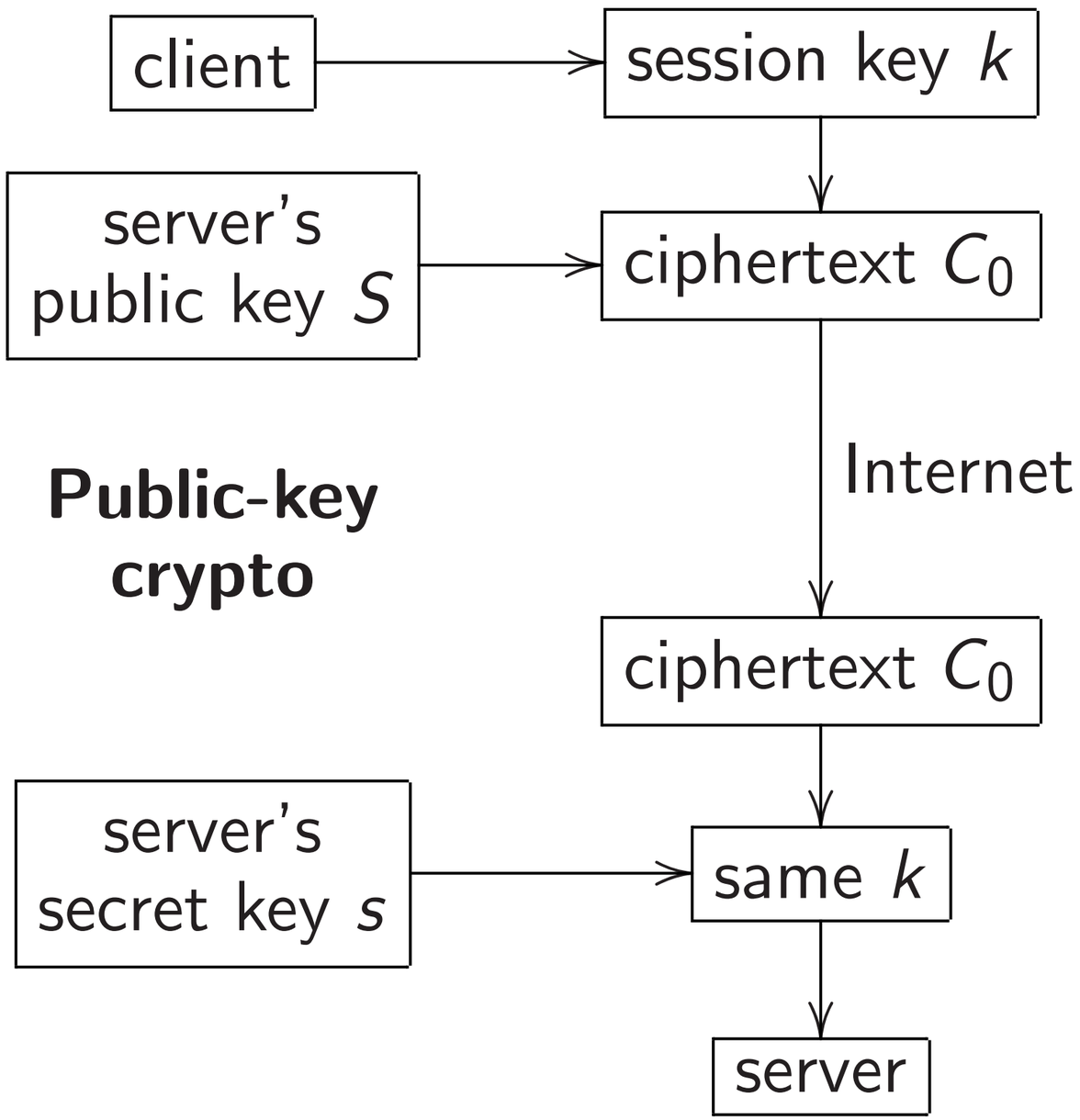
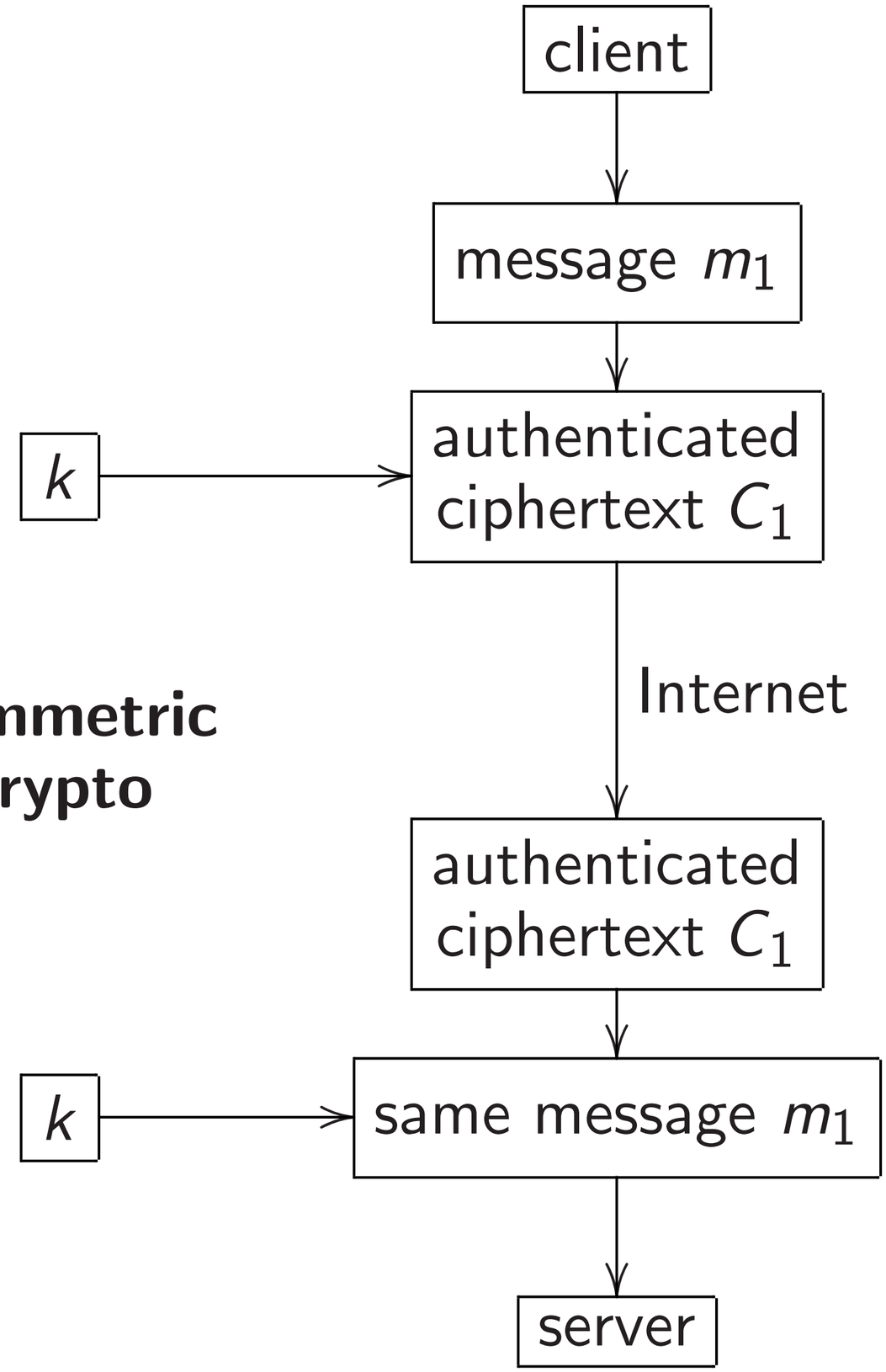
Symmetric crypto, part 2

