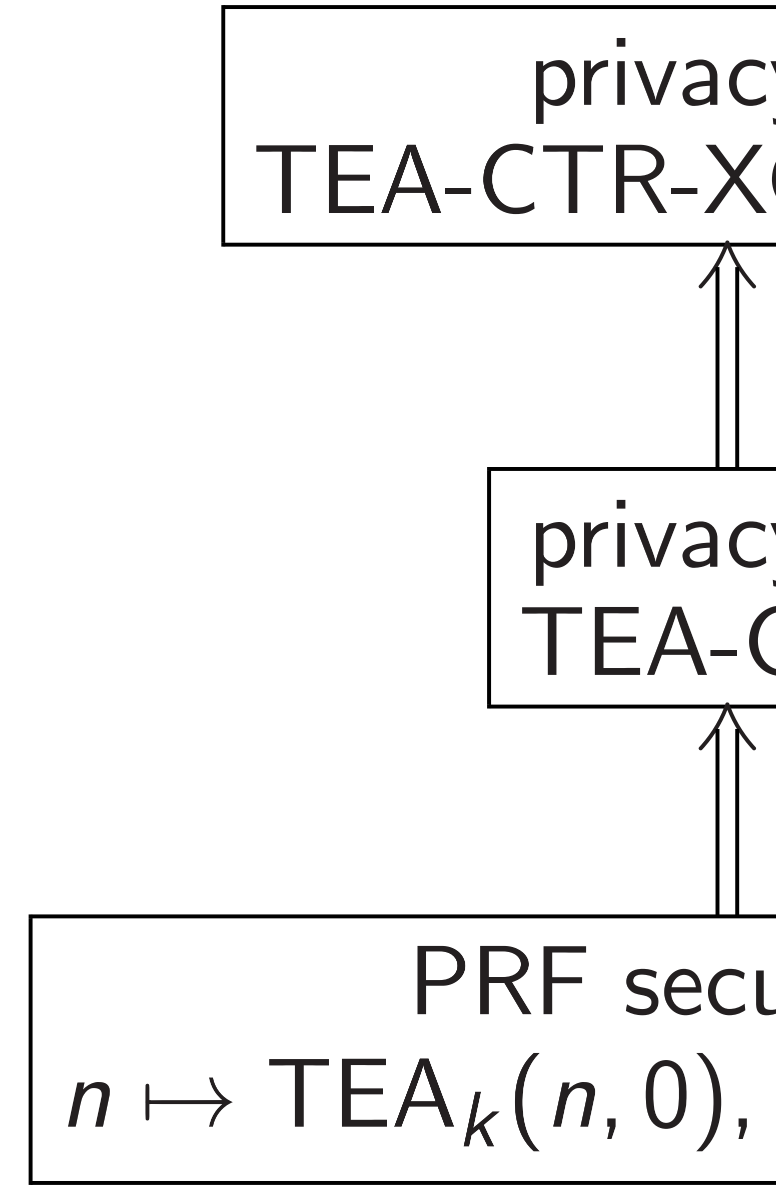
"It's secure
ne the key."
ciphers
llow forgeries
ey.

xtreme:
an attack."
arly.
"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 $b \mapsto \text{TEA}_k(b)$.

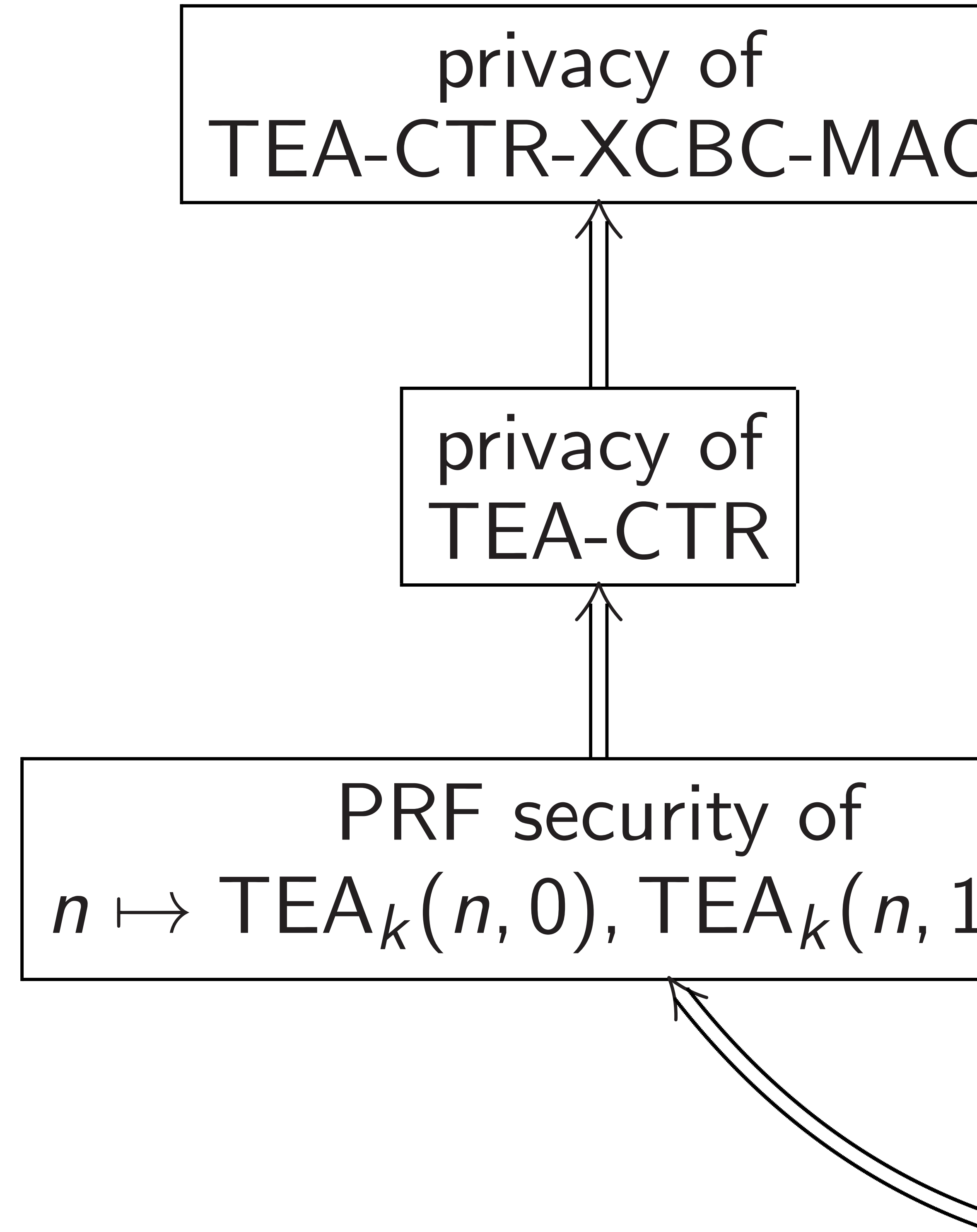


e
”
ries
”
le”.

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 $b \mapsto \text{TEA}_k(b)$.

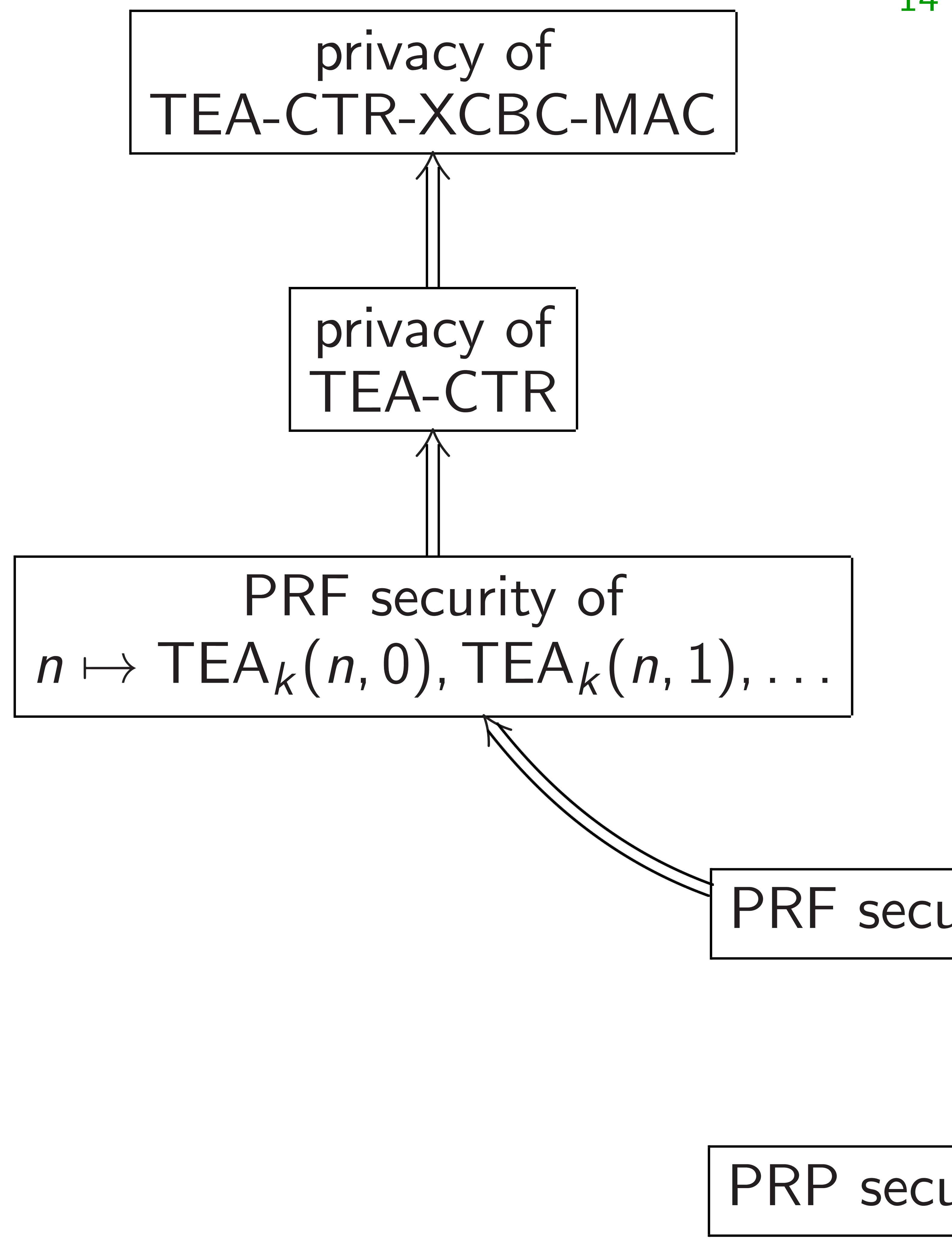
i.e. Prove that any PRF attack against $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ implies PRF attack against $b \mapsto \text{TEA}_k(b)$.



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 $b \mapsto \text{TEA}_k(b)$.

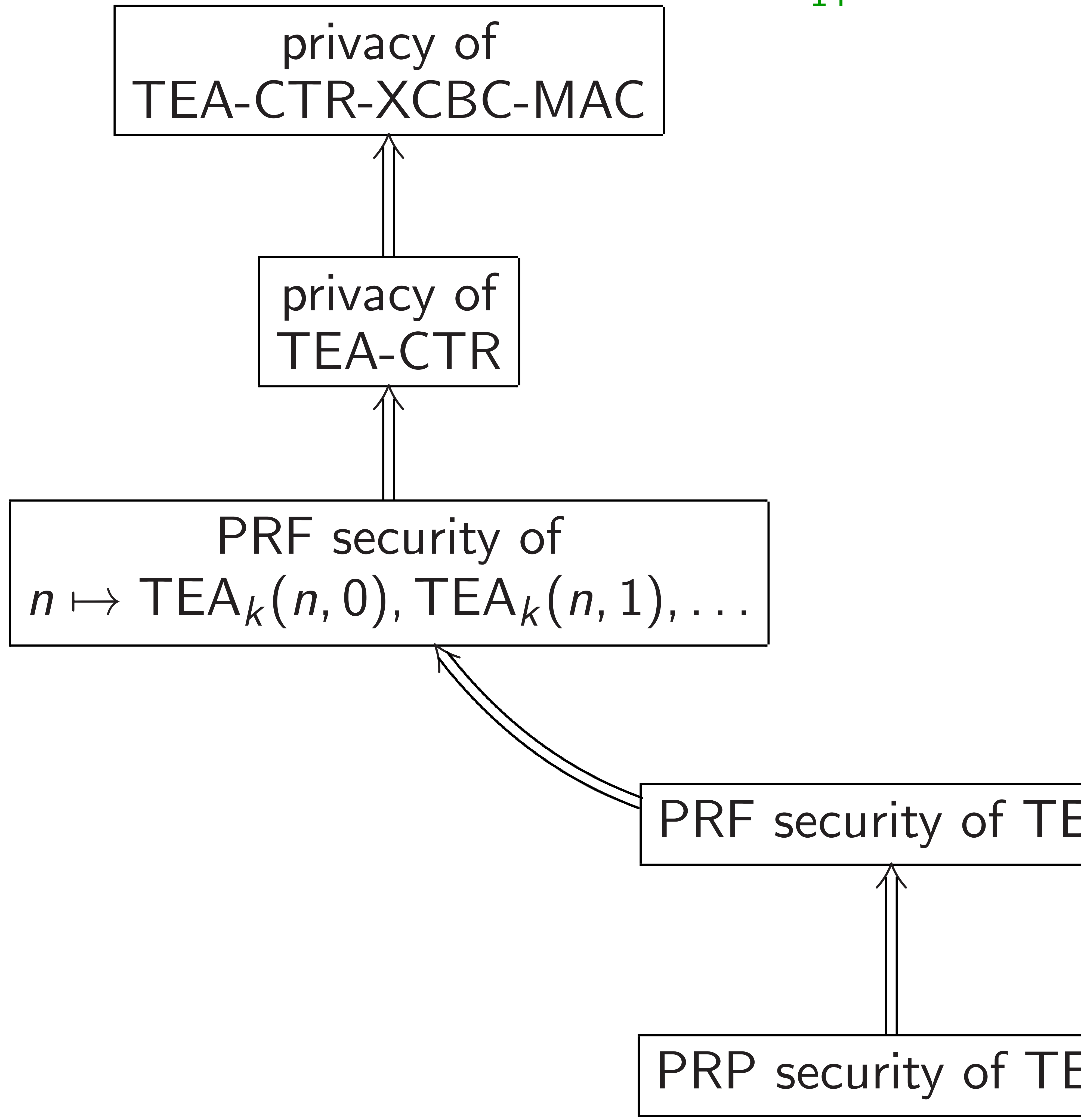
i.e. Prove that any PRF attack against $n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots$ implies PRF attack against $b \mapsto \text{TEA}_k(b)$.



After settling on
ecurity definition,
at security follows
npler properties.

ve PRF security of
$\text{A}_k(n, 0), \text{TEA}_k(n, 1), \ldots$
g PRF security of
$\text{A}_k(b)$.

