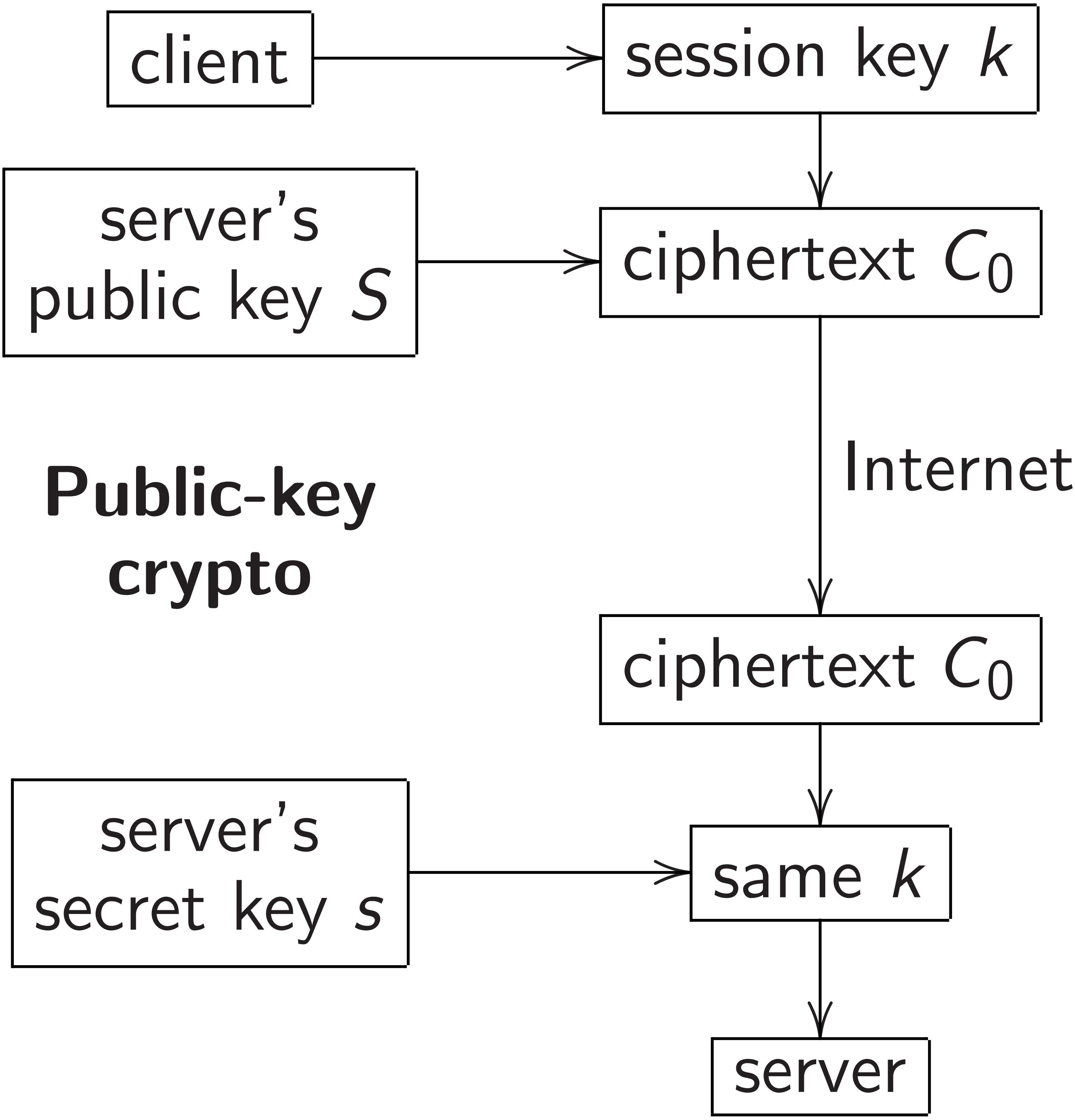