D. J. Bernstein



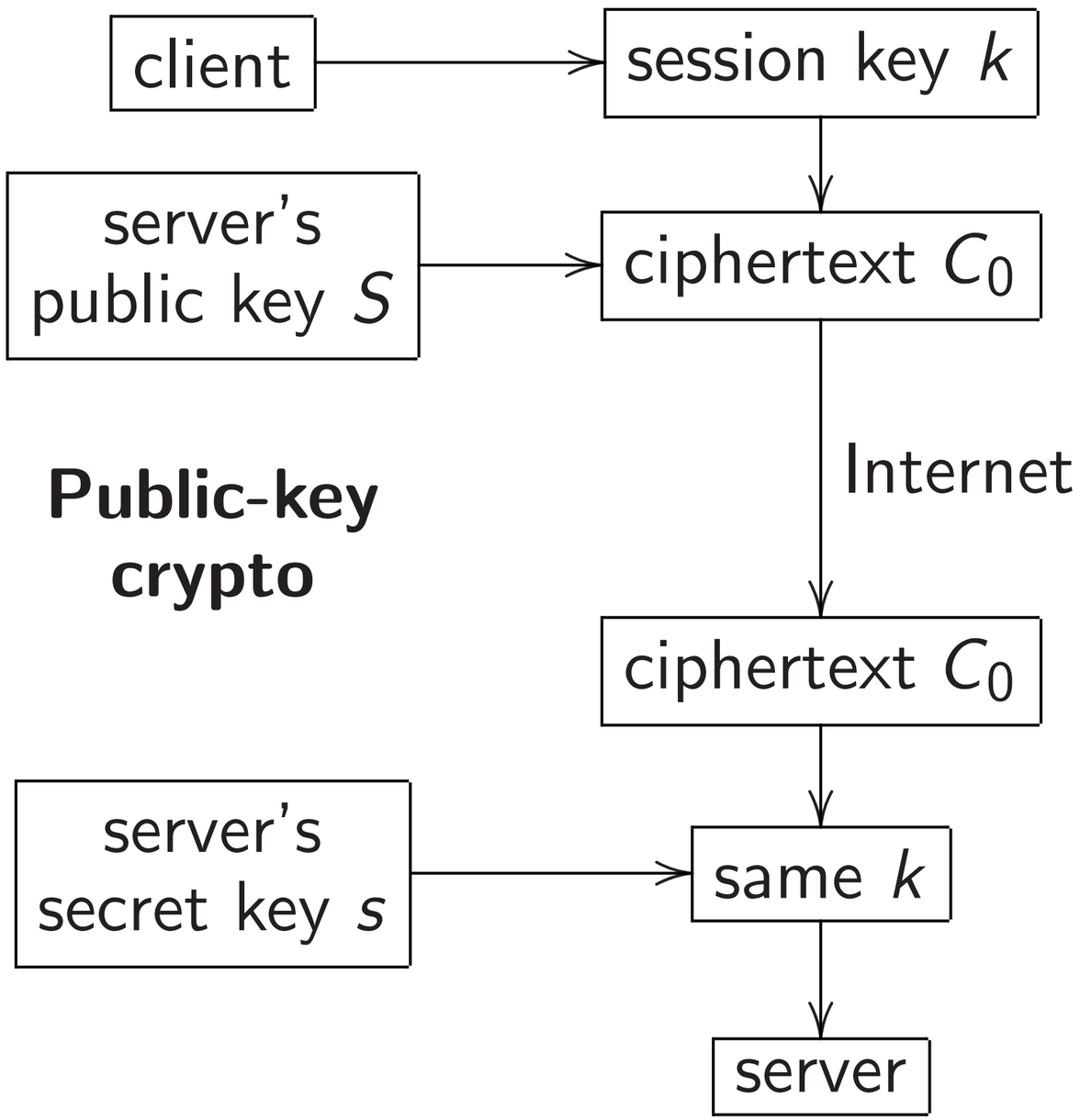
Public-key crypto

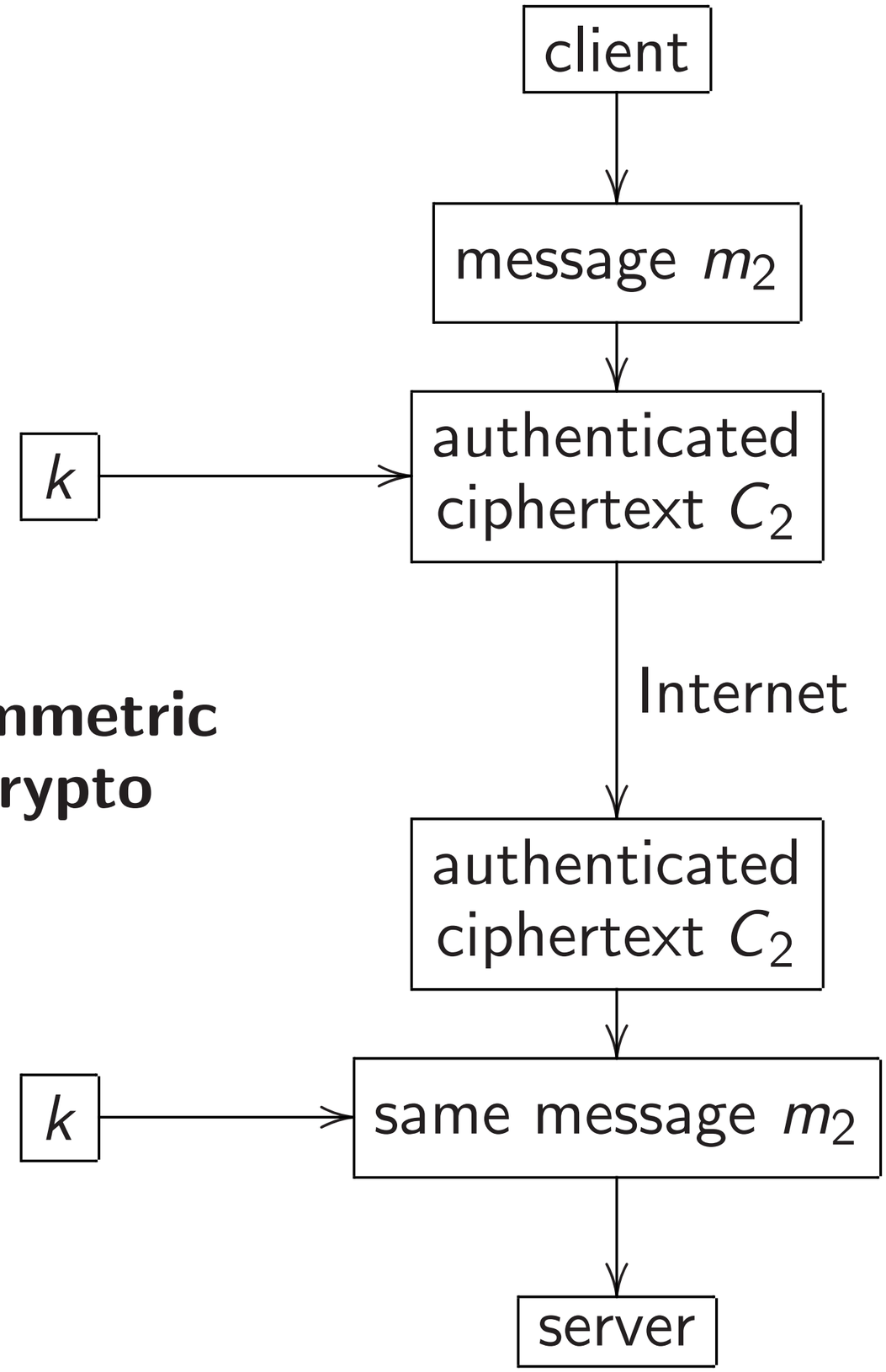


Symmetric crypto

Symmetric crypto, part 2