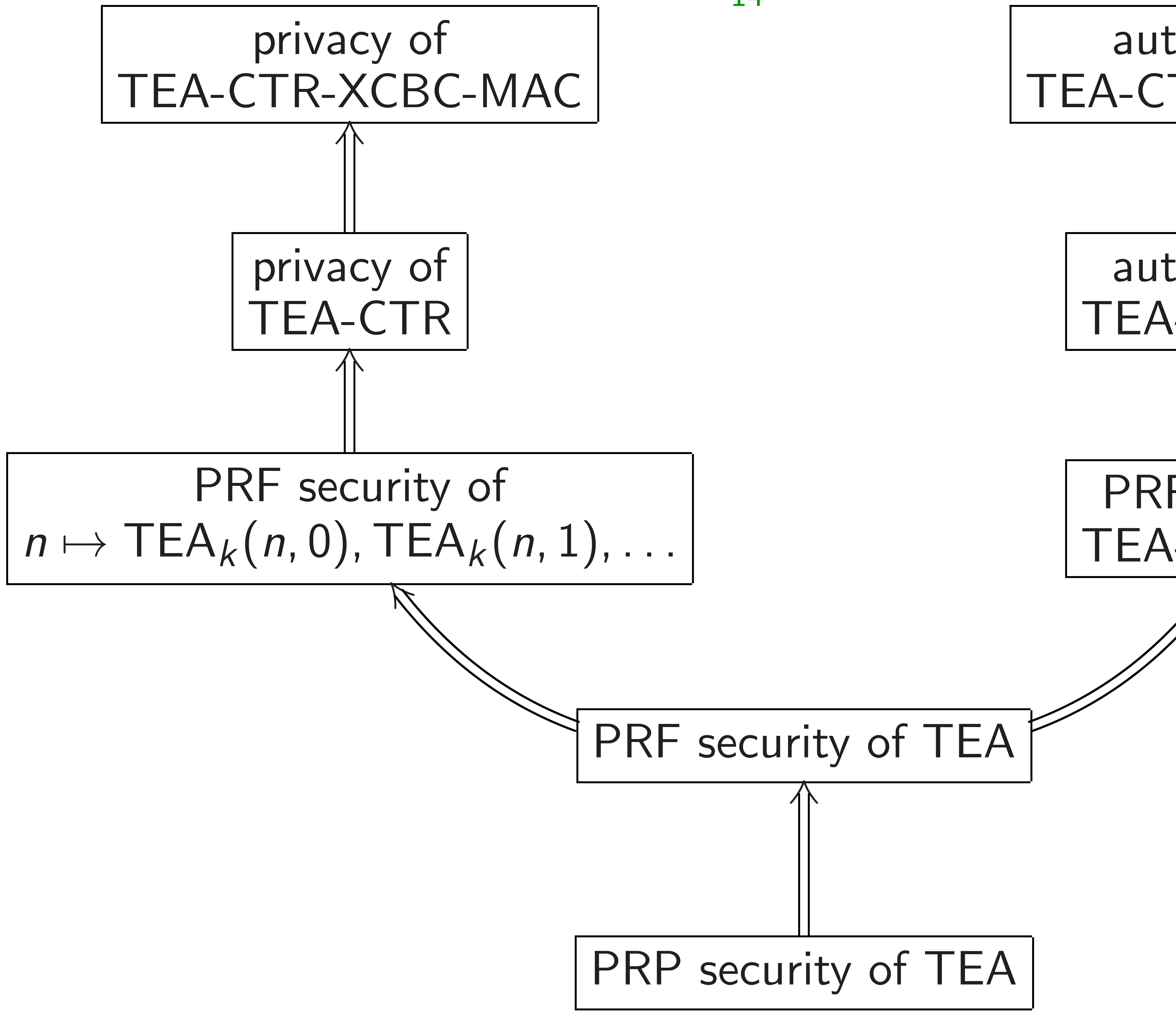
ve that
F attack against
$\text{A}_k(n, 0), \text{TEA}_k(n, 1), \ldots$
PRF attack against
$\text{A}_k(b)$.



ling on
inition,
y follows
erties.

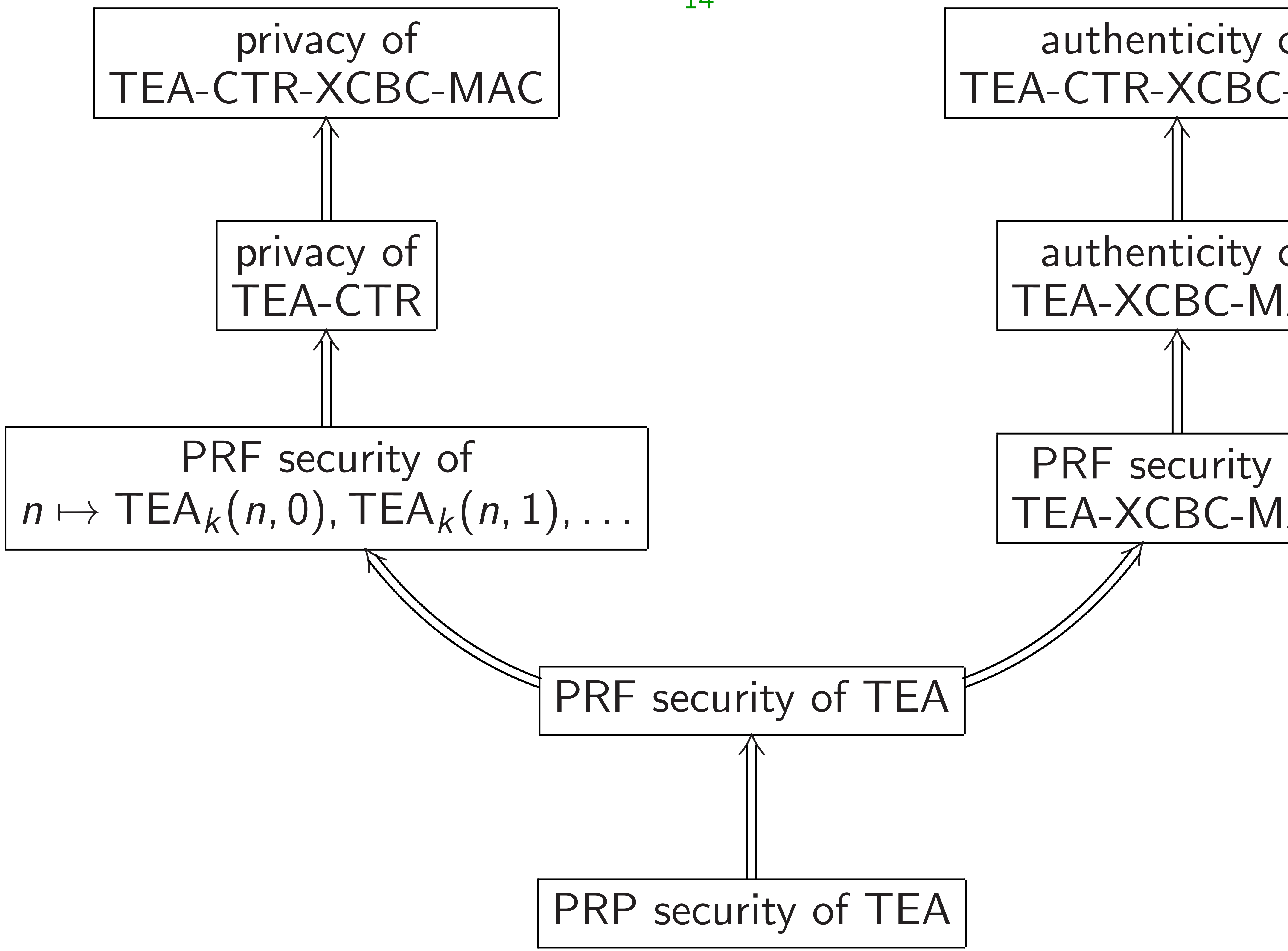
curity of
TEA$_k(n, 1), \ldots$
urity of

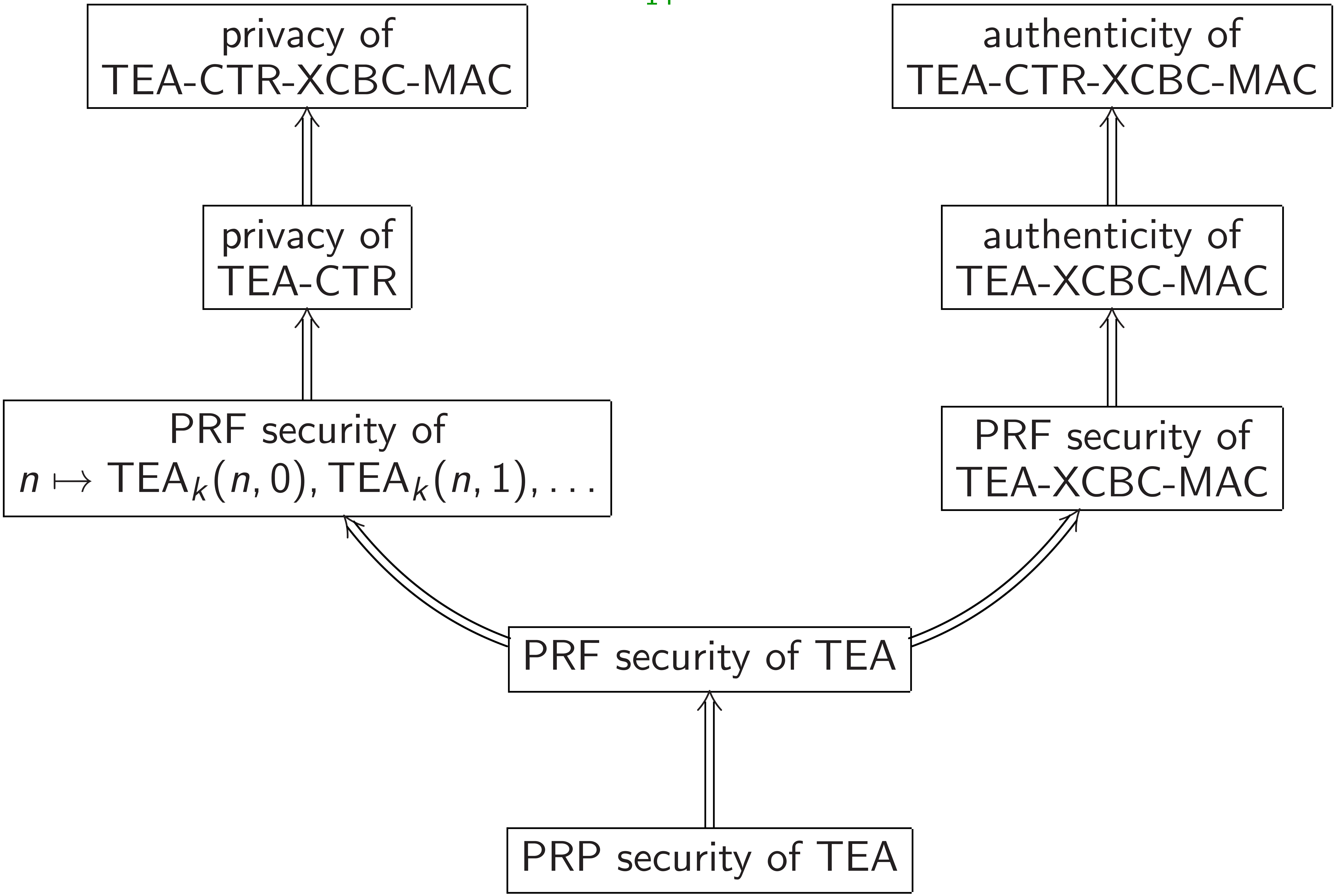
gainst
TEA$_k(n, 1), \ldots$
k against

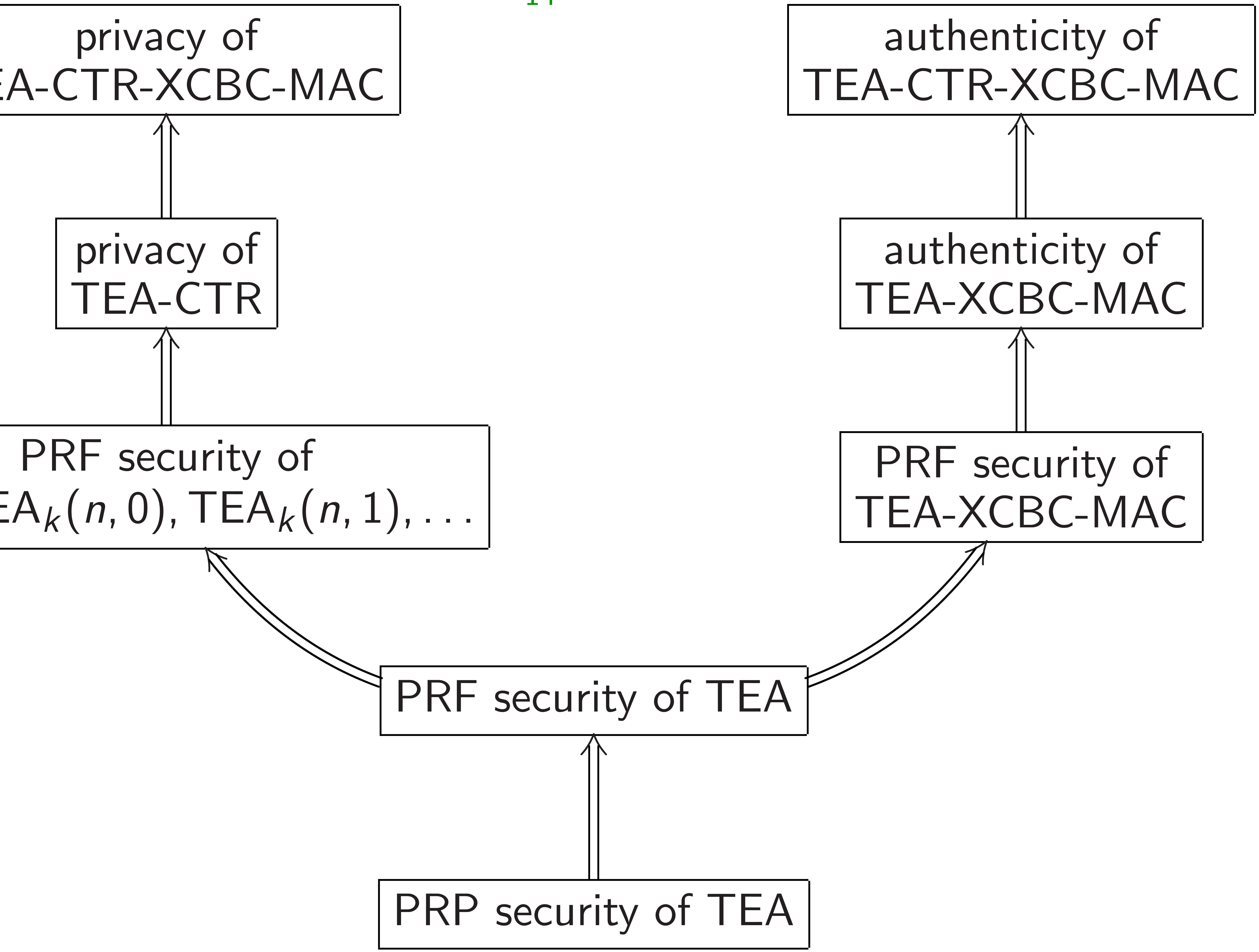


), ...

), ...