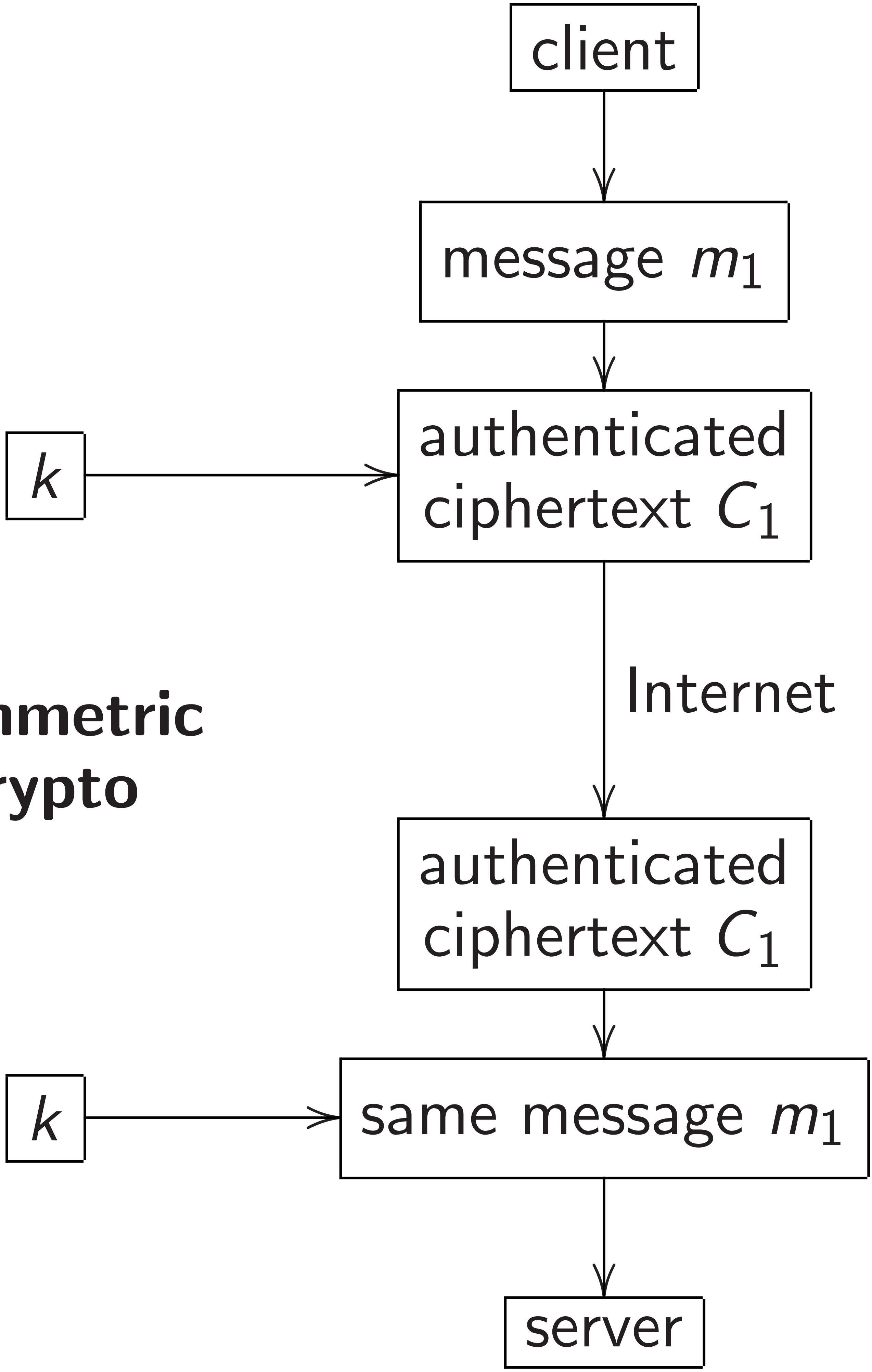
Symmetric crypto, part 2