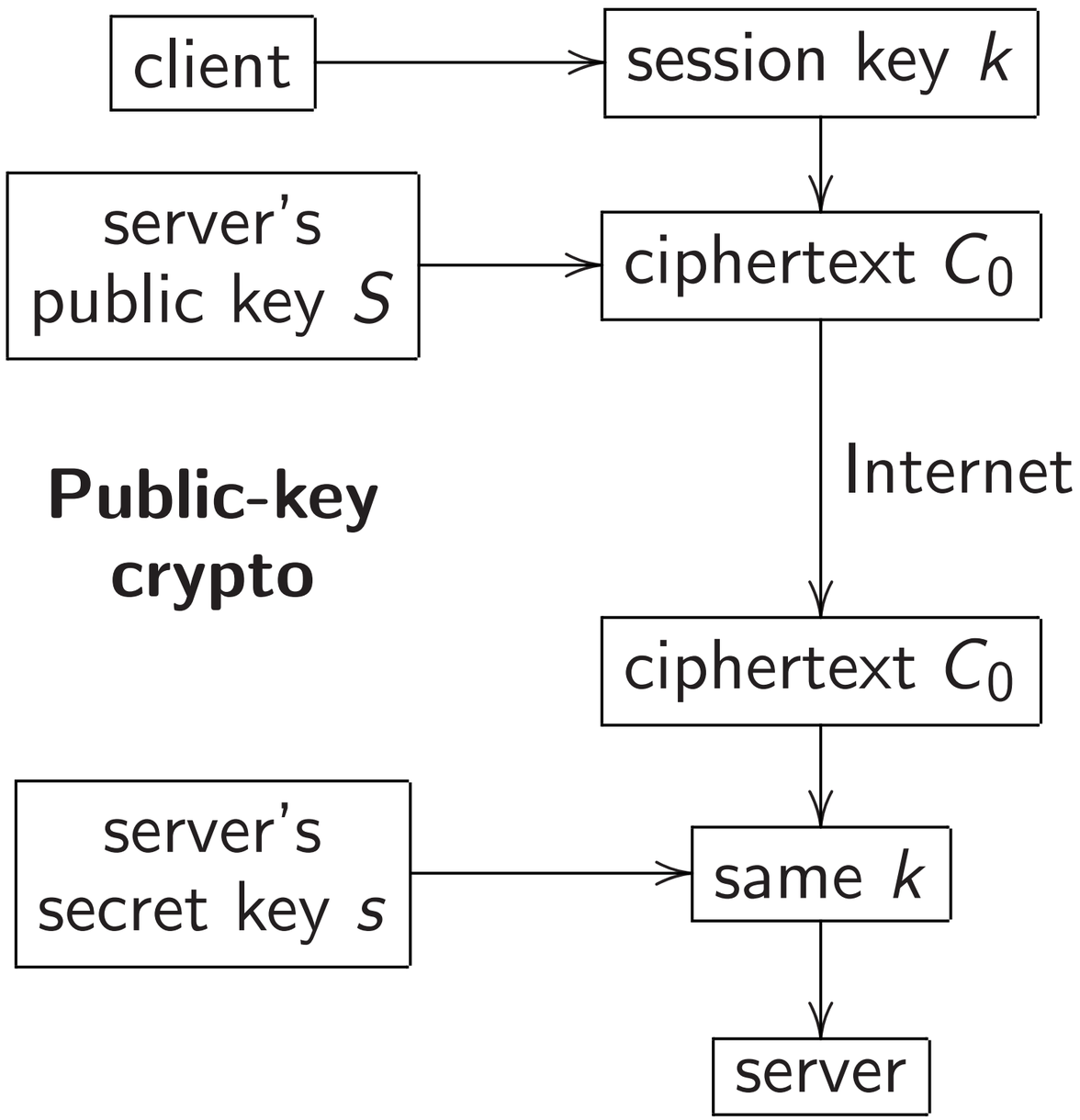
D. J. Bernstein



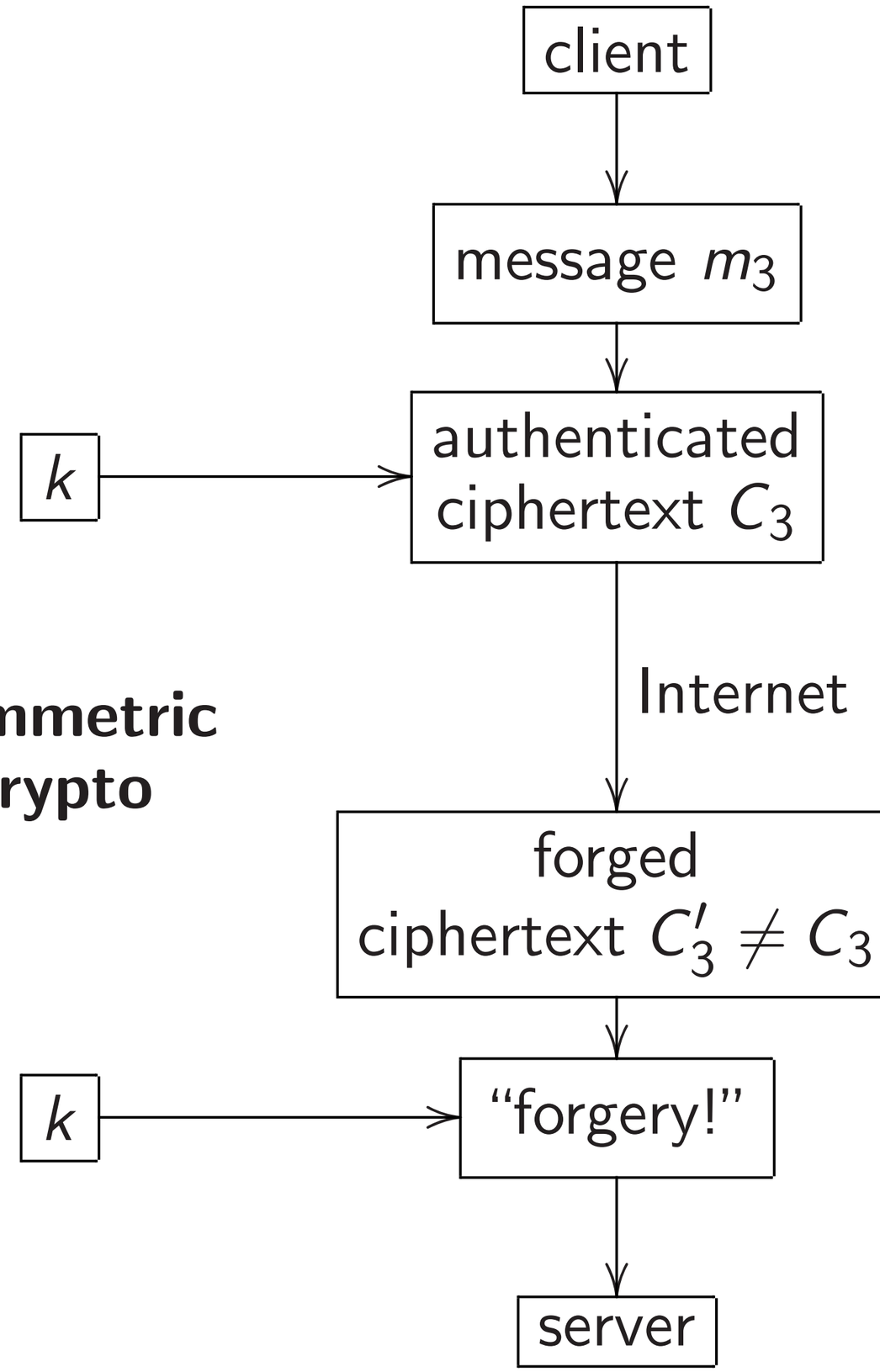


Symmetric crypto, part 2

D. J. Bernstein