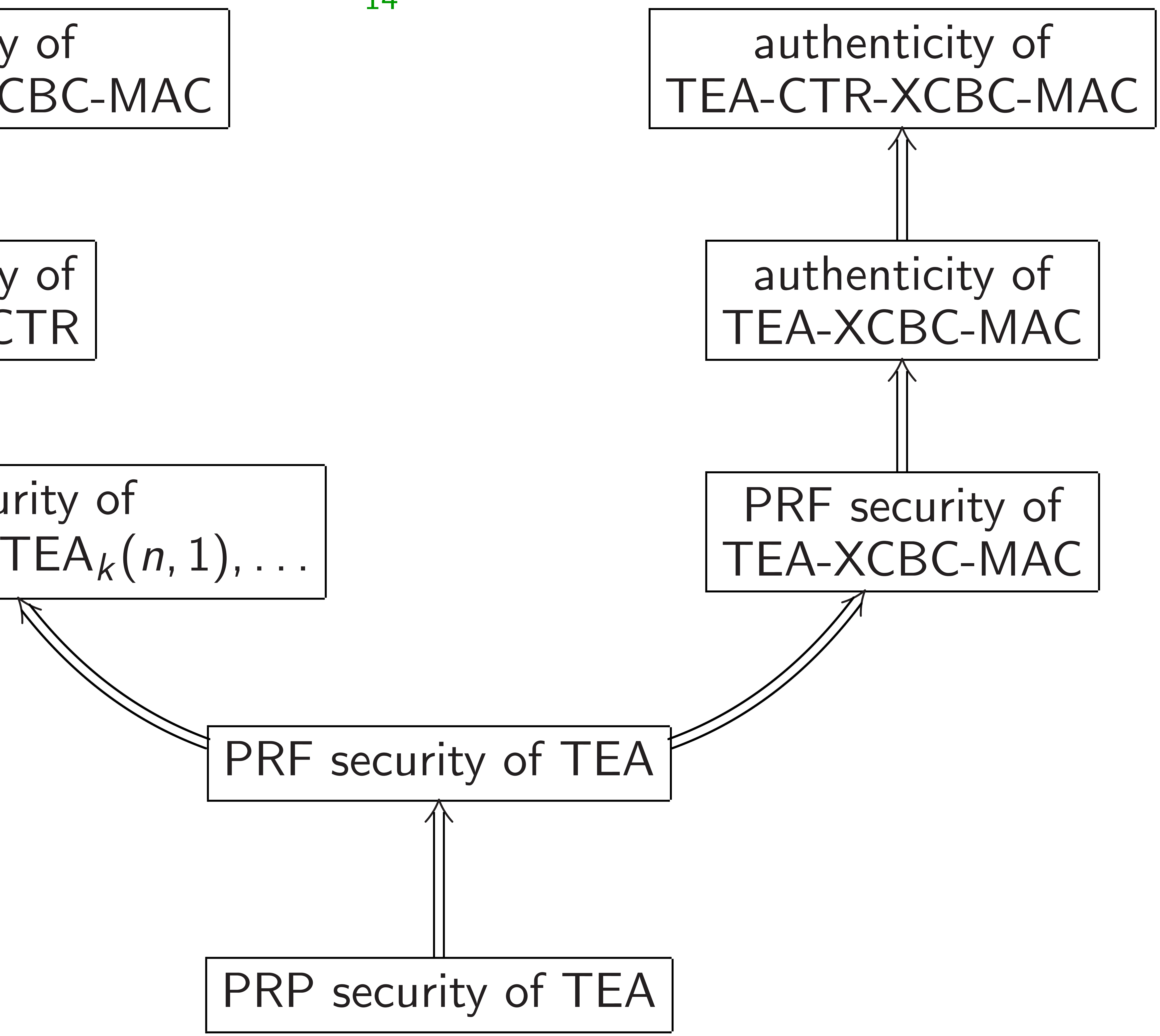

privacy of
TEA-CTR-XCBC-MAC
privacy of
TEA-CTR
PRF security of
$n \mapsto \mathrm{TEA}_k(n, 0), \mathrm{TEA}_k(n, 1), \ldots$
authenticity o
TEA-CTR-XCBC-
authenticity o
TEA-XCBC-MA
PRF security
TEA-XCBC-MA
PRF security of TEA
PRP security of TEA


privacy of TEA-CTR-XCBC-MAC
authenticity of TEA-CTR-XCBC-MAC
privacy of TEA-CTR
authenticity of TEA-XCBC-MAC
PRF security of $n \mapsto \text{TEA}_k(n,0), \text{TEA}_k(n,1), \ldots$
PRF security of TEA-XCBC-MAC
PRF security of TEA
PRP security of TEA



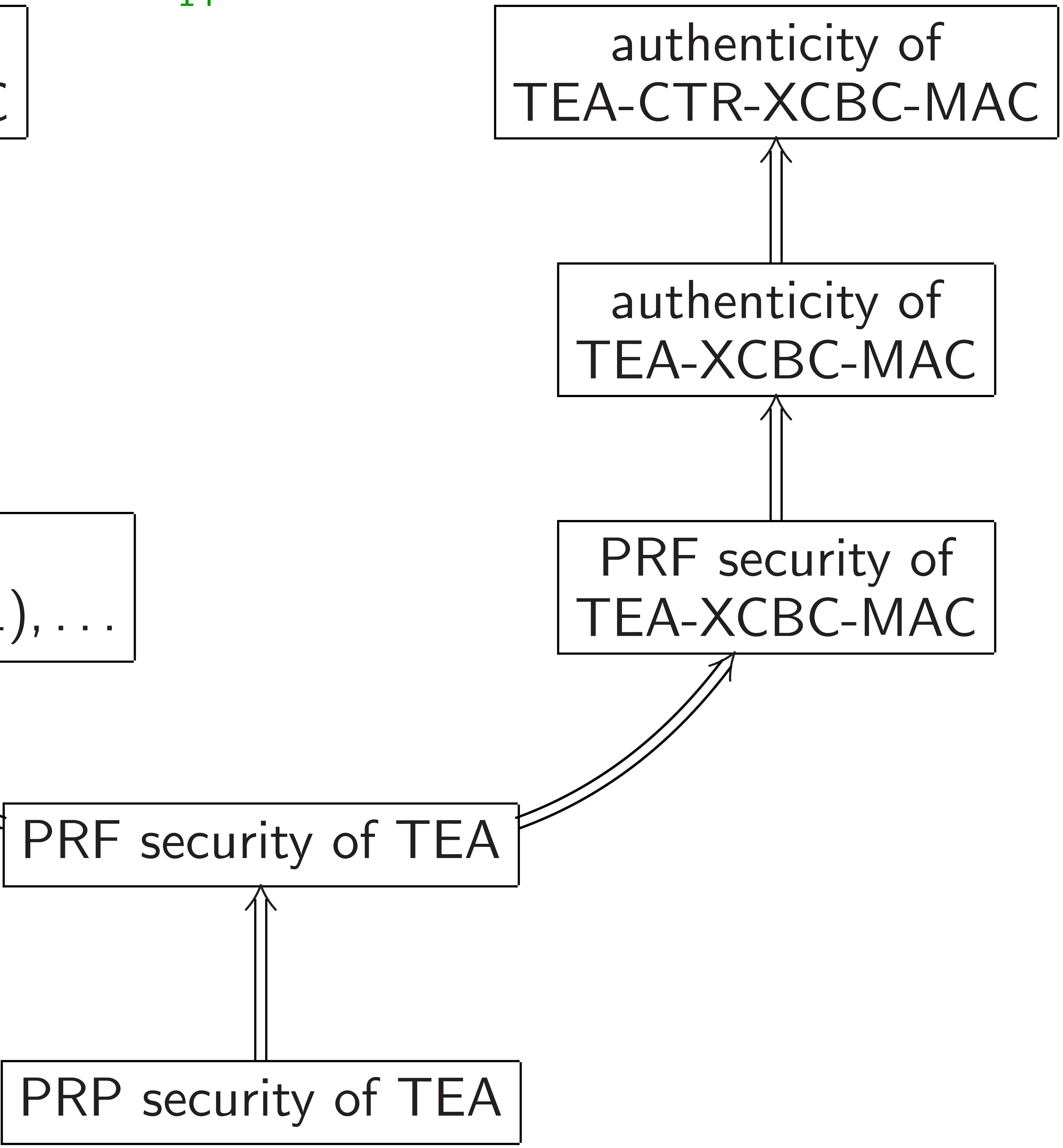
Many th

1. Secur



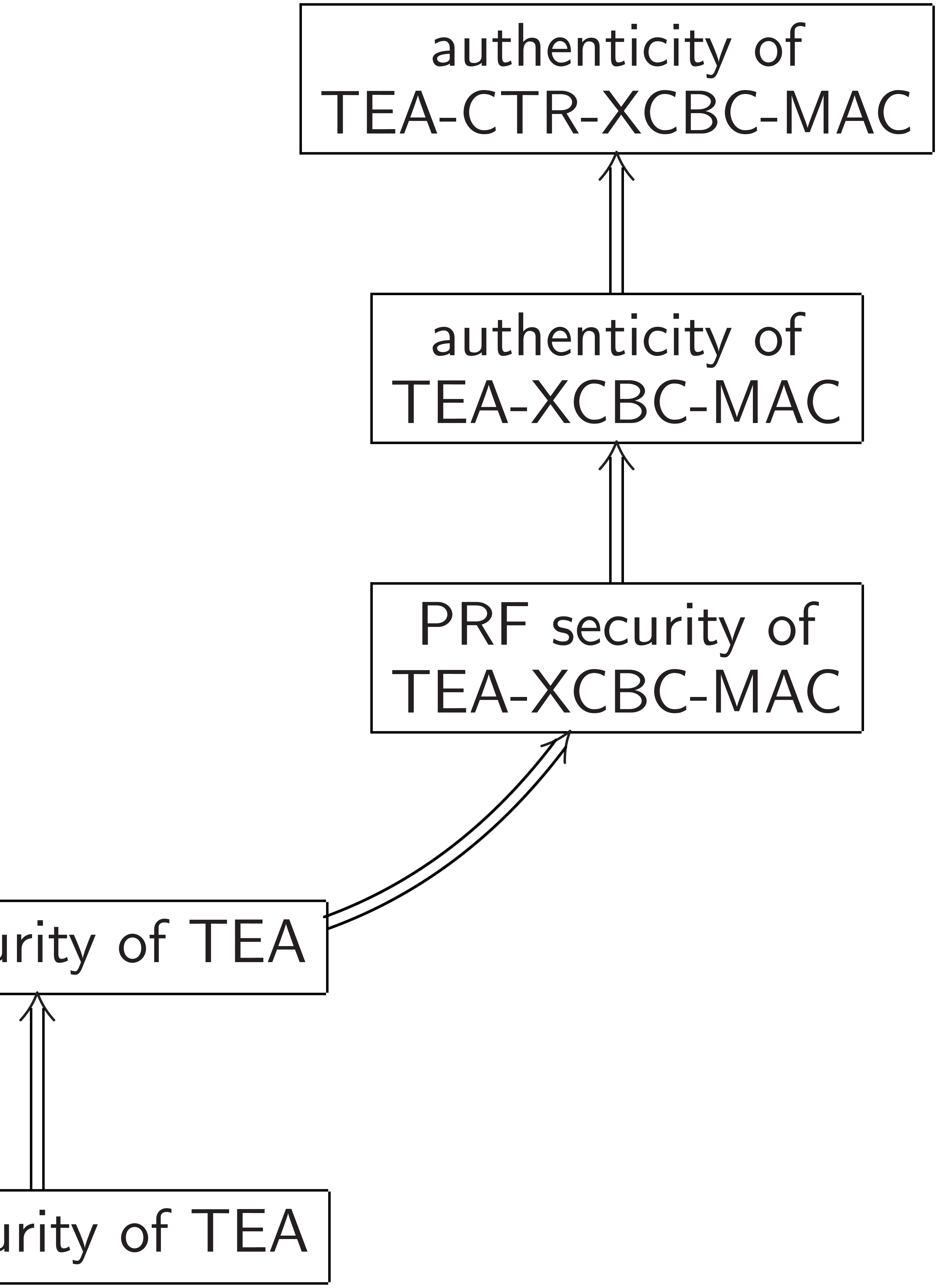
Many things can g

1. Security definit



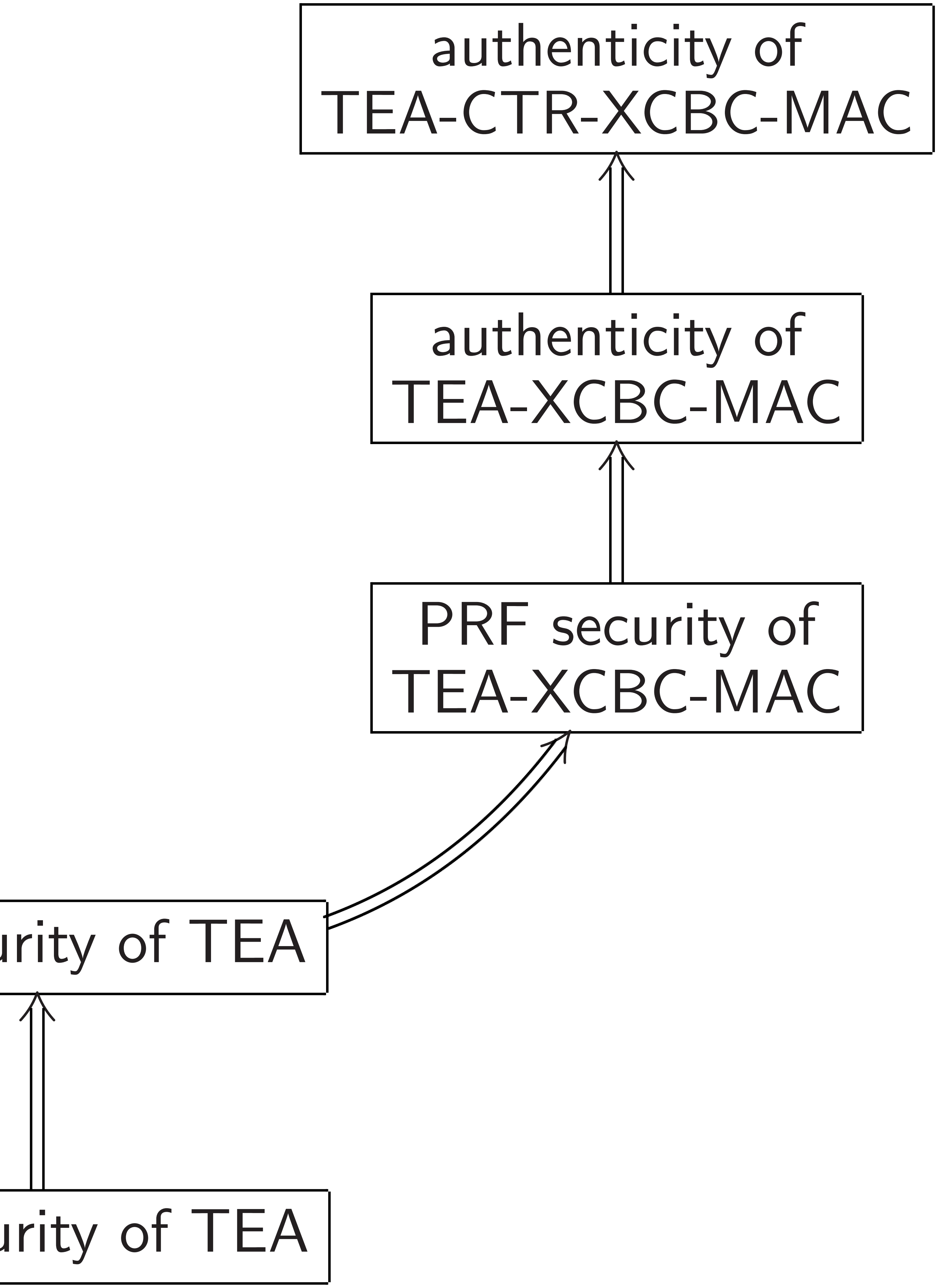
Many things can go wrong h

1. Security definition too we



Many things can go wrong here:

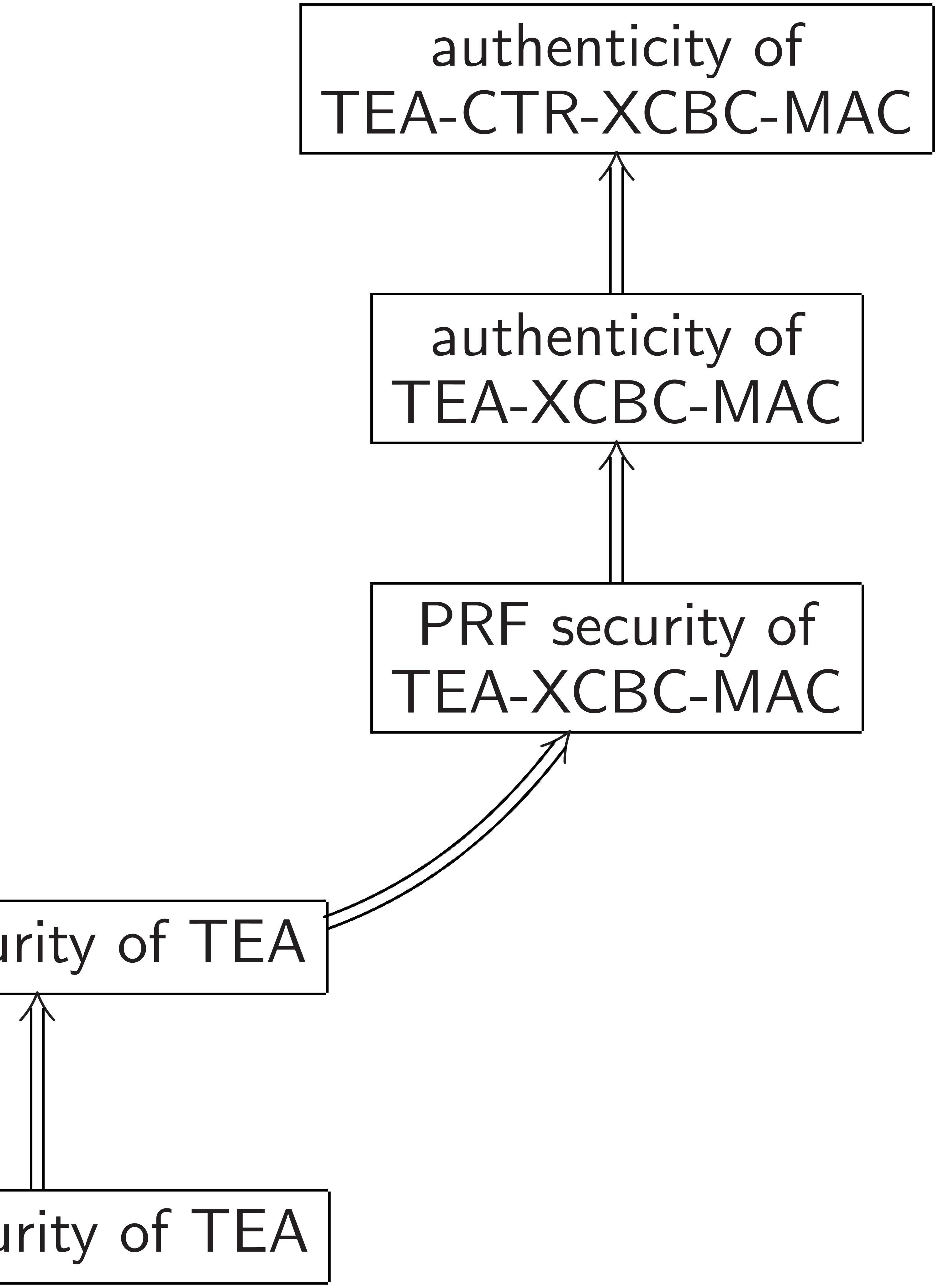
1. Security definition too weak.



Many things can go wrong here:

1. Security definition too weak.

2. Internal mismatch between hypotheses and conclusions.

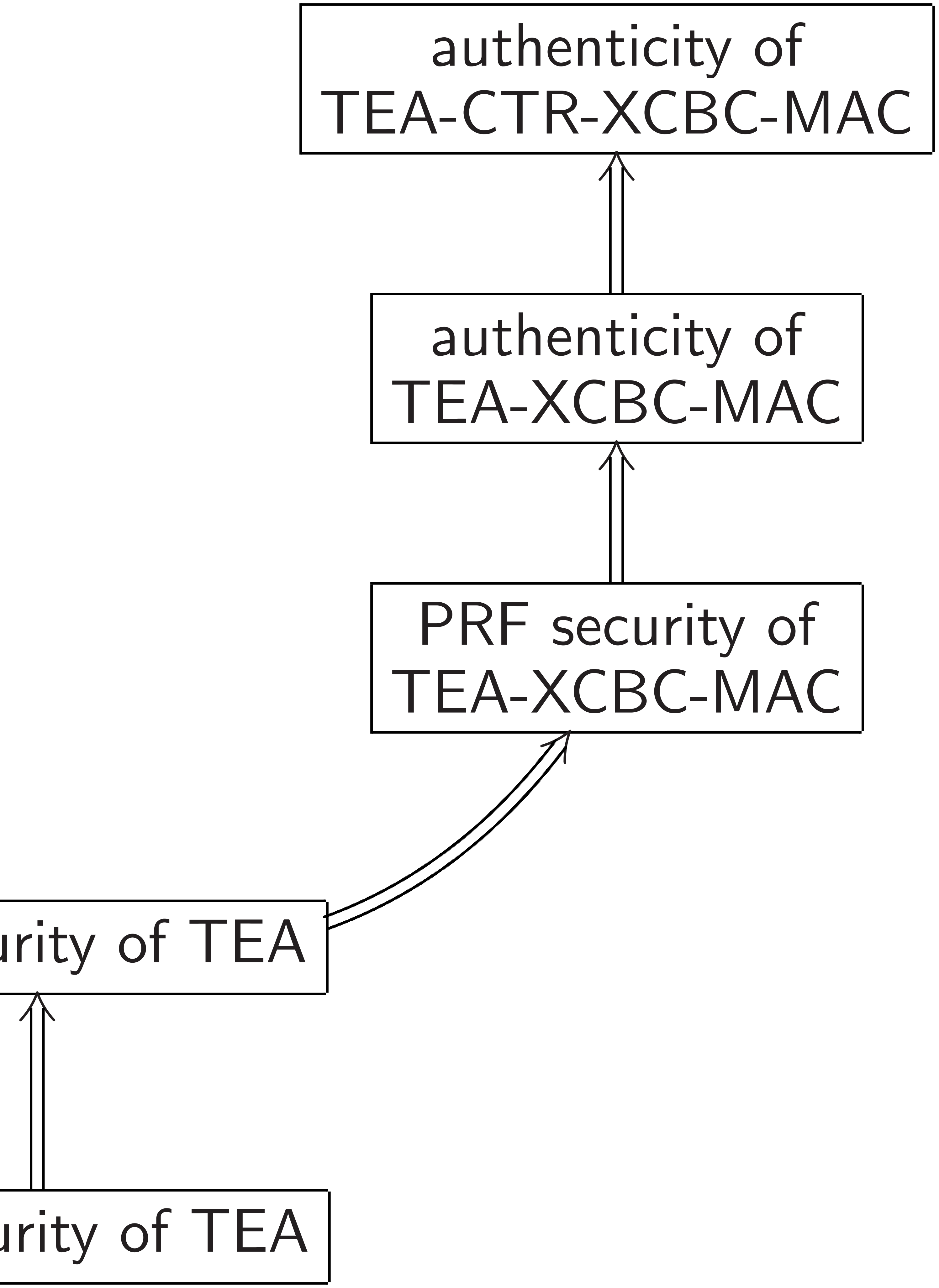


Many things can go wrong here:

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?



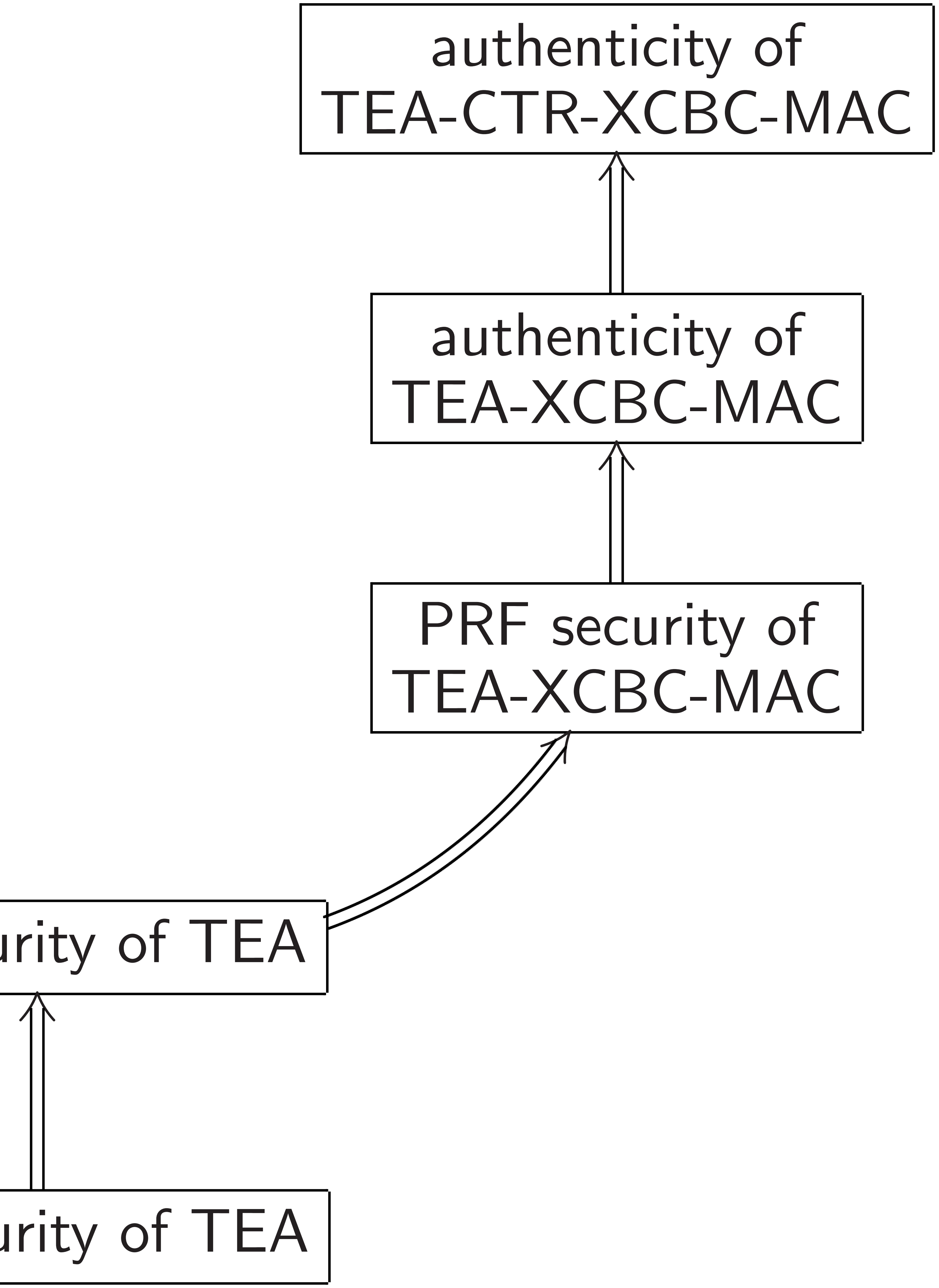
Many things can go wrong here:

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:

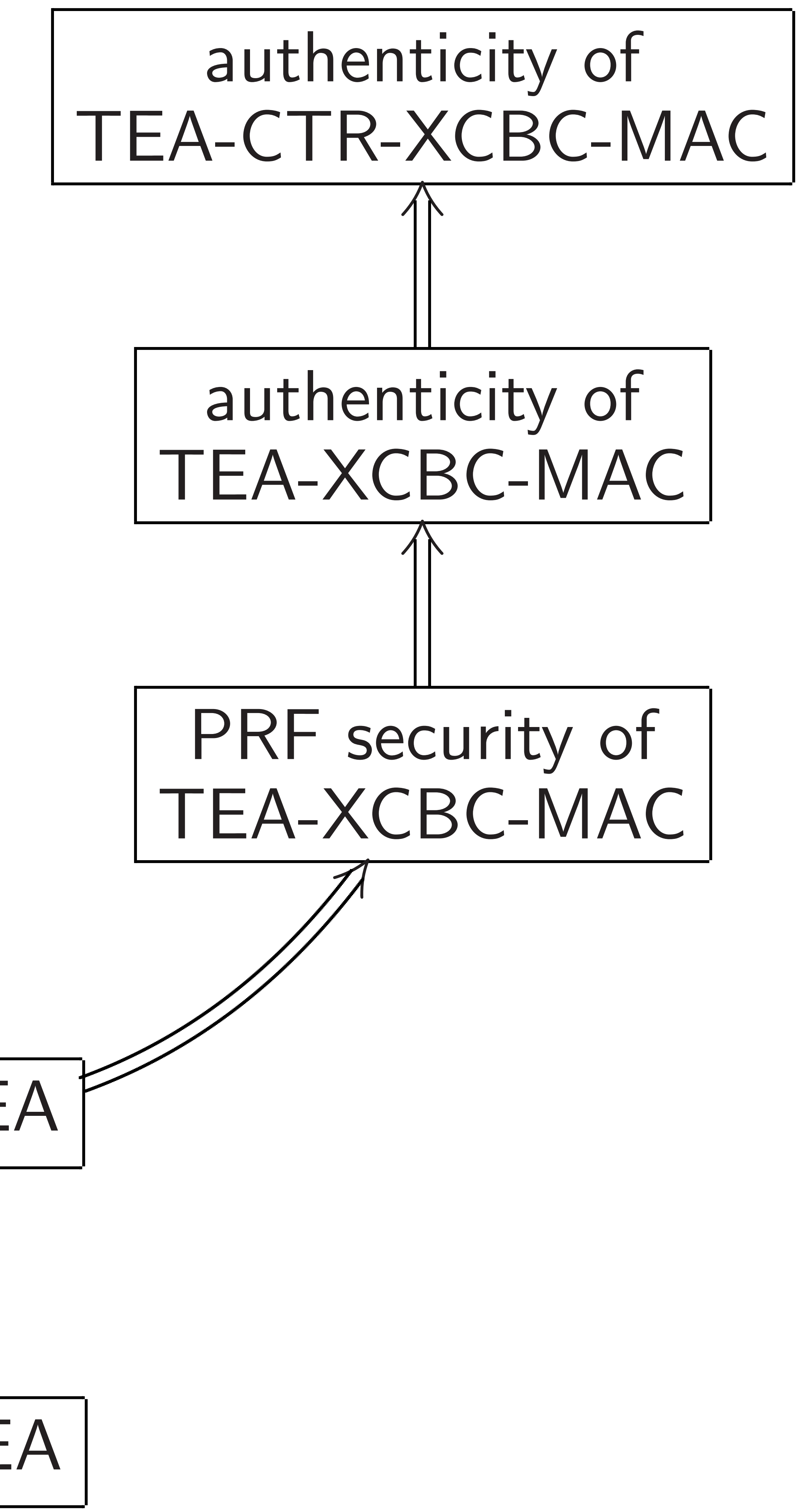
1. Security definition too weak.

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.