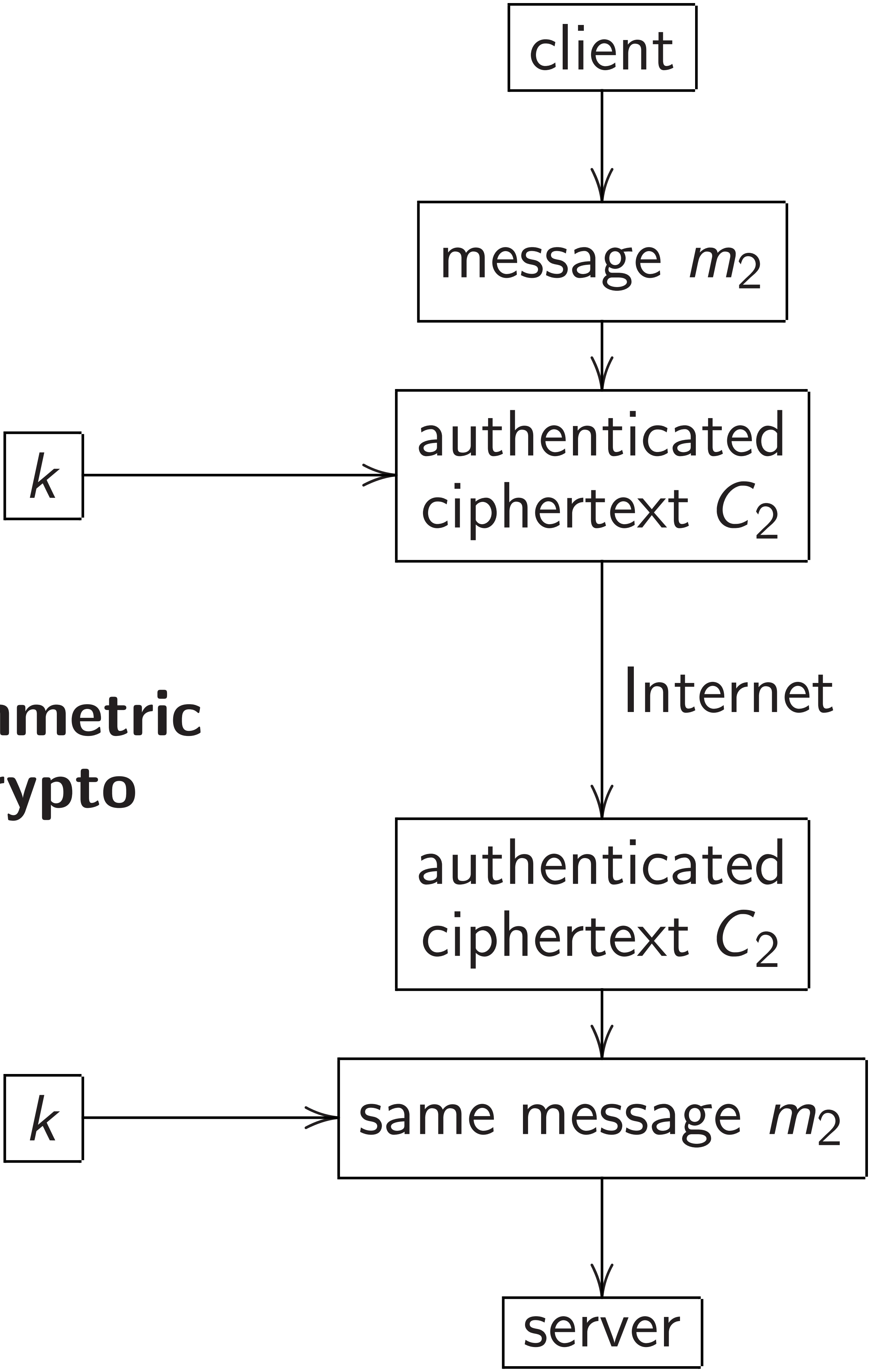
D. J. Bernstein



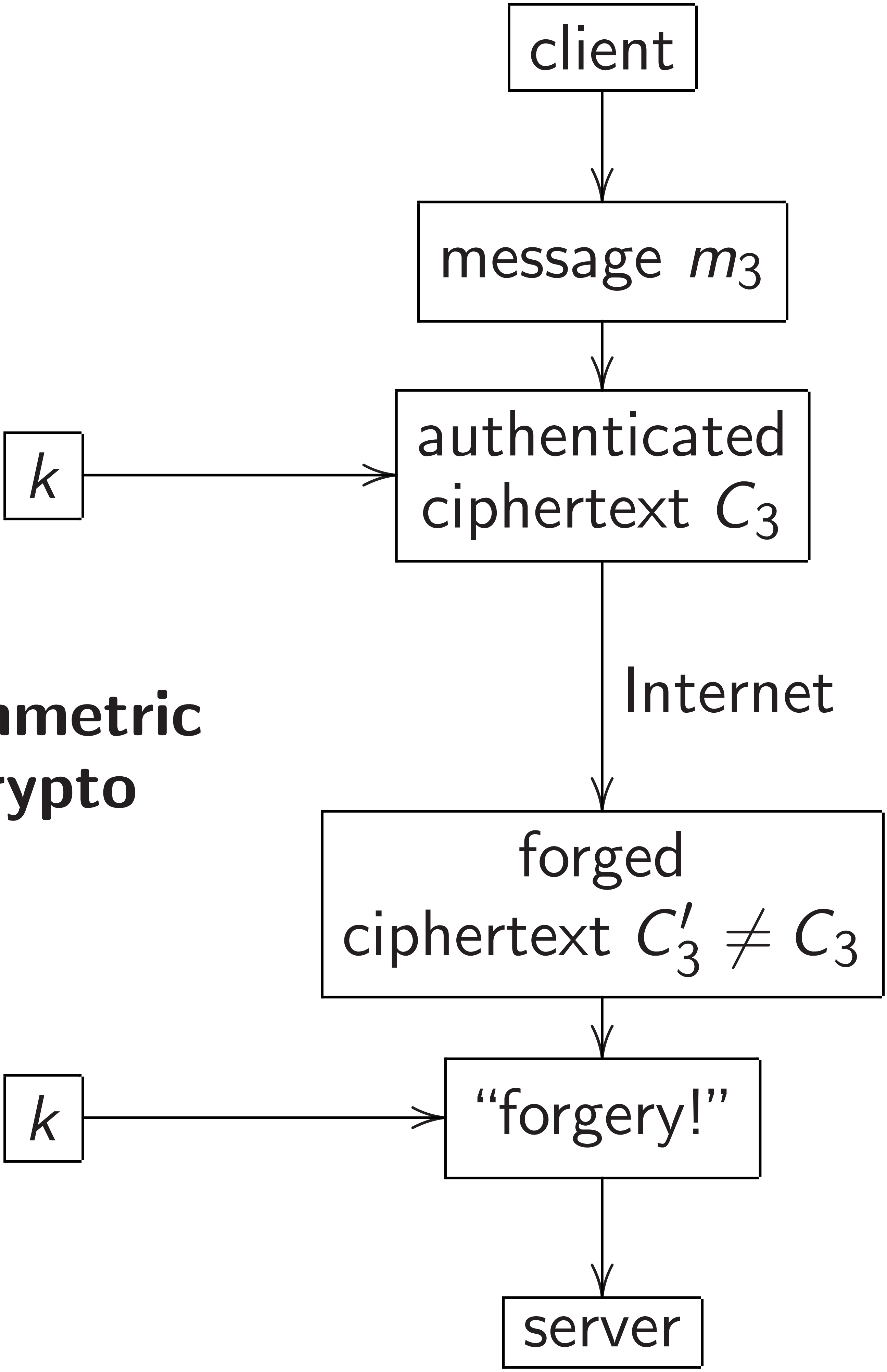
Symmetric crypto



Symmetric crypto



Symmetric crypto



Symmetric crypto: main objectives

Integrity:

Attacker can't forge ciphertexts.

Symmetric crypto: main objectives

Integrity:
Attacker can't forge ciphertexts.

Confidentiality: Attacker seeing ciphertexts can't figure out message contents. (But can see message number, length, timing.)

Symmetric crypto: main objectives

Integrity:
Attacker can't forge ciphertexts.

Confidentiality: Attacker seeing ciphertexts can't figure out message contents. (But can see message number, length, timing.)