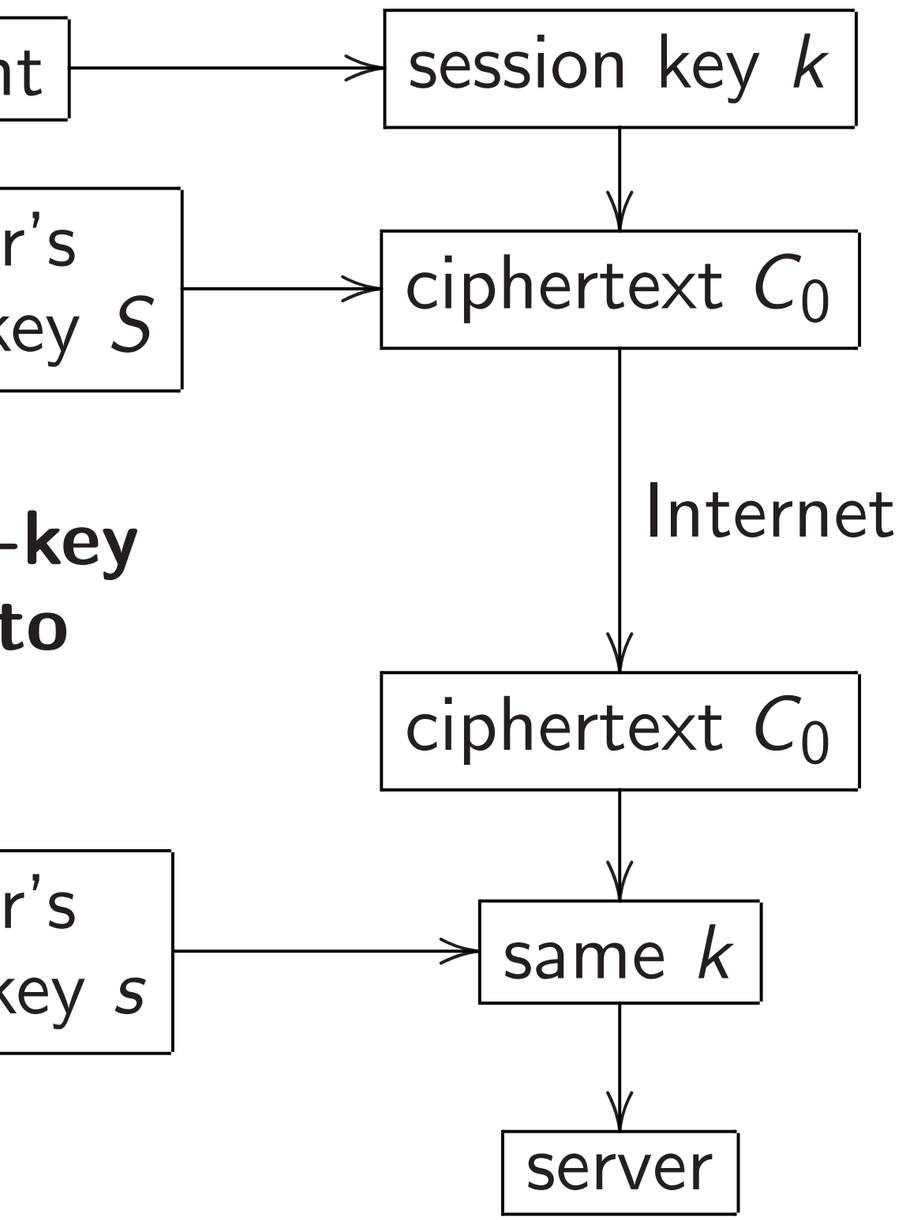
Public-key crypto



Symmetric crypto

ric crypto, part 2

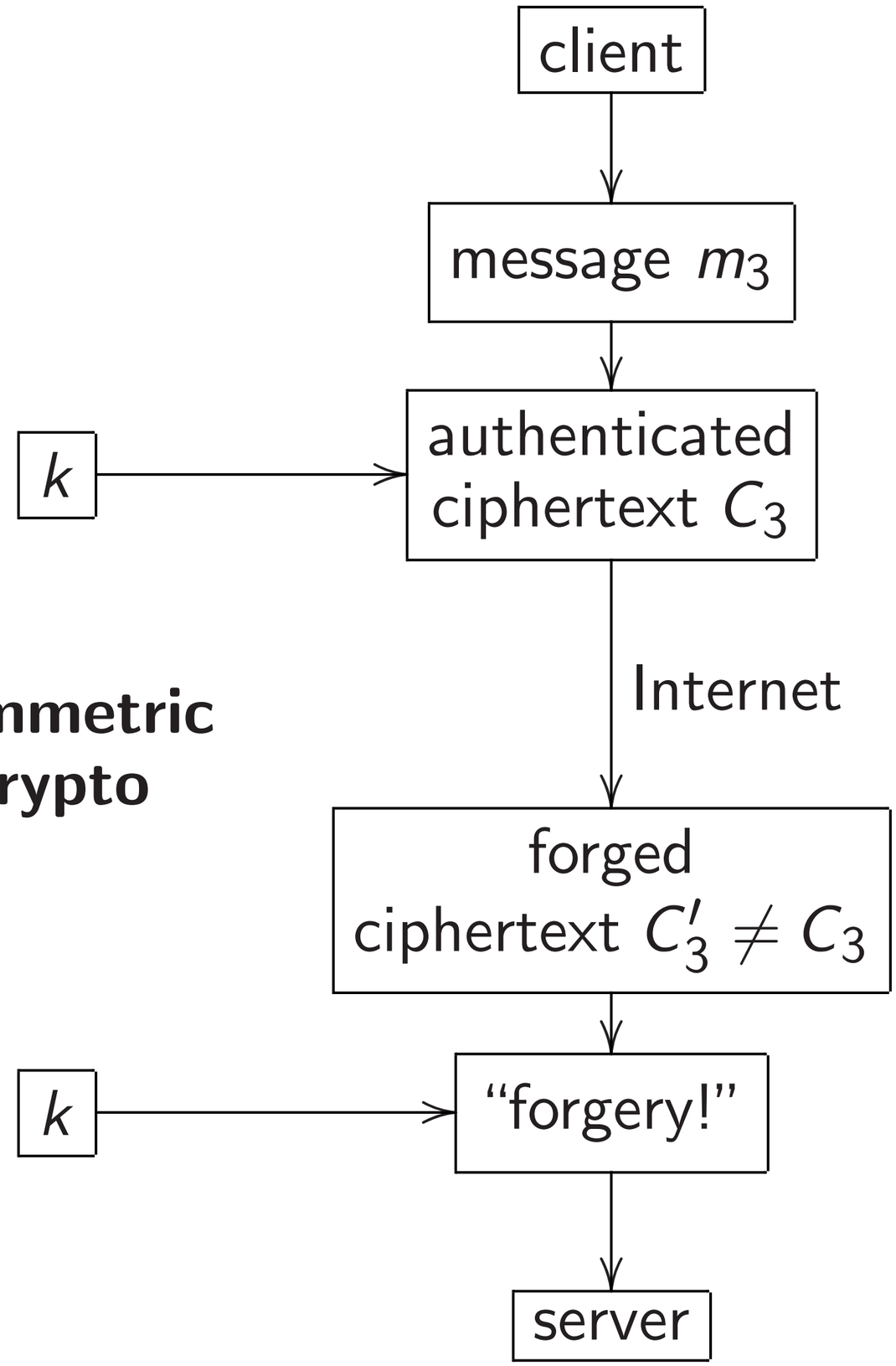
ernstein