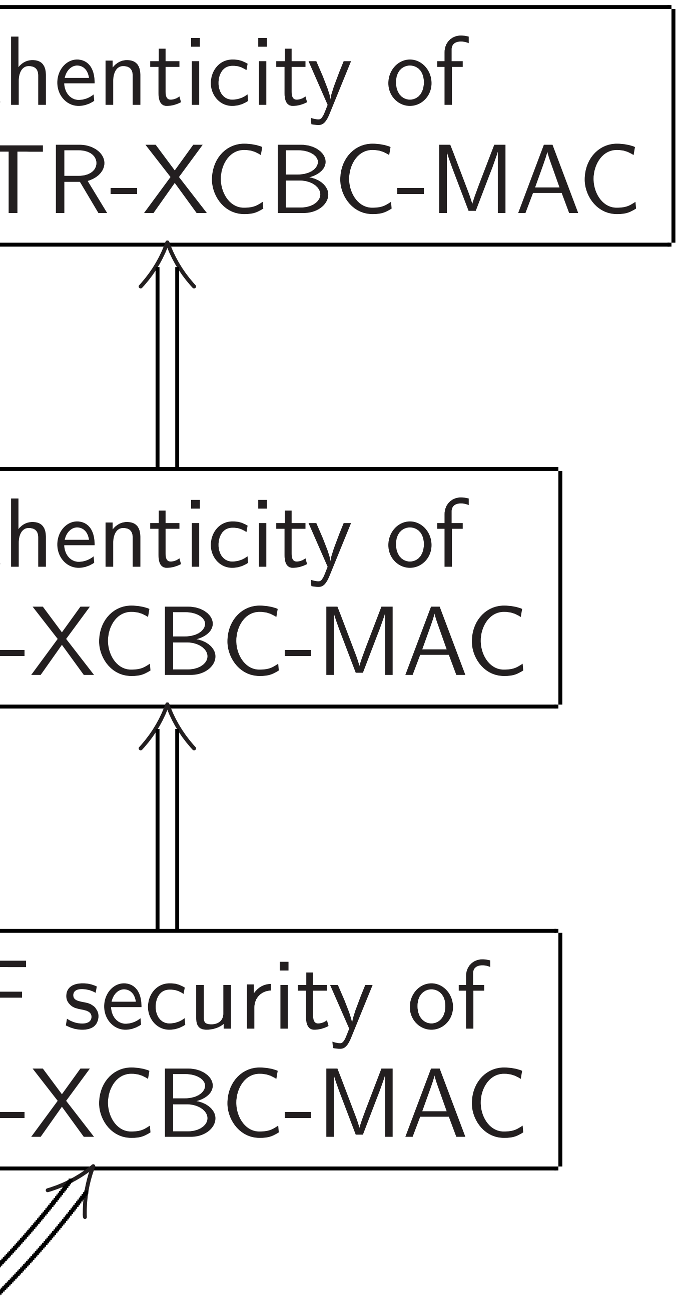
2. Internal mismatch between hypotheses and conclusions.

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-tim
proof of
as long a



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
proof of privacy, b
as long as total of

of
-MAC

of
AC

of
AC

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 mu as long as total of all messa

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."

ings can go wrong here:

rity definition too weak.

nal mismatch between
ses and conclusions.

s in proofs.
one write full proofs?
one check all details?

titative problems.
6 Bhargavan–Leurent
2.info: Triple-DES
n TLS; PRP-PRF switch
k for 64-bit block ciphers.

EA 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

```
void enc
{
  uint32
  uint32
  for (i
    c +=
    x ^=


    y ^=


  }
  b[0] =
}
```

go wrong here:

ion too weak.

tch between
nclusions.

s.
full proofs?
all details?

oblems.
an–Leurent
Triple-DES
RP-PRF switch
t block ciphers.

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 c

```
void encrypt(uin
{
  uint32 x = b[0
  uint32 r, c =
  for (r = 0;r <
    c += 0x9e377
    x ^= y^c ^ (
             ^ (
    y ^= x^c ^ (
             ^ (
  }
  b[0] = x; b[1]
}
```

here:

eak.

en

?

?

t

witch

phers.

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

```
void encrypt(uint32 *b,ui
{
  uint32 x = b[0], y = b[
  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;
}
```

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

```
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;
}
```

e pad has complete
privacy, but key must be
as total of all messages.

n–Carter authenticator has
e proof of authenticity,
length is proportional to
of messages.

y cipher handling many
s: **no complete proofs**.

fecture security
ough failed attack efforts.
hese attacks fail and we
ve better attack ideas."

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;
}
```

"Hardwa
xor circu

complete
ut key must be
all messages.

uthenticator has
authenticity,
proportional to
es.

andling many
nplete proofs.

urity
attack efforts.
ks fail and we
attack ideas."

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-friendli
xor circuit is cheap

st be
ges.

or has
ity,
al to

any
oofs.

fforts.
d we
as."

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

xor circuit is cheaper than a

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.

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

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

A: a bad cipher

```
crypt(uint32 *b,uint32 *k)

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

= 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
with coe
such tha
XORTE

cipher

```
t32 *b,uint32 *k)

], y = b[1];
0;
 32;r += 1) {
9b9;
y<<4)^k[0]
y>>5)^k[1];
x<<4)^k[2]
x>>5)^k[3];

 = 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
with coefficients ir
such that, for all (
$\mathrm{XORTEA}_k(b) = ($

```
nt32 *k)
[1];
1) {
]
];
]
];
```


"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 $\mathbf{F}_2$ 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 $\mathbf{F}_2$ 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
$1 \oplus k_0 \oplus k_1 \oplus k_3 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{20} \oplus k_{21} \oplus k_{30} \oplus k_{32} \oplus k_{33} \oplus k_{35} \oplus k_{42} \oplus k_{43} \oplus k_{44} \oplus k_{52} \oplus k_{53} \oplus k_{62} \oplus k_{64} \oplus k_{67} \oplus k_{69} \oplus k_{76} \oplus k_{85} \oplus k_{94} \oplus k_{96} \oplus k_{99} \oplus k_{101} \oplus k_{108} \oplus k_{117} \oplus k_{126} \oplus b_1 \oplus b_3 \oplus b_{10} \oplus b_{12} \oplus b_{21} \oplus b_{30} \oplus b_{32} \oplus b_{33} \oplus b_{35} \oplus b_{37} \oplus b_{39} \oplus b_{42} \oplus b_{43} \oplus b_{44} \oplus b_{47} \oplus b_{52} \oplus b_{53} \oplus b_{57} \oplus b_{62}$.

There is a matrix M with coefficients in $\mathbf{F}_2$ such that, for all (k, b), $\text{XORTEA}_k(b) = (1, k, b)M$.

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

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 $\mathbf{F}_2$ 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 $F(b_1) \oplus F(b_2) = F(b_3) \oplus F(b_4)$.

are-friendlier" cipher, since
uit is cheaper than add.

put bits are linear
s of input bits!

t output bit is
$k_1 \oplus k_3 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus$
$1 \oplus k_{30} \oplus k_{32} \oplus k_{33} \oplus k_{35} \oplus$
$3 \oplus k_{44} \oplus k_{52} \oplus k_{53} \oplus k_{62} \oplus$
$7 \oplus k_{69} \oplus k_{76} \oplus k_{85} \oplus k_{94} \oplus$
$\oplus k_{101} \oplus k_{108} \oplus k_{117} \oplus k_{126} \oplus$
$b_{10} \oplus b_{12} \oplus b_{21} \oplus b_{30} \oplus b_{32} \oplus$
$5 \oplus b_{37} \oplus b_{39} \oplus b_{42} \oplus b_{43} \oplus$
$7 \oplus b_{52} \oplus b_{53} \oplus b_{57} \oplus b_{62}.$

There is a matrix M
with coefficients in $\mathbf{F}_2$
such that, for all (k, b),
$\text{XORTEA}_k(b) = (1, k, b)M$.

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

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

```
void enc
{
  uint32
  uint32
  for (i
    c +=
    x +=

    y +=

  }
  b[0] =
}
```

er" cipher, since
per than add.

e linear
bits!

it is
$k_{10} \oplus k_{11} \oplus k_{12} \oplus$
$k_{32} \oplus k_{33} \oplus k_{35} \oplus$
$k_{52} \oplus k_{53} \oplus k_{62} \oplus$
$k_{76} \oplus k_{85} \oplus k_{94} \oplus$
$_{108} \oplus k_{117} \oplus k_{126} \oplus$
$\oplus b_{21} \oplus b_{30} \oplus b_{32} \oplus$
$b_{39} \oplus b_{42} \oplus b_{43} \oplus$
$b_{53} \oplus b_{57} \oplus b_{62}$.

There is a matrix M with coefficients in $\mathbf{F}_2$ such that, for all (k, b), $\text{XORTEA}_k(b) = (1, k, b)M$.

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

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

```
void encrypt(uin
{
  uint32 x = b[0
  uint32 r, c =
  for (r = 0;r <
    c += 0x9e377
    x += y+c ^ (
             ^ (
    y += x+c ^ (
             ^ (
  }
  b[0] = x; b[1]
}
```

, since
dd.

$\oplus k_{12} \oplus$
$\oplus k_{35} \oplus$
$\oplus k_{62} \oplus$
$\oplus k_{94} \oplus$
$\oplus k_{126} \oplus$
$\oplus b_{32} \oplus$
$\oplus b_{43} \oplus$
$\oplus b_{62}$.

There is a matrix M with coefficients in $\mathbf{F}_2$ such that, for all (k, b), $\text{XORTEA}_k(b) = (1, k, b)M$.

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

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 ciphe

```
void encrypt(uint32 *b,ui
{
  uint32 x = b[0], y = b[
  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;
}
```

There is a matrix M with coefficients in $\mathbf{F}_2$ such that, for all (k, b), $\text{XORTEA}_k(b) = (1, k, b)M$.

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

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

```
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 matrix M
efficients in $\mathbf{F}_2$
nt, for all (k, b),
$\text{A}_k(b) = (1, k, b)M$.

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

t attack:
$b_1 \oplus b_2 \oplus b_3$ then
$\text{A}_k(b_1) \oplus \text{XORTEA}_k(b_2) =$
$\text{A}_k(b_3) \oplus \text{XORTEA}_k(b_4)$.

aks PRP (and PRF):
random permutation
tion) F almost never has
$F(b_2) = F(b_3) \oplus F(b_4)$.

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
but addi

First out
$1 \oplus k_0 \oplus$

M

n $\mathbf{F}_2$

(k, b),

$1, k, b)M$.

$\text{XORTEA}_k(b_2)$

M.

b_3 then

$\text{XORTEA}_k(b_2) =$

$\text{XORTEA}_k(b_4)$.

(and PRF):

ermutation

most never has

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

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 $\mathbf{F}_2$

but addition mod

First output bit is

$1 \oplus k_0 \oplus k_{32} \oplus k_{64}$

$_k(b_2)$

$(b_2) =$
$_k(b_4)$.

):
n
r has
(b_4).

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 $\mathbf{F}_2$-linear,
but addition mod 2 is $\mathbf{F}_2$-lin

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

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 $\mathbf{F}_2$-linear,
but addition mod 2 is $\mathbf{F}_2$-linear.

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

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 $\mathbf{F}_2$-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.

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 $\mathbf{F}_2$-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: >>5 **diffuses** nonlinear changes from high bits to low bits.

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 $\mathbf{F}_2$-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: >>5 **diffuses** nonlinear changes from high bits to low bits.

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

A: another bad cipher

```
crypt(uint32 *b,uint32 *k)

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

= x; b[1] = y;
```


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: >>5 **diffuses** nonlinear changes from high bits to low bits.

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

TEA4: a

```
void enc
{
  uint32
  uint32
  for (r
    c +=
    x +=

    y +=

  }
  b[0] =
}
```

bad cipher

```
t32 *b,uint32 *k)

], y = b[1];
0;
 32;r += 1) {
9b9;
y<<4)+k[0]
y<<5)+k[1];
x<<4)+k[2]
x<<5)+k[3];

 = y;
```


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: >>5 **diffuses** nonlinear changes from high bits to low bits.

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

TEA4: another ba

```
void encrypt(uin
{
  uint32 x = b[0
  uint32 r, c =
  for (r = 0;r <
    c += 0x9e377
    x += y+c ^ (
             ^ (
    y += x+c ^ (
             ^ (
  }
  b[0] = x; b[1]
}
```

r

```
nt32 *k)

1];

 1) {

]

];

]

];
```


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:
>>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

```
void encrypt(uint32 *b,ui
{
  uint32 x = b[0], y = b[
  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 $\mathbf{F}_2$-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:
>>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

```
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;
}
```

n is not $\mathbf{F}_2$-linear,
tion mod 2 is $\mathbf{F}_2$-linear.

tput bit is
$\oplus k_{32} \oplus k_{64} \oplus k_{96} \oplus b_{32}$.

output bits
easingly nonlinear
never affect first bit.

A avoids this problem:
uses nonlinear changes
h bits to low bits.

on from low bits to high
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<<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 atta
$\text{TEA4}_k($
$\text{TEA4}_k($

-linear,
2 is $\mathbf{F}_2$-linear.

$\oplus k_{96} \oplus b_{32}$.

s
onlinear
ect first bit.

this problem:
near changes
ow bits.

w bits to high
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<<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:

$\mathrm{TEA4}_k(x + 2^{31}, y$

$\mathrm{TEA4}_k(x, y)$ have

ear.

b_{32}.

t.

m:

ges

nigh

n.)

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:

$\text{TEA4}_k(x + 2^{31}, y)$ and

$\text{TEA4}_k(x, y)$ have same first

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:

$\mathrm{TEA4}_k(x + 2^{31}, y)$ and $\mathrm{TEA4}_k(x, y)$ 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:
$\text{TEA4}_k(x + 2^{31}, y)$ 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 $2^{1}, 2^{0}$.

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:
$\text{TEA4}_k(x+2^{31}, y)$ 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 $1/2$.

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:

$\text{TEA4}_k(x + 2^{31}, y)$ 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 $3/4$.

another bad cipher

```
crypt(uint32 *b,uint32 *k)

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

= x; b[1] = y;
```


Fast attack:

$\mathrm{TEA4}_k(x + 2^{31}, y)$ and
$\mathrm{TEA4}_k(x, y)$ 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 so
trace *pro*
probabili
probabili
differenc
$C(x + \delta$
Use alge
non-ranc

nd cipher

```
t32 *b,uint32 *k)

], y = b[1];
0;
 4;r += 1) {
9b9;
y<<4)+k[0]
y>>5)+k[1];
x<<4)+k[2]
x>>5)+k[3];


 = y;
```


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 sophisticatec
trace *probabilities*
probabilities of line
probabilities of hig
differences $C(x +$
$C(x + \delta) - C(x +$
Use algebra+statis
non-randomness ir

```
nt32 *k)
[1];
) {
]
];
]
];
```

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 differen
probabilities of linear equati
probabilities of higher-order
differences $C(x + \delta + \epsilon) -$
$C(x + \delta) - C(x + \epsilon) + C(x)$
Use algebra+statistics to ex
non-randomness in probabili

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 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 probabilities.

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

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 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?

ack:
$x + 2^{31}, y)$ and
$x, y)$ have same first bit.

y differences
steps in computation.
multiples of $2^{31}, 2^{26}$.
multiples of $2^{21}, 2^{16}$.
multiples of $2^{11}, 2^{6}$.
multiples of $2^{1}, 2^{0}$.

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

vantage $1/2$.
rs (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

```
void enc
{
  uint32
  uint32
  for (i
    x +=

    y +=

  }
  b[0] =
}
```

) and
same first bit.

ces
omputation.
f $2^{31}, 2^{26}$.
f $2^{21}, 2^{16}$.
f $2^{11}, 2^{6}$.
f $2^{1}, 2^{0}$.

unction F:
$F(x, y)$ have
probability $1/2$.

/2.
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

```
void encrypt(uin
{
  uint32 x = b[0
  uint32 r, c =
  for (r = 0;r <
    x += y+c ^ (
             ^ (
    y += x+c ^ (
             ^ (
  }
  b[0] = x; b[1]
}
```

t bit.

on.

t

ave

ty $1/2$.

e $3/4$.

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 ciphe

```
void encrypt(uint32 *b,ui
{
  uint32 x = b[0], y = b[
  uint32 r, c = 0x9e3779b
  for (r = 0;r < 1000;r +
    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

```
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;
}
```

phisticated attacks:
obabilities of differences;
ities of linear equations;
ities of higher-order
ces $C(x + \delta + \epsilon) -$
$) - C(x + \epsilon) + C(x)$; etc.
ebra+statistics to exploit
domness in probabilities.

get beyond $r = 4$
dly lose effectiveness.
from full TEA.

estion in cipher design:
ny "rounds" are
eeded for security?

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

where $I_{\!}$

d attacks:
of differences;
ear equations;
gher-order
$\delta + \epsilon) -$
$\epsilon) + C(x)$; etc.
stics to exploit
n probabilities.

d $r = 4$
fectiveness.
TEA.

cipher design:
ls" are
ecurity?