Can define further objectives.
Example: If crypto is too slow, attacker can flood server's CPU.
Real client messages are lost.
This damages **availability**.

Easy encryption mechanism:
Assume 30-digit messages.
Assume client, server know
secret 30-digit numbers
t_1 to use for message 1;
t_2 to use for message 2;
t_3 to use for message 3; etc.

Easy encryption mechanism:
Assume 30-digit messages.
Assume client, server know secret 30-digit numbers
t_1 to use for message 1;
t_2 to use for message 2;
t_3 to use for message 3; etc.

$C_1 = (m_1 + t_1) \bmod 10^{30}$;
$C_2 = (m_2 + t_2) \bmod 10^{30}$;
$C_3 = (m_3 + t_3) \bmod 10^{30}$; etc.
This protects confidentiality.

Easy encryption mechanism:
Assume 30-digit messages.
Assume client, server know secret 30-digit numbers
t_1 to use for message 1;
t_2 to use for message 2;
t_3 to use for message 3; etc.

$C_1 = (m_1 + t_1) \bmod 10^{30}$;
$C_2 = (m_2 + t_2) \bmod 10^{30}$;
$C_3 = (m_3 + t_3) \bmod 10^{30}$; etc.
This protects confidentiality.

AES-GCM, ChaCha20-Poly1305 work this way, scaled up to groups larger than $\mathbf{Z}/10^{30}$.

Last time: For each message
compute **authenticator**
using another secret number.

Sender attaches authenticator
to message before sending it.
Receiver checks authenticator.
This protects integrity.

Last time: For each message
compute **authenticator**
using another secret number.

Sender attaches authenticator
to message before sending it.
Receiver checks authenticator.
This protects integrity.

Details use multiplications.
AES-GCM, ChaCha20-Poly1305
work this way, again scaled up.

Last time: For each message compute **authenticator** using another secret number.

Sender attaches authenticator to message before sending it. Receiver checks authenticator. This protects integrity.

Details use multiplications. AES-GCM, ChaCha20-Poly1305 work this way, again scaled up.

This would be the whole picture *if* client, server started with enough secret random numbers.

AES expands 256-bit secret k into $F(k, 1), F(k, 2), F(k, 3), \ldots$ simulating many independent secrets $r, s_1, t_1, \ldots$.

AES expands 256-bit secret k into $F(k, 1), F(k, 2), F(k, 3), \ldots$ simulating many independent secrets $r, s_1, t_1, \ldots$.

ChaCha20 also does this, using a different function F.

AES expands 256-bit secret k into $F(k, 1), F(k, 2), F(k, 3), \ldots$ simulating many independent secrets $r, s_1, t_1, \ldots$.

ChaCha20 also does this, using a different function F.

Definition of **PRG** ("pseudorandom generator"): Attacker can't distinguish $F(k, 1), F(k, 2), F(k, 3), \ldots$ from string of independent uniform random blocks.

AES expands 256-bit secret k into $F(k, 1), F(k, 2), F(k, 3), \ldots$ simulating many independent secrets $r, s_1, t_1, \ldots$.

ChaCha20 also does this, using a different function F.

Definition of **PRG** ("pseudorandom generator"): Attacker can't distinguish $F(k, 1), F(k, 2), F(k, 3), \ldots$ from string of independent uniform random blocks.

Warning: "pseudorandom" has many other meanings.

PRF ("pseudorandom function"):
Attacker can't distinguish
$F(k,1), F(k,2), F(k,3), \ldots$ from
independent uniform random
blocks, given access to a server
that returns $F(k,i)$ given i.
Server is called an **oracle**.

PRF ("pseudorandom function"):
Attacker can't distinguish
$F(k,1), F(k,2), F(k,3), \ldots$ from
independent uniform random
blocks, given access to a server
that returns $F(k,i)$ given i.
Server is called an **oracle**.

PRP ("... permutation"):
Attacker can't distinguish
$F(k,1), F(k,2), F(k,3), \ldots$
from independent uniform random
distinct blocks, given oracle.