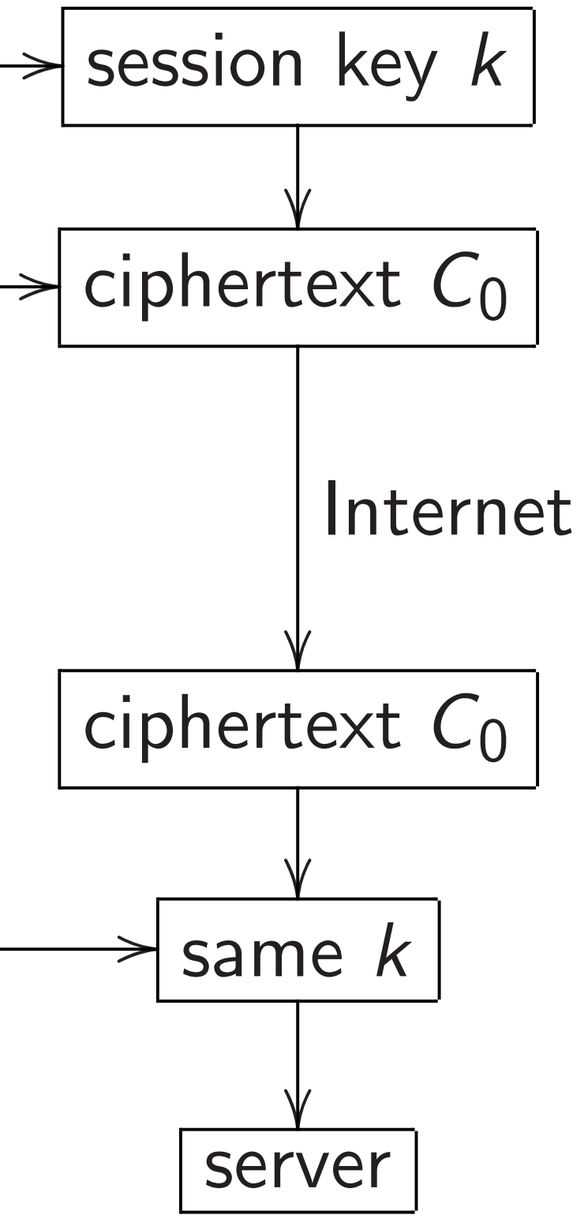
Symmet

Integrit

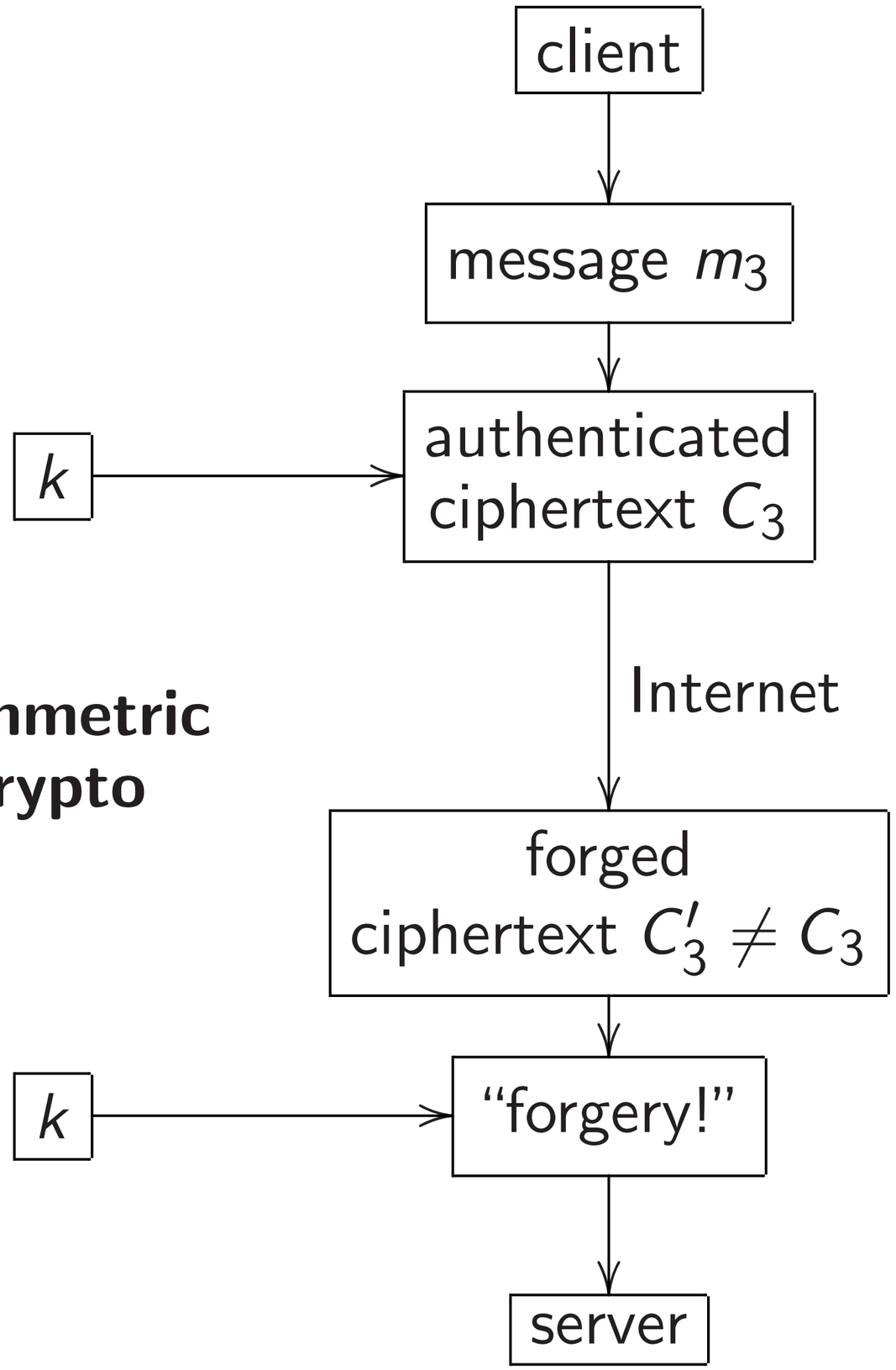
Attacker



part 2



Symmetric crypto



Symmetric crypto:

Integrity:

Attacker can't forg

key k

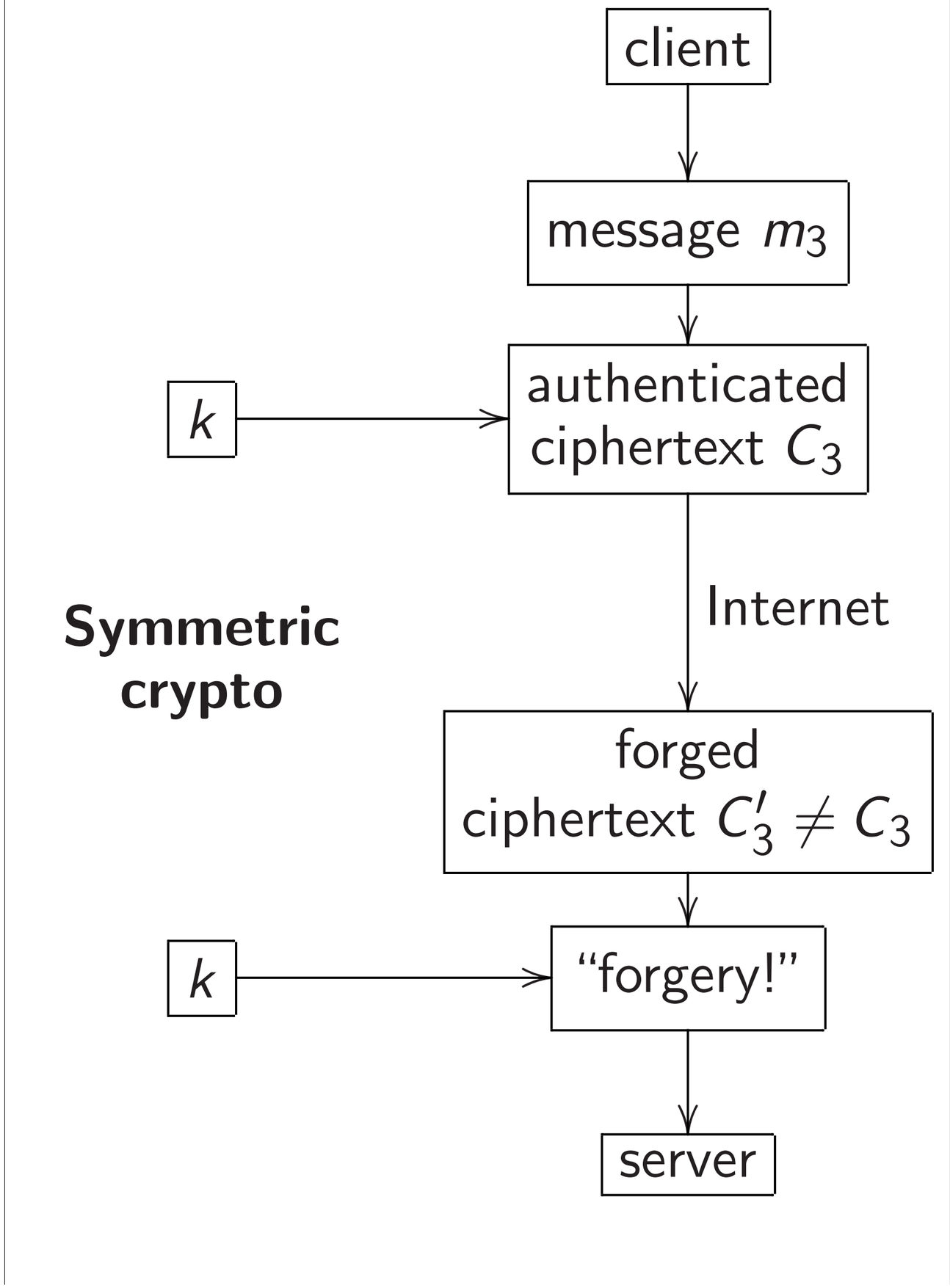
ext C_0

Internet

ext C_0

e k

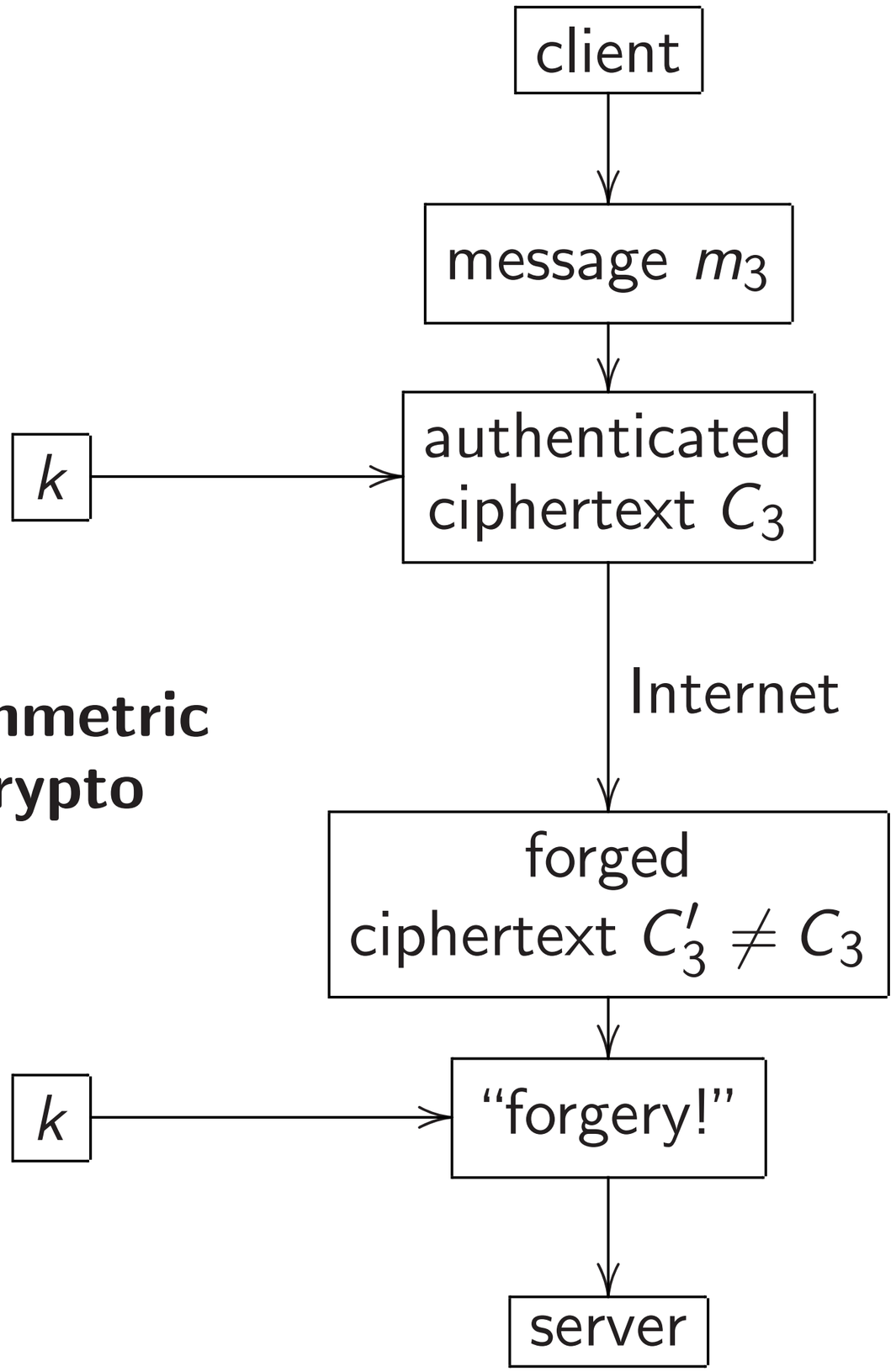
ver



Symmetric crypto: main obj

Integrity:

Attacker can't forge cipherte

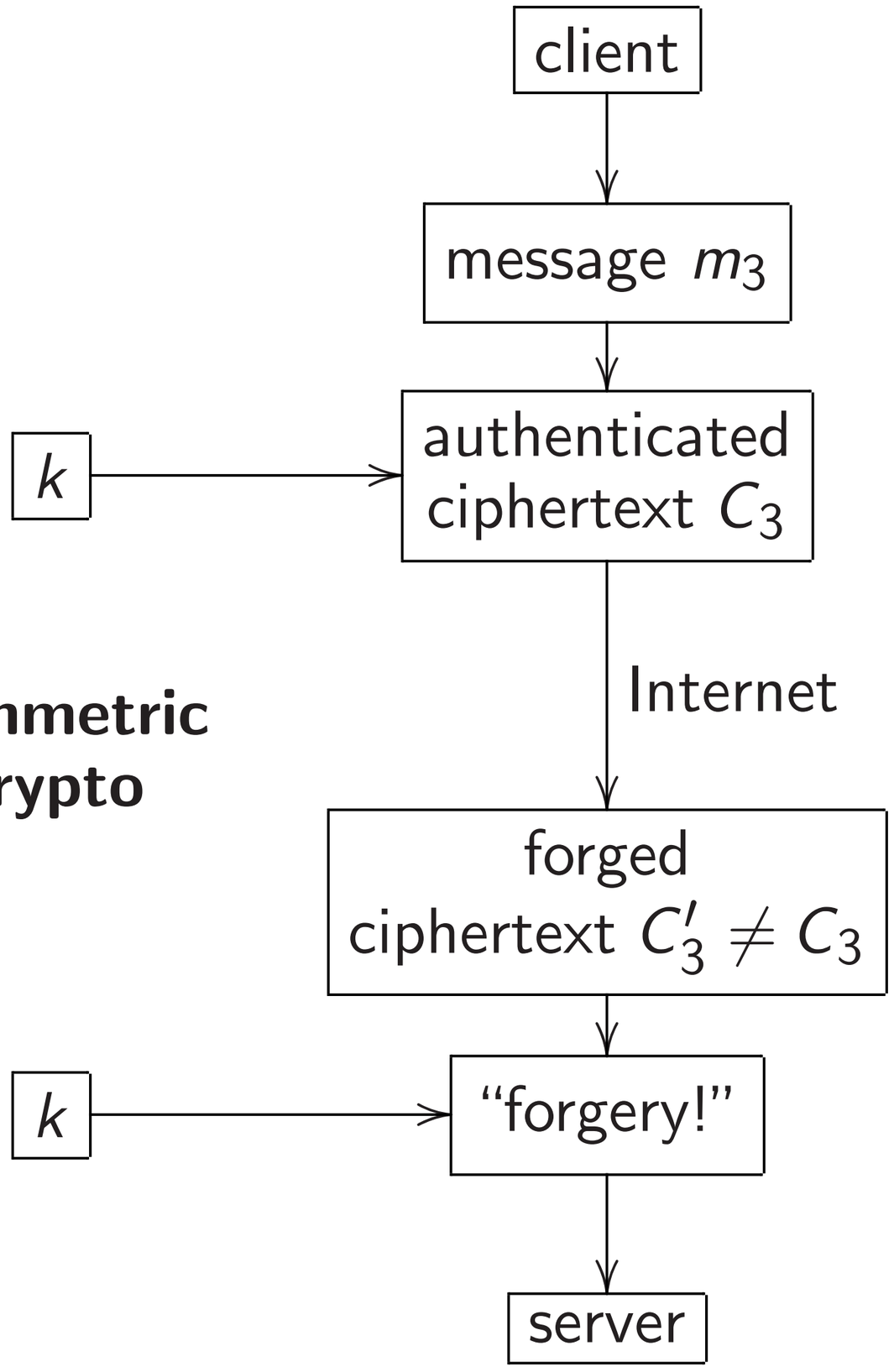


Symmetric crypto

Symmetric crypto: main objectives

Integrity:

Attacker can't forge ciphertexts.

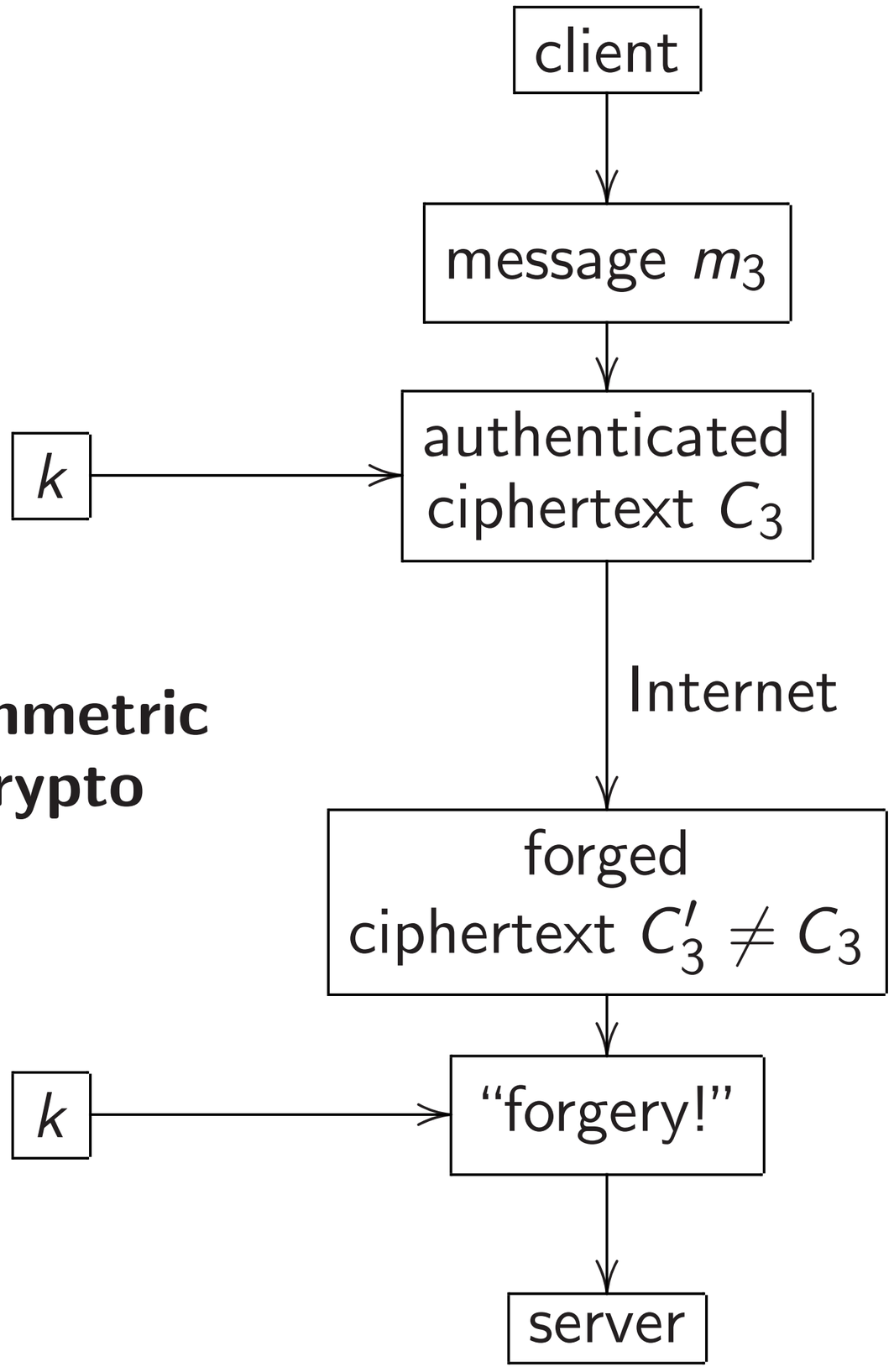


Symmetric crypto

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.)