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;
}
```

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

where I_k does x+=

nces;

ons;

); etc.

ploit

ties.

s.

gn:

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;
}
```

$\mathrm{REPTEA}_k(b) = I_k^{1000}(b)$

where I_k does `x+=...;y+=`

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;
}
```


$\text{REPTEA}_k(b) = I_k^{1000}(b)$

where I_k does `x+=...;y+=....`

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;
}
```


$\text{REPTEA}_k(b) = I_k^{1000}(b)$
where I_k does `x+=...;y+=...`.

Try list of 2^{32} inputs b.
Collect outputs $\text{REPTEA}_k(b)$.

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;
}
```


$\text{REPTEA}_k(b) = I_k^{1000}(b)$
where I_k does `x+=...;y+=...`.

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

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;
}
```


$\text{REPTEA}_k(b) = I_k^{1000}(b)$
where I_k does `x+=...;y+=...`.

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.)

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;
}
```


$\text{REPTEA}_k(b) = I_k^{1000}(b)$
where I_k does `x+=...;y+=...`.

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`.

A: another bad cipher

```
crypt(uint32 *b,uint32 *k)

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

= x; b[1] = y;
```


$\text{REPTEA}_k(b) = I_k^{1000}(b)$
where I_k does `x+=...;y+=...`.

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`.

What ab

```
void enc
{
  uint32
  uint32
  for (
    c +=
    x +=

    y +=

  }
  b[0] =
}
```

bad cipher

```
t32 *b,uint32 *k)

], y = b[1];
0x9e3779b9;
 1000;r += 1) {
y<<4)+k[0]
y>>5)+k[1];
x<<4)+k[2]
x>>5)+k[3];

 = y;
```


$\text{REPTEA}_k(b) = I_k^{1000}(b)$
where I_k does `x+=...;y+=...`.

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`.

What about origin

```
void encrypt(uin
{
  uint32 x = b[0
  uint32 r, c =
  for (r = 0;r <
    c += 0x9e377
    x += y+c ^ (
             ^ (
    y += x+c ^ (
             ^ (
  }
  b[0] = x; b[1]
}
```

er

```
nt32 *k)

[1];
9;
= 1) {
]
];
]
];
```

$\mathrm{REPTEA}_k(b) = I_k^{1000}(b)$
where I_k does `x+=...;y+=....`

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

For each (b, a) from list:
Try solving equations $a = I_k(b)$, $\mathrm{REPTEA}_k(a) = I_k(\mathrm{REPTEA}_k(b))$ 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?

```
void encrypt(uint32 *b,ui
{
  uint32 x = b[0], y = b[
  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;
}
```

$\text{REPTEA}_k(b) = I_k^{1000}(b)$
where I_k does x+=...;y+=....

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.

What about original TEA?

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

hance that some b in list

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

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

(b, a) from list:

ing equations $a = I_k(b)$,

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

out k. (More equations:

ncrypting these outputs.)

slide attack.

oids this by varying c.

What about original TEA?

```
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

$\text{TEA}_{k'}(b$

where $(k$

$(k[0] + 2$

$_k^{1000}(b)$

=...;y+=....

uts b.

EPTEA$_k(b)$.

some b in list

) in list. Then

(REPTEA$_k(b)$).

om list:

ons $a = I_k(b)$,

(REPTEA$_k(b)$)

More equations:

these outputs.)

ack.

y varying c.

What about original TEA?

```
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$_k$

where $(k'[0], k'[1],$

$(k[0] + 2^{31}, k[1] +$

. . . .

b).

n list

Then

$_k(b)$).

$_k(b)$,

$_k(b)$)

tions:

uts.)

c.

What about original TEA?

```
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.,

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

where $(k'[0], k'[1], k'[2], k'[3$

$(k[0] + 2^{31}, k[1] + 2^{31}, k[2],$

What about original TEA?

```
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?

```
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?

```
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 TEA_k, for one secret key k, from uniform random permutation.

What about original TEA?

```
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 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.

What about original TEA?

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

bout original TEA?

```
crypt(uint32 *b,uint32 *k)

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

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

1997 Ke
Fancier
has char
a particu

al TEA?

```
t32 *b,uint32 *k)

], y = b[1];
0;
 32;r += 1) {
9b9;
y<<4)+k[0]
y>>5)+k[1];
x<<4)+k[2]
x>>5)+k[3];

 = 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 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–Schn
Fancier relationshi
has chance 2^{-11} c
a particular outpu

```
nt32 *k)
[1];
1) {
)]
];
2]
3];
```

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 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–Wagn
Fancier relationship between
has chance 2^{-11} of producin
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 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 $\Rightarrow$ 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 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 $\Rightarrow$ 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.

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 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 $\Rightarrow$ 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.

keys: e.g.,
$b) = \text{TEA}_k(b)$
$k'[0], k'[1], k'[2], k'[3]) =$
$2^{31}, k[1] + 2^{31}, k[2], k[3])$.

n attack?

ack goal: distinguish
or one secret key k, from
random permutation.

rce attack:
ey g, see if TEA_g
TEA_k on some outputs.

keys $\Rightarrow$ g succeeds with
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.

Some wa
about ci
hash-fun

Take up
"Selecte
Includes

Read att
especiall
Try to b
e.g., find
Reasona
2000 Sc
in block-

,

$_k(b)$

$, k'[2], k'[3]) =$

$2^{31}, k[2], k[3])$.

distinguish

ret key k, from

ermutation.

:

f TEA_g

some outputs.

succeeds with

ll 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.

Some ways to lear

about cipher attac

hash-function atta

Take upcoming co

"Selected areas in

Includes symmetri

Read attack paper

especially from FS

Try to break ciphe

e.g., find attacks o

Reasonable startin

2000 Schneier "Se

in block-cipher cry

3]) =
$k[3]$).
h
from
n.
puts.
with
all.

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 cryptolog
Includes symmetric attacks.

Read attack papers, especially from FSE conferen
Try to break ciphers yoursel
e.g., find attacks on FEAL.
Reasonable starting point:
2000 Schneier "Self-study co
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".

lsey–Schneier–Wagner:
relationship between k, k'
nce 2^{-11} of producing
ular output equation.

ence in literature that
s brute-force attack,
wise affects PRP security.
enge to security analysis
CTR-XCBC-MAC.

ertised as
-key cryptanalysis"
med to justify
endations for designers
g 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 ci

1973, a
U.S. Na
Standar
for a Da

eier–Wagner:
p between k, k'
of producing
t equation.

erature that
rce attack,
ts PRP security.
ecurity analysis
C-MAC.

analysis"
stify
for designers
eduling.

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 histor

1973, and again in
U.S. National Bur
Standards solicits
for a Data Encryp

her:
n k, k'
ng
.
at
,
curity.
lysis
ers

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

1973, and again in 1974: U.S. National Bureau of Standards solicits proposals for a Data Encryption Stanc

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

1973, and again in 1974:
U.S. National Bureau of
Standards solicits proposals
for a Data Encryption Standard.

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

1973, and again in 1974:
U.S. National Bureau of
Standards solicits proposals
for a Data Encryption Standard.

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

1973, and again in 1974: U.S. National Bureau of Standards solicits proposals for a Data Encryption Standard.

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."

ays to learn more
pher attacks,
nction attacks, etc.:

coming course
d areas in cryptology".
symmetric attacks.

tack papers,
y from FSE conference.
reak ciphers yourself:
l attacks on FEAL.
ble starting point:
hneier "Self-study course
-cipher cryptanalysis".

Some cipher history

1973, and again in 1974:
U.S. National Bureau of
Standards solicits proposals
for a Data Encryption Standard.

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: D

1977: D
publish c
$200000
hundreds

n more
cks,
cks, etc.:

urse
cryptology".
c attacks.

rs,
E conference.
ers yourself:
on FEAL.
g point:
elf-study course
ptanalysis".

Some cipher history

1973, and again in 1974:
U.S. National Bureau of
Standards solicits proposals
for a Data Encryption Standard.

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 stan

1977: Diffie and H
publish detailed de
$20000000 machin
hundreds of DES k

y”.

nce.

f:

ourse

”.

Some cipher history

1973, and again in 1974: U.S. National Bureau of Standards solicits proposals for a Data Encryption Standard.

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 brea
hundreds of DES keys per ye

Some cipher history

1973, and again in 1974:
U.S. National Bureau of Standards solicits proposals for a Data Encryption Standard.

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

1973, and again in 1974: U.S. National Bureau of Standards solicits proposals for a Data Encryption Standard.

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

1973, and again in 1974: U.S. National Bureau of Standards solicits proposals for a Data Encryption Standard.

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".

1983, 1988, 1993: Government reaffirms DES standard.

Some cipher history

1973, and again in 1974:
U.S. National Bureau of Standards solicits proposals for a Data Encryption Standard.

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".

1983, 1988, 1993: Government reaffirms DES standard.

Researchers publish new cipher proposals and security analysis.

pher history

nd again in 1974:
tional Bureau of
ds solicits proposals
ta Encryption Standard.

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

SA meets Diffie and
to discuss criticism.
'somewhere over
0,000" to break a DES
don't think you can tell
gressman what's going to
e 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".

1983, 1988, 1993: Government reaffirms DES standard.

Researchers publish new cipher proposals and security analysis.

1997: U
of Stand
(NIST, f
for prop
Encrypti
block, 1

ry

n 1974:
eau of
proposals
tion Standard.

hes IBM DES
lock, 56-bit key.

Diffie and
s criticism.
re over
break a DES
k you can tell
what's going to
. 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".

1983, 1988, 1993: Government reaffirms DES standard.

Researchers publish new cipher proposals and security analysis.

1997: U.S. Nation
of Standards and T
(NIST, formerly N
for proposals for A
Encryption Standa
block, 128/192/25

lard.

ES

it key.

ES

tell

ing to

.”

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”.

1983, 1988, 1993: Government reaffirms DES standard.

Researchers publish new cipher proposals and security analysis.

1997: U.S. National Institut
of Standards and Technolog
(NIST, formerly NBS) calls
for proposals for Advanced
Encryption Standard. 128-b
block, 128/192/256-bit key.

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".

1983, 1988, 1993: Government reaffirms DES standard.

Researchers publish new cipher proposals and security analysis.

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

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".

1983, 1988, 1993: Government reaffirms DES standard.

Researchers publish new cipher proposals and security analysis.

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

1998: 15 AES proposals.

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".

1983, 1988, 1993: Government reaffirms DES standard.

Researchers publish new cipher proposals and security analysis.

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

1998: 15 AES proposals.

1998: EFF builds "Deep Crack" for under \$250000 to break hundreds of DES keys per year.

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".

1983, 1988, 1993: Government reaffirms DES standard.

Researchers publish new cipher proposals and security analysis.

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

1998: 15 AES proposals.

1998: EFF builds "Deep Crack" for under $250000 to break hundreds of DES keys per year.

1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

ES is standardized.

iffie and Hellman
detailed design of
00 machine to break
s of DES keys per year.

ongressional investigation
A influence concludes
onvinced IBM that a
key size was sufficient".

988, 1993: Government
s DES standard.

hers publish new cipher
s and security analysis.

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

1998: 15 AES proposals.

1998: EFF builds "Deep Crack" for under $250000 to break hundreds of DES keys per year.

1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: N
selects R

"Securit
factor in

ndardized.

Hellman
esign of
ne to break
keys per year.

nal investigation
e concludes
BM that a
vas sufficient".

Government
ndard.

h new cipher
urity analysis.

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

1998: 15 AES proposals.

1998: EFF builds "Deep Crack" for under $250000 to break hundreds of DES keys per year.

1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advis
selects Rijndael as

"Security was the
factor in the evalu

k

ear.

gation

es

n

ent".

ent

her

sis.

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

1998: 15 AES proposals.

1998: EFF builds "Deep Crack" for under $250000 to break hundreds of DES keys per year.

1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.

2000: NIST, advised by NSA

selects Rijndael as AES.

"Security was the most imp

factor in the evaluation"—R

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

1998: 15 AES proposals.

1998: EFF builds "Deep Crack" for under $250000 to break hundreds of DES keys per year.

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?

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

1998: 15 AES proposals.

1998: EFF builds "Deep Crack" for under $250000 to break hundreds of DES keys per year.

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."

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

1998: 15 AES proposals.

1998: EFF builds "Deep Crack" for under $250000 to break hundreds of DES keys per year.

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."

2004–2008: eSTREAM competition for stream ciphers.

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

1998: 15 AES proposals.

1998: EFF builds "Deep Crack" for under $250000 to break hundreds of DES keys per year.

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."

2004–2008: eSTREAM competition for stream ciphers.

2007–2012: SHA-3 competition.

1997: U.S. National Institute of Standards and Technology (NIST, formerly NBS) calls for proposals for Advanced Encryption Standard. 128-bit block, 128/192/256-bit key.

1998: 15 AES proposals.

1998: EFF builds "Deep Crack" for under $250000 to break hundreds of DES keys per year.

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."

2004–2008: eSTREAM competition for stream ciphers.

2007–2012: SHA-3 competition.

2013–now: CAESAR competition.

.S. National Institute
lards and Technology
formerly NBS) calls
osals for Advanced
on Standard. 128-bit
28/192/256-bit key.

5 AES proposals.

FF builds "Deep Crack"
r $250000 to break
s of DES keys per year.

IST selects five
alists: MARS, RC6,
, 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."

2004–2008: eSTREAM competition for stream ciphers.

2007–2012: SHA-3 competition.

2013–now: CAESAR competition.

Main op
add rour
apply **su**
$x \mapsto x^{25}$
to each
linearly

nal Institute
Technology
BS) calls
Advanced
ard. 128-bit
56-bit key.

posals.

"Deep Crack"
to break
keys per year.

ts five
RS, RC6,
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."

2004–2008: eSTREAM competition for stream ciphers.

2007–2012: SHA-3 competition.

2013–now: CAESAR competition.

Main operations in
add round key to
apply **substitution**
$x \mapsto x^{254}$ in $\mathbf{F}_{256}$
to each byte in blo
linearly mix bits ac

e
y

it

ack”

ear.

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.

2007–2012: SHA-3 competition.

2013–now: CAESAR competition.

Main operations in AES: add round key to block; apply **substitution box** $x \mapsto x^{254}$ in $\mathbf{F}_{256}$ 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.

2007–2012: SHA-3 competition.

2013–now: CAESAR competition.

Main operations in AES:
add round key to block;
apply **substitution box** $x \mapsto x^{254}$ in $\mathbf{F}_{256}$ 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.

2007–2012: SHA-3 competition.

2013–now: CAESAR competition.

Main operations in AES:
add round key to block;
apply **substitution box** $x \mapsto x^{254}$ in $\mathbf{F}_{256}$ 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."

2004–2008: eSTREAM competition for stream ciphers.

2007–2012: SHA-3 competition.

2013–now: CAESAR competition.

Main operations in AES: add round key to block; apply **substitution box** $x \mapsto x^{254}$ in $\mathbf{F}_{256}$ 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?

IST, advised by NSA,
Rijndael as AES.

y was the most important
the evaluation"—Really?

el appears to offer an
e security margin. . . .
appears to offer a
urity margin."

008: eSTREAM
tion for stream ciphers.

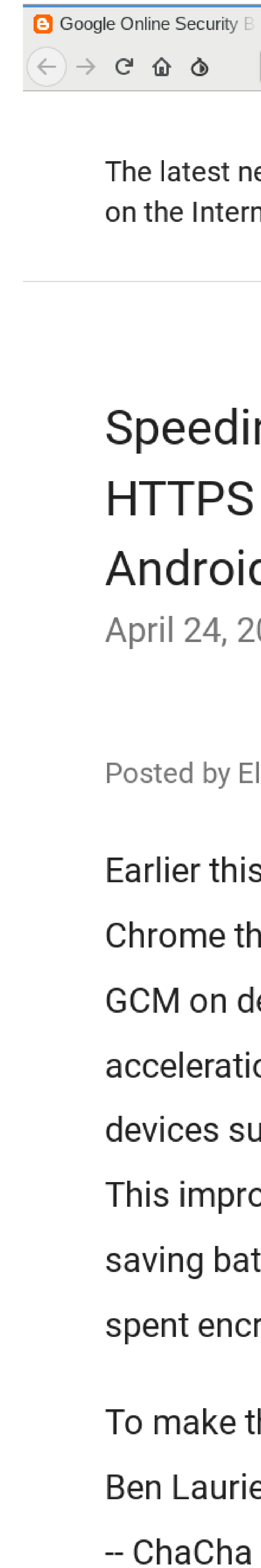
012: SHA-3 competition.

ow: CAESAR competition.