PRF ("pseudorandom function"):
Attacker can't distinguish
$F(k,1), F(k,2), F(k,3), \ldots$ from
independent uniform random
blocks, given access to a server
that returns $F(k,i)$ given i.
Server is called an **oracle**.

PRP ("... permutation"):
Attacker can't distinguish
$F(k,1), F(k,2), F(k,3), \ldots$
from independent uniform random
distinct blocks, given oracle.

If block size is big then
PRP $\Rightarrow$ PRF $\Rightarrow$ PRG.

Small block sizes are dangerous.
PRF property fails, and often
application security fails.

Small block sizes are dangerous.
PRF property fails, and often application security fails.

e.g. 2016 Bhargavan–Leurent `sweet32.info`: Triple-DES broken in TLS. Same attack also breaks small block sizes in NSA's Simon, Speck.

Small block sizes are dangerous.
PRF property fails, and often application security fails.

e.g. 2016 Bhargavan–Leurent `sweet32.info`: Triple-DES broken in TLS. Same attack also breaks small block sizes in NSA's Simon, Speck.

AES block size: 128 bits.
PRF attack chance $\approx q^2/2^{129}$ if AES is used for q blocks.
Is this safe? How big is q?

Small block sizes are dangerous.
PRF property fails, and often application security fails.

e.g. 2016 Bhargavan–Leurent `sweet32.info`: Triple-DES broken in TLS. Same attack also breaks small block sizes in NSA's Simon, Speck.

AES block size: 128 bits.
PRF attack chance $\approx q^2/2^{129}$ if AES is used for q blocks.
Is this safe? How big is q?

ChaCha20 block size: 512 bits.

Can prove confidentiality and integrity of AES-GCM and ChaCha20-Poly1305 *assuming* AES and ChaCha20 are PRFs.

Can prove confidentiality and integrity of AES-GCM and ChaCha20-Poly1305 *assuming* AES and ChaCha20 are PRFs.

Generalization: Prove security of $M(F)$ assuming cipher F is a PRF. M is a **mode of use** of F.

Can prove confidentiality and integrity of AES-GCM and ChaCha20-Poly1305 *assuming* AES and ChaCha20 are PRFs.

Generalization: Prove security of $M(F)$ assuming cipher F is a PRF. M is a **mode of use** of F.

Good modes: CTR ("counter mode"), CBC, OFB, many more.

Bad modes: ECB, many more.

Can prove confidentiality and integrity of AES-GCM and ChaCha20-Poly1305 *assuming* AES and ChaCha20 are PRFs.

Generalization: Prove security of $M(F)$ assuming cipher F is a PRF. M is a **mode of use** of F.

Good modes: CTR ("counter mode"), CBC, OFB, many more.

Bad modes: ECB, many more.

Mode that claimed proof but was recently broken: OCB2. Have to check proofs carefully!

How do we know that AES and ChaCha20 are PRFs? **We don't.**

How do we know that AES and ChaCha20 are PRFs? **We don't.**

We **conjecture** security after enough failed attack efforts. "All of these attacks fail and we don't have better attack ideas."

How do we know that AES and ChaCha20 are PRFs? **We don't.**

We **conjecture** security after enough failed attack efforts. "All of these attacks fail and we don't have better attack ideas."

Remaining slides today:

- Simple example of block cipher. Seems to be a good cipher, except block size is too small.
- Variants of this block cipher that look similar but can be quickly broken.

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

Functionality

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

Functionality

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

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

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

Functionality

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

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

Output: 64-bit **ciphertext** (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;
}
```

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);
```

TEA again for comparison

```
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;
}
```

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.

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

But output bits are linear functions of input bits!

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

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

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

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

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

TEA again for comparison

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

LEFTEA: another bad cipher

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

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

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

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.

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.

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.)

TEA again for comparison

```
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;
}
```

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.

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 $2^{1}, 2^{0}$.

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.

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.

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.

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?

TEA again for comparison

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

REPTEA: 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+= . . . .

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

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

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

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

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

Related keys: e.g.,

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

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

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

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?

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.

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: 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.

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 modes using TEA.

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 modes using TEA.

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".