Main operations in AES:
add round key to block;
apply **substitution box**
$x \mapsto x^{254}$ in $\mathbf{F}_{256}$
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?



sed by NSA,
AES.

most important
ation"—Really?

to offer an
margin. . . .
o offer a
gin."

EAM
ream ciphers.

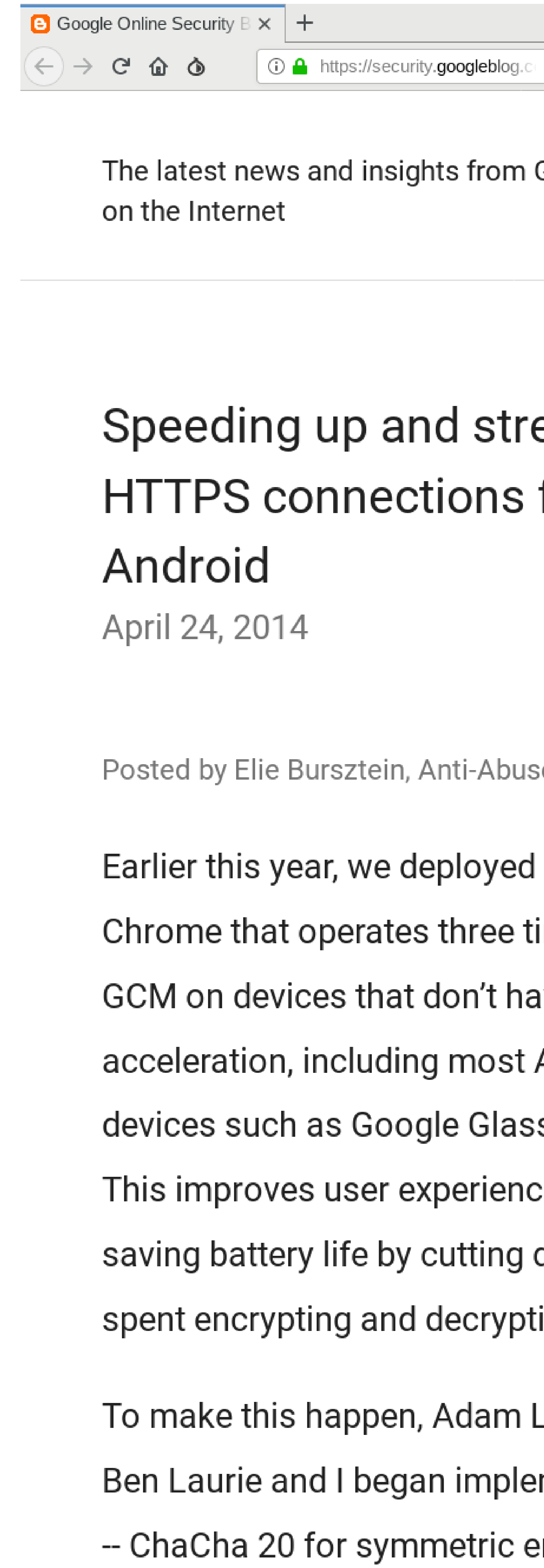
3 competition.

AR competition.

Main operations in AES:
add round key to block;
apply **substitution box**
$x \mapsto x^{254}$ in $\mathbf{F}_{256}$
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?



A,

ortant
eally?

n

. .

ers.

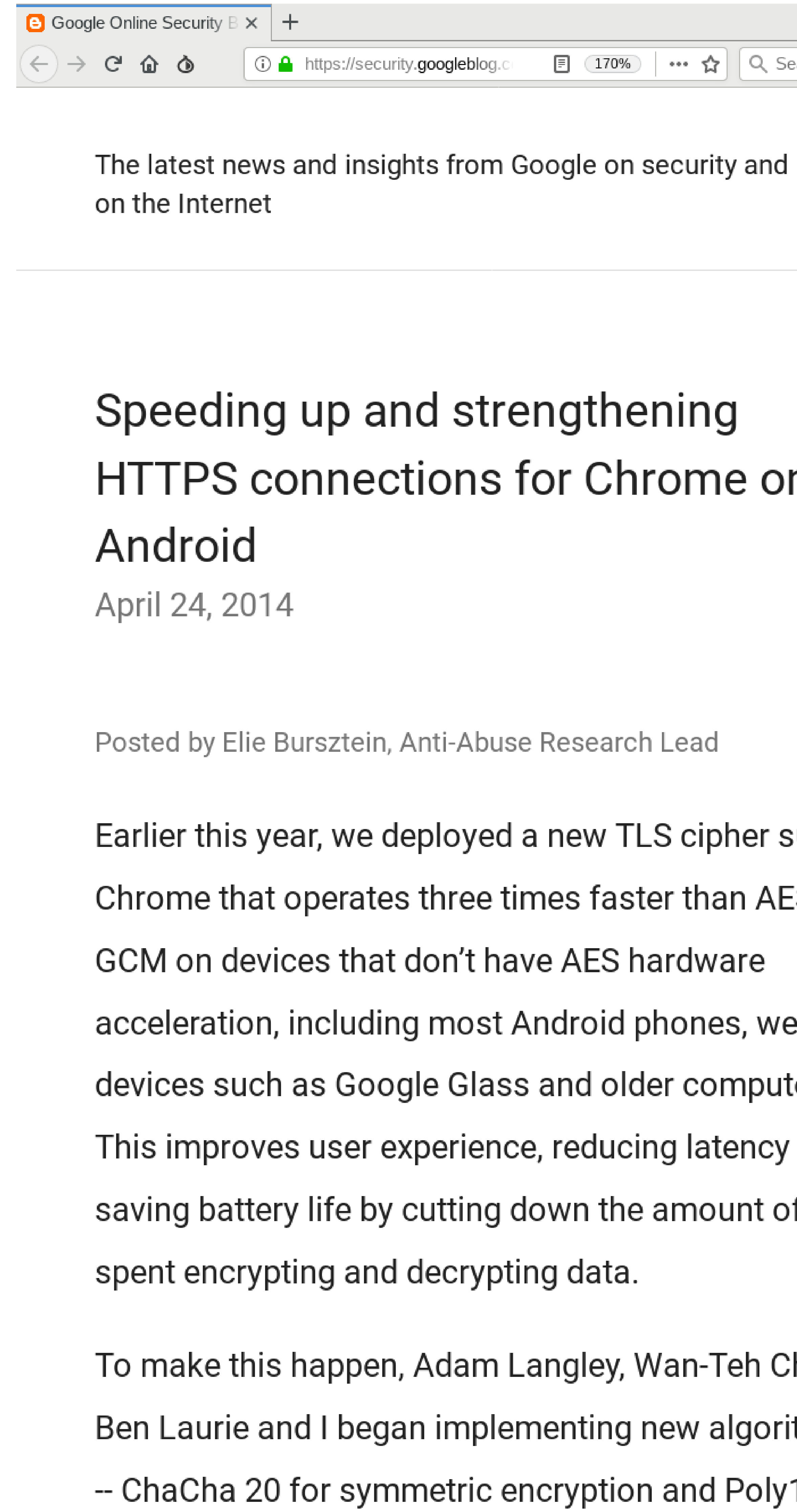
tion.

tition.

Main operations in AES:
add round key to block;
apply **substitution box**
$x \mapsto x^{254}$ in $\mathbf{F}_{256}$
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.

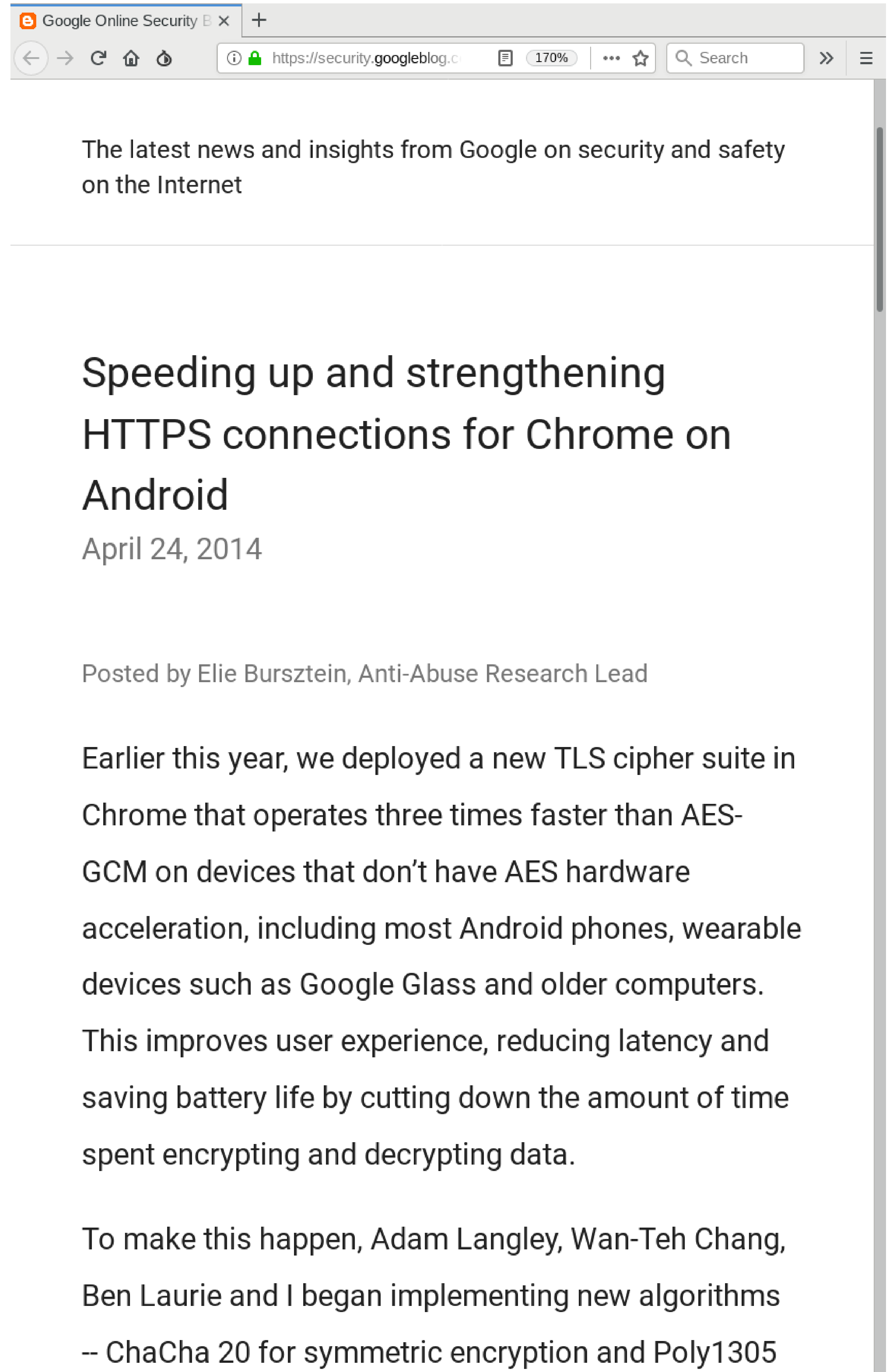
So why isn't AES-256 the end
of the symmetric-crypto story?



Main operations in AES:
add round key to block;
apply **substitution box**
$x \mapsto x^{254}$ in $\mathbf{F}_{256}$
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?



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

erations in AES:
nd key to block;
bstitution box
^{4} in $\mathbf{F}_{256}$
byte in block;
mix bits across block.

e security analysis.
us threats to AES-256
rget SPRP security
mplies PRP security),
a post-quantum world.

isn't AES-256 the end
ymmetric-crypto story?

Google Online Security B

https://security.googleblog.c   170%   Search

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

Date:
Message
[Downloa

From: Er

Hi all,

(Please
it to be

It was c
encrypti
storage
"Androic
these de
have to
Cryptogr

As we ex
challeng
the very
suitable
Speck, i
has a la

Therefor
encrypti
ChaCha s
paper he

n AES:
block;
n box

ock;
cross block.

analysis.
to AES-256
? security
P security),
ntum world.

256 the end
crypto story?

Google Online Security B ×

https://security.googleblog.c 170% Search

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


```
Date:        201
Message-ID:  201
[Download message

From: Eric Biggers

Hi all,

(Please note that
it to be merged qu

It was officially
encryption [1].  W
storage encryption
"Android Go" devic
these devices stil
have to use older
Cryptography Exten

As we explained in
challenging proble
the very strict pe
suitable for pract
Speck, in this day
has a large politi

Therefore, we (wel
encryption mode, H
ChaCha stream ciph
paper here: https:
```

k.

56

),

ld.

nd

ry?

36

Google Online Security B ×

https://security.googleblog.c 170% Search

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

37

```
Date:        2018-08-06 2
Message-ID:  201808062233
[Download message RAW]

From: Eric Biggers <ebiggers

Hi all,

(Please note that this patch
it to be merged quite yet!)

It was officially decided to
encryption [1].  We've been
storage encryption to entry-
"Android Go" devices sold in
these devices still ship wit
have to use older CPUs like
Cryptography Extensions, mak

As we explained in detail ea
challenging problem due to t
the very strict performance
suitable for practical use i
Speck, in this day and age t
has a large political elemen

Therefore, we (well, Paul Cr
encryption mode, HPolyC.  In
ChaCha stream cipher for dis
paper here: https://eprint.i
```

Google Online Security B × +

https://security.googleblog.c 170% Search

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

Date: 2018-08-06 22:32:51
Message-ID: 20180806223300.11389
[Download message RAW]

From: Eric Biggers <ebiggers@google.co

Hi all,

(Please note that this patchset is a t
it to be merged quite yet!)

It was officially decided to *not* all
encryption [1]. We've been working to
storage encryption to entry-level Andr
"Android Go" devices sold in developin
these devices still ship with no encry
have to use older CPUs like ARM Cortex
Cryptography Extensions, making AES-XT

As we explained in detail earlier, e.g
challenging problem due to the lack of
the very strict performance requiremen
suitable for practical use in dm-crypt
Speck, in this day and age the choice
has a large political element, restric

Therefore, we (well, Paul Crowley did
encryption mode, HPolyC. In essence,
ChaCha stream cipher for disk encrypti
paper here: https://eprint.iacr.org/20

37

https://security.googleblog.c | 170% | Search

ews and insights from Google on security and safety
net

ng up and strengthening connections for Chrome on d

014

ie Bursztein, Anti-Abuse Research Lead

s year, we deployed a new TLS cipher suite in
at operates three times faster than AES-
evices that don't have AES hardware
on, including most Android phones, wearable
ch as Google Glass and older computers.
oves user experience, reducing latency and
tery life by cutting down the amount of time
ypting and decrypting data.

his happen, Adam Langley, Wan-Teh Chang,
e and I began implementing new algorithms
20 for symmetric encryption and Poly1305

```
Date:        2018-08-06 22:32:51
Message-ID:  20180806223300.113891-1-ebigg
[Download message RAW]

From: Eric Biggers <ebiggers@google.com>

Hi all,

(Please note that this patchset is a true RFC, i.
it to be merged quite yet!)

It was officially decided to *not* allow Android
encryption [1].  We've been working to find an al
storage encryption to entry-level Android devices
"Android Go" devices sold in developing countries
these devices still ship with no encryption, sinc
have to use older CPUs like ARM Cortex-A7; and th
Cryptography Extensions, making AES-XTS much too

As we explained in detail earlier, e.g. in [2], t
challenging problem due to the lack of encryption
the very strict performance requirements, while s
suitable for practical use in dm-crypt and fscryp
Speck, in this day and age the choice of cryptogr
has a large political element, restricting the op

Therefore, we (well, Paul Crowley did the real wo
encryption mode, HPolyC.  In essence, HPolyC make
ChaCha stream cipher for disk encryption.  HPoly(
paper here: https://eprint.iacr.org/2018/720.pdf
```

37

Google on security and safety

engthening
for Chrome on

e Research Lead

a new TLS cipher suite in
mes faster than AES-
ve AES hardware
Android phones, wearable
s and older computers.
e, reducing latency and
down the amount of time
ng data.

Langley, Wan-Teh Chang,
menting new algorithms
ncryption and Poly1305

Date: 2018-08-06 22:32:51
Message-ID: 20180806223300.113891-1-ebiggers () ke
[Download message RAW]

From: Eric Biggers <ebiggers@google.com>

Hi all,

(Please note that this patchset is a true RFC, i.e. we're n
it to be merged quite yet!)

It was officially decided to *not* allow Android devices to
encryption [1]. We've been working to find an alternative
storage encryption to entry-level Android devices like the
"Android Go" devices sold in developing countries. Unfortu
these devices still ship with no encryption, since for cost
have to use older CPUs like ARM Cortex-A7; and these CPUs l
Cryptography Extensions, making AES-XTS much too slow.

As we explained in detail earlier, e.g. in [2], this is a v
challenging problem due to the lack of encryption algorithm
the very strict performance requirements, while still being
suitable for practical use in dm-crypt and fscrypt. And as
Speck, in this day and age the choice of cryptographic prim
has a large political element, restricting the options even

Therefore, we (well, Paul Crowley did the real work) design
encryption mode, HPolyC. In essence, HPolyC makes it secur
ChaCha stream cipher for disk encryption. HPolyC is specif
paper here: https://eprint.iacr.org/2018/720.pdf ("HPolyC:

arch

safety

uite in
S-

arable
ers.
and
f time

hang,
thms
1305

37

Date: 2018-08-06 22:32:51
Message-ID: 20180806223300.113891-1-ebiggers () kernel ! o
[Download message RAW]

From: Eric Biggers <ebiggers@google.com>

Hi all,

(Please note that this patchset is a true RFC, i.e. we're not ready f
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 bri
storage encryption to entry-level Android devices like the inexpensiv
"Android Go" devices sold in developing countries. Unfortunately, of
these devices still ship with no encryption, since for cost reasons t
have to use older CPUs like ARM Cortex-A7; and these CPUs lack the AR
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 mee
the very strict performance requirements, while still being secure an
suitable for practical use in dm-crypt and fscrypt. And as we saw wi
Speck, in this day and age the choice of cryptographic primitives als
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 t
ChaCha stream cipher for disk encryption. HPolyC is specified by our
paper here: https://eprint.iacr.org/2018/720.pdf ("HPolyC:

Date: 2018-08-06 22:32:51
Message-ID: 20180806223300.113891-1-ebiggers () kernel ! org
[Download message RAW]

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, 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 ("HPolyC:

2018-08-06 22:32:51

e-ID: 20180806223300.113891-1-ebiggers () kernel ! org

ad message RAW]

ric Biggers <ebiggers@google.com>

note that this patchset is a true RFC, i.e. we're not ready for
e merged quite yet!)

officially decided to *not* allow Android devices to use Speck
ion [1]. We've been working to find an alternative way to bring
encryption to entry-level Android devices like the inexpensive
d Go" devices sold in developing countries. Unfortunately, often
evices still ship with no encryption, since for cost reasons they
use older CPUs like ARM Cortex-A7; and these CPUs lack the ARMv8
raphy Extensions, making AES-XTS much too slow.

xplained in detail earlier, e.g. in [2], this is a very
ging problem due to the lack of encryption algorithms that meet
strict performance requirements, while still being secure and
e for practical use in dm-crypt and fscrypt. And as we saw with
in this day and age the choice of cryptographic primitives also
arge political element, restricting the options even further.

re, we (well, Paul Crowley did the real work) designed a new
ion mode, HPolyC. In essence, HPolyC makes it secure to use the
stream cipher for disk encryption. HPolyC is specified by our
ere: https://eprint.iacr.org/2018/720.pdf ("HPolyC:

39

AES per
in both
by small
heavy S-

8-08-06 22:32:51
8080622330 0.113891-1-ebiggers () kernel ! org
RAW]

: <ebiggers@google.com>

this patchset is a true RFC, i.e. we're not ready for
ite yet!)

decided to *not* allow Android devices to use Speck
/e've been working to find an alternative way to bring
to entry-level Android devices like the inexpensive
es sold in developing countries. Unfortunately, often
l ship with no encryption, since for cost reasons they
CPUs like ARM Cortex-A7; and these CPUs lack the ARMv8
sions, making AES-XTS much too slow.

detail earlier, e.g. in [2], this is a very
m due to the lack of encryption algorithms that meet
rformance requirements, while still being secure and
ical use in dm-crypt and fscrypt. And as we saw with
and age the choice of cryptographic primitives also
cal element, restricting the options even further.

l, Paul Crowley did the real work) designed a new
PolyC. In essence, HPolyC makes it secure to use the
er for disk encryption. HPolyC is specified by our
//eprint.iacr.org/2018/720.pdf ("HPolyC:

39

AES performance
in both hardware a
by small 128-bit b
heavy S-box desig

22:32:51
300.113891-1-ebiggers () kernel ! org

@google.com>

set is a true RFC, i.e. we're not ready for

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
h no encryption, since for cost reasons they
ARM Cortex-A7; and these CPUs lack the ARMv8
ing AES-XTS much too slow.

rlier, e.g. in [2], this is a very
he lack of encryption algorithms that meet
requirements, while still being secure and
n dm-crypt and fscrypt. And as we saw with
he choice of cryptographic primitives also
t, restricting the options even further.

owley 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 lim
in both hardware and softwa
by small 128-bit block size,
heavy S-box design strategy.

1-1-ebiggers () kernel ! org

m>

rue RFC, i.e. we're not ready for

ow 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
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.

1-1-ebiggers () kernel ! org

m>

rue RFC, i.e. we're not ready for

ow 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
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.

1-1-ebiggers () kernel ! org

m>

rue RFC, i.e. we're not ready for

ow 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
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.

```
ers () kernel ! org

.e. we're not ready for

 devices to use Speck
lternative way to bring
s like the inexpensive
s.  Unfortunately, often
ce for cost reasons they
hese CPUs lack the ARMv8
 slow.

this is a very
n algorithms that meet
still being secure and
ot.  And as we saw with
raphic primitives also
ptions even further.

ork) designed a new
es it secure to use the
C is specified by our
 ("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.

ChaCha
with mu

ernel ! org

not ready for

o use Speck
way to bring
inexpensive
unately, often
t reasons they
lack the ARMv8

very
ms that meet
g secure and
s we saw with
mitives also
n further.

ned a new
re to use the
fied by our

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 sa
with much less wo

rg

or

ng
e
ten
hey
Mv8

t
d
th
o

he

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 system
with much less work than A

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.

rformance seems limited
hardware and software
128-bit block size,
-box design strategy.

tware ecosystem is
ated and dangerous.
tware implementations
S-box often leak
through timing.

is worse for high-security
cated ciphers. 128-bit
ze limits PRF security.
ounds 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.

Next slic
from 201
Lucks–M
Schneide
Todo–Vi
cross-pla

Gimli pe

seems limited
and software
lock size,
n strategy.

ystem is
langerous.
ementations
n leak
ming.

r high-security
ers. 128-bit
RF security.
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.

Next slides: refere
from 2017 Bernste
Lucks–Massolino–
Schneider–Schwab
Todo–Viguier for
cross-platform per

Gimli permutes {0

ited
are
.
ons
urity
bit
ty.
dit.

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.

Next slides: reference softwa
from 2017 Bernstein–Kölbl–
Lucks–Massolino–Mendel–N
Schneider–Schwabe–Standae
Todo–Viguier for “Gimli: a
cross-platform permutation”

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.

Next slides: reference software from 2017 Bernstein–Kölbl–Lucks–Massolino–Mendel–Nawaz–Schneider–Schwabe–Standaert–Todo–Viguier for “Gimli: a cross-platform permutation”.

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.

Next slides: reference software from 2017 Bernstein–Kölbl–Lucks–Massolino–Mendel–Nawaz–Schneider–Schwabe–Standaert–Todo–Viguier for "Gimli: a cross-platform permutation".

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.

Next slides: reference software from 2017 Bernstein–Kölbl–Lucks–Massolino–Mendel–Nawaz–Schneider–Schwabe–Standaert–Todo–Viguier for "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: $(k, 0) \oplus \text{Gimli}(k, m)$.

creates safe systems
ch less work than AES.

amples of how symmetric
es have been improving
implicity, security:

NT is better than DES.

s better than
nd Speck.

BLAKE2, Ascon
er than MD5, SHA-0,
SHA-256, SHA-512.

Next slides: reference software from 2017 Bernstein–Kölbl–Lucks–Massolino–Mendel–Nawaz–Schneider–Schwabe–Standaert–Todo–Viguier for “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: $(k, 0) \oplus \text{Gimli}(k, m)$.

```
void gi
{
  int r
  uint3

  for (
    for
      x
      y
      z
      b
      b
      b
    }
```

fe systems
rk than AES.

how symmetric
en improving
security:

er than DES.

han

Ascon
D5, SHA-0,
SHA-512.

Next slides: reference software from 2017 Bernstein–Kölbl–Lucks–Massolino–Mendel–Nawaz–Schneider–Schwabe–Standaert–Todo–Viguier for "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: $(k, 0) \oplus \text{Gimli}(k, m)$.

```
void gimli(uint3
{
  int r,c;
  uint32 x,y,z;

  for (r = 24;r
    for (c = 0;c
      x = rotate
      y = rotate
      z =
      b[8+c]=x^(
      b[4+c]=y^x
      b[  c]=z^y
    }
```

s
ES.

metric
ing

ES.

0,

Next slides: reference software from 2017 Bernstein–Kölbl–Lucks–Massolino–Mendel–Nawaz–Schneider–Schwabe–Standaert–Todo–Viguier for "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: $(k, 0) \oplus \text{Gimli}(k, m)$.

```
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],
      y = rotate(b[4+c],
      z =        b[8+c];
      b[8+c]=x^(z<<1)^((y
      b[4+c]=y^x     ^((x
      b[  c]=z^y     ^((x
    }
```

Next slides: reference software from 2017 Bernstein–Kölbl–Lucks–Massolino–Mendel–Nawaz–Schneider–Schwabe–Standaert–Todo–Viguier for "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: $(k, 0) \oplus \text{Gimli}(k, m)$.


```
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);
    }
```

des: reference software
17 Bernstein–Kölbl–
Massolino–Mendel–Nawaz–
er–Schwabe–Standaert–
iguier for "Gimli: a
atform permutation".

ermutes $\{0,1\}^{384}$.

where's the key?"

ansour SPRP mode:
$= k \oplus \text{Gimli}(k \oplus m)$.

haCha PRF mode:
$= (k, m) \oplus \text{Gimli}(k, m)$.

$0) \oplus \text{Gimli}(k, m)$.


```
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
      x=
      x=
    }

    if
      x=
      x=
    }

    if
      b
  }
}
```

nce software
ein–Kölbl–
Mendel–Nawaz–
be–Standaert–
“Gimli: a
mutation”.

$, 1\}^{384}$.

e key?”

RP mode:
li($k \oplus m$).

F mode:
Gimli(k, m).

(k, m).


```
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)
      x=b[0]; b[
      x=b[2]; b[
    }

    if ((r & 3)
      x=b[0]; b[
      x=b[1]; b[
    }

    if ((r & 3)
      b[0] ^= (0
  }
}
```

are

awaz–

ert–

.

:

m).


```
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];
      x=b[2]; b[2]=b[3];
    }

    if ((r & 3) == 2) {
      x=b[0]; b[0]=b[2];
      x=b[1]; b[1]=b[3];
    }

    if ((r & 3) == 0)
      b[0] ^= (0x9e377900
  }
}
```

```
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;
      x=b[2]; b[2]=b[3]; b[3]=x;
    }

    if ((r & 3) == 2) {
      x=b[0]; b[0]=b[2]; b[2]=x;
      x=b[1]; b[1]=b[3]; b[3]=x;
    }

    if ((r & 3) == 0)
      b[0] ^= (0x9e377900 | r);
  }
}
```

```
mli(uint32 *b)

,c;
2 x,y,z;

r = 24;r > 0;--r) {
 (c = 0;c < 4;++c) {
 = rotate(b[  c], 24);
 = rotate(b[4+c],  9);
 =         b[8+c];
[8+c]=x^(z<<1)^((y&z)<<2);
[4+c]=y^x      ^((x|z)<<1);
[  c]=z^y      ^((x&y)<<3);
```



```
    if ((r & 3) == 0) {
      x=b[0]; b[0]=b[1]; b[1]=x;
      x=b[2]; b[2]=b[3]; b[3]=x;
    }

    if ((r & 3) == 2) {
      x=b[0]; b[0]=b[2]; b[2]=x;
      x=b[1]; b[1]=b[3]; b[3]=x;
    }

    if ((r & 3) == 0)
      b[0] ^= (0x9e377900 | r);
  }
}
```

No addit
are repla
(Idea sto

Big rota
quickly a

x, y, z i
changes
(0, 4, 8;

Other sw
through
swaps pe
on a wid

```
2 *b)




> 0;--r) {
 < 4;++c) {
(b[  c], 24);
(b[4+c],  9);
 b[8+c];
z<<1)^((y&z)<<2);
     ^((x|z)<<1);
     ^((x&y)<<3);
```



```
    if ((r & 3) == 0) {
      x=b[0]; b[0]=b[1]; b[1]=x;
      x=b[2]; b[2]=b[3]; b[3]=x;
    }

    if ((r & 3) == 2) {
      x=b[0]; b[0]=b[2]; b[2]=x;
      x=b[1]; b[1]=b[3]; b[3]=x;
    }

    if ((r & 3) == 0)
      b[0] ^= (0x9e377900 | r);
  }
}
```

No additions. Nor
are replaced by sh
(Idea stolen from

Big rotations diffu
quickly across bit

x, y, z interaction
changes quickly th
(0, 4, 8; 1, 5, 9; 2, 6

Other swaps diffus
through rows. Del
swaps per round =
on a wide range of

```
{
 {
24);
 9);

&z)<<2);
|z)<<1);
&y)<<3);
```



```
    if ((r & 3) == 0) {
      x=b[0]; b[0]=b[1]; b[1]=x;
      x=b[2]; b[2]=b[3]; b[3]=x;
    }

    if ((r & 3) == 2) {
      x=b[0]; b[0]=b[2]; b[2]=x;
      x=b[1]; b[1]=b[3]; b[3]=x;
    }

    if ((r & 3) == 0)
      b[0] ^= (0x9e377900 | r);
  }
}
```

No additions. Nonlinear carr
are replaced by shifts of &, |
(Idea stolen from NORX cip

Big rotations diffuse change
quickly across bit positions.

x, y, z interaction diffuses
changes quickly through colu
(0, 4, 8; 1, 5, 9; 2, 6, 10; 3, 7,

Other swaps diffuse changes
through rows. Deliberately l
swaps per round ⇒ faster ro
on a wide range of platforms

```
    if ((r & 3) == 0) {
      x=b[0]; b[0]=b[1]; b[1]=x;
      x=b[2]; b[2]=b[3]; b[3]=x;
    }

    if ((r & 3) == 2) {
      x=b[0]; b[0]=b[2]; b[2]=x;
      x=b[1]; b[1]=b[3]; b[3]=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.