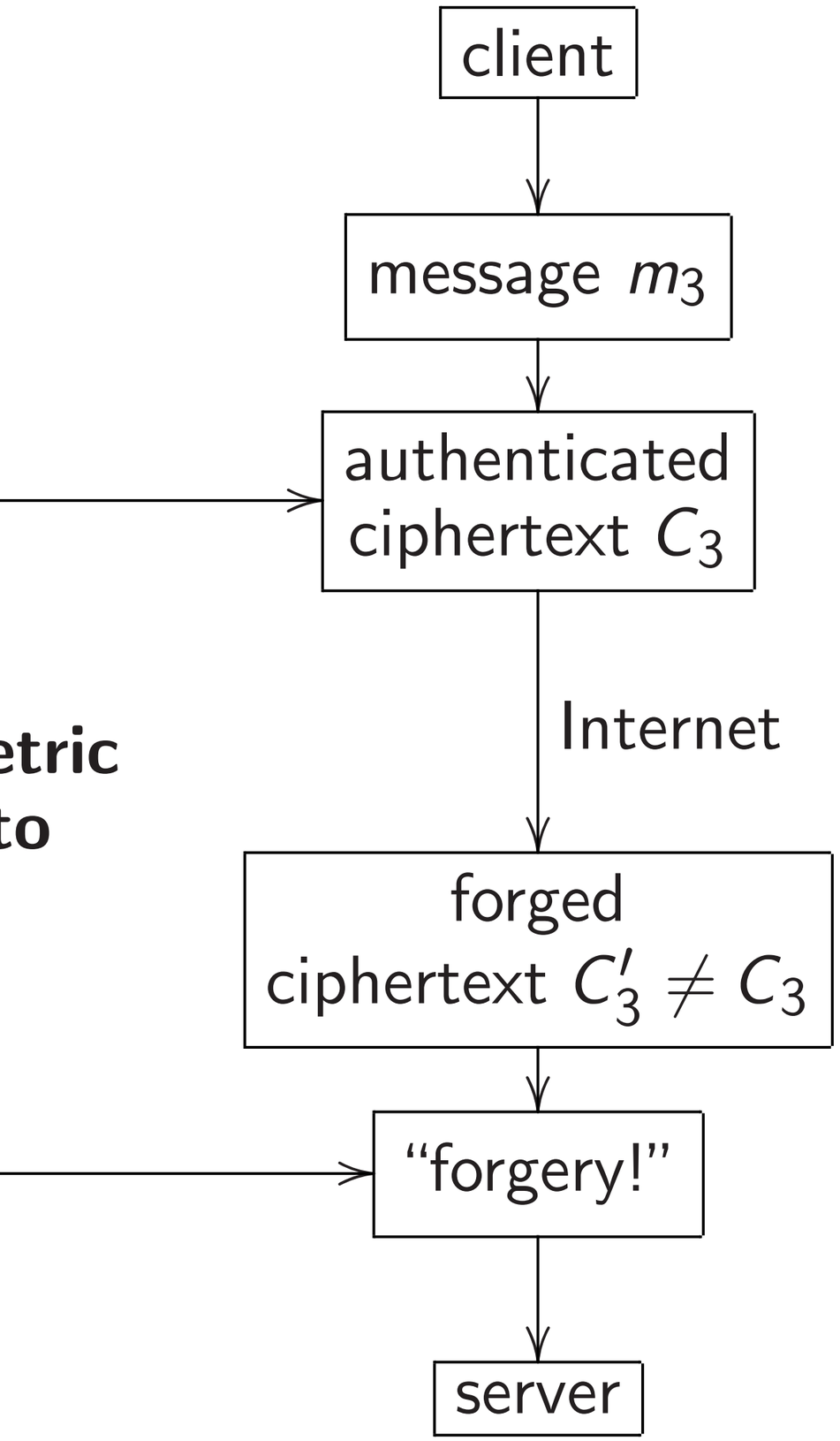
Symmetric crypto

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**.



etric
to

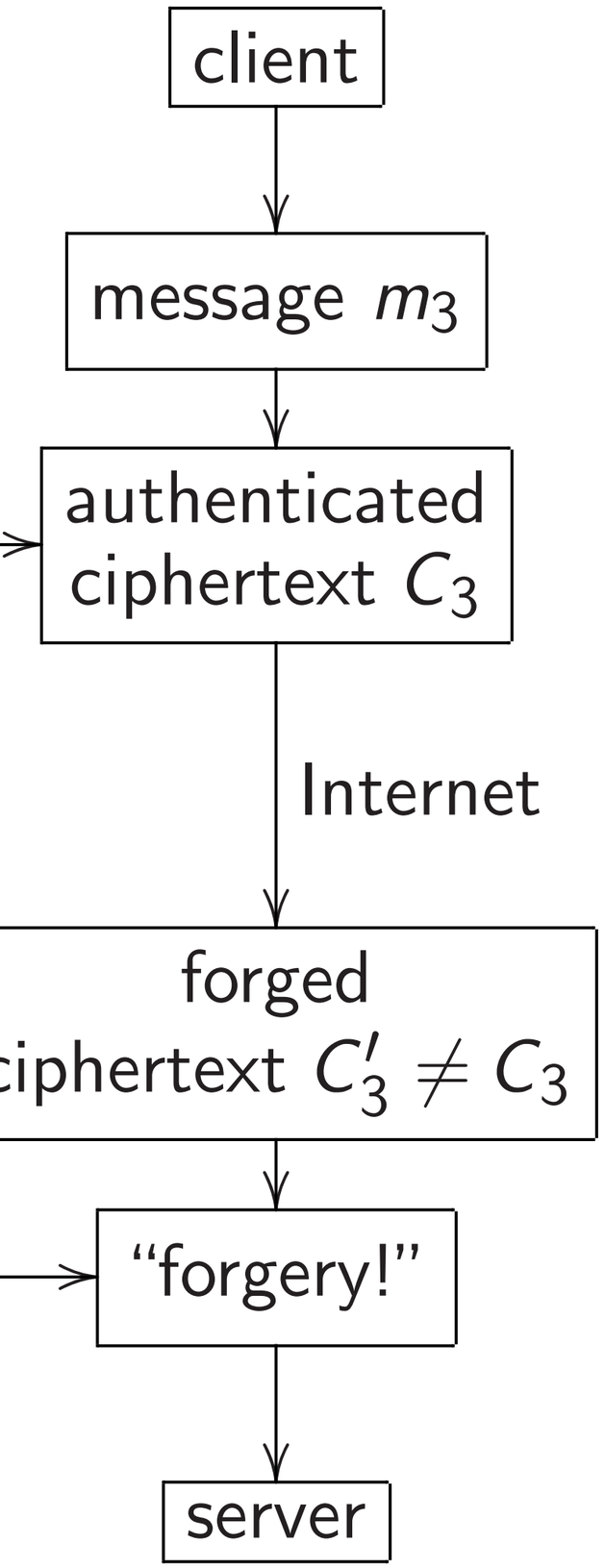
Symmetric crypto: main objectives

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 enc
Assume
Assume
secret 30
t_1 to use
t_2 to use
t_3 to use



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 m
Assume 30-digit m
Assume client, ser
secret 30-digit nur
t_1 to use for messa
t_2 to use for messa
t_3 to use for messa

nt

e m_3

icated
ext C_3

Internet

ed
$C_3' \neq C_3$

ry!"

er

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.

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.

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.

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

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.

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

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

ric crypto: main objectives

y:

r can't forge ciphertexts.

ntiality: Attacker seeing
xts can't figure out
contents. (But can see
number, length, timing.)

ne further objectives.

e: If crypto is too slow,
can flood server's CPU.

ent messages are lost.

mages **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.

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

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

Last tim
compute
using an

Sender a
to messa
Receiver
This pro

main objectives

ge ciphertexts.

Attacker seeing
igure out
(But can see
length, timing.)

objectives.
o is too slow,
server's CPU.
ges are lost.
ilability.

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 eac
compute **authenti**
using another secr

Sender attaches a
to message before
Receiver checks a
This protects integ

iectives

exts.

eeing

see

ning.)

s.

ow,

CPU.

t.

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 authenticat
to message before sending it
Receiver checks authenticato
This protects integrity.

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.

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.

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

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.

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.

cryption mechanism:
30-digit messages.
client, server know
0-digit numbers
e for message 1;
e for message 2;
e for message 3; etc.

$n_1 + t_1) \bmod 10^{30}$;
$n_2 + t_2) \bmod 10^{30}$;
$n_3 + t_3) \bmod 10^{30}$; etc.
tects confidentiality.

M, ChaCha20-Poly1305
s way, scaled up to
arger 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.

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 exp
into $F(k$
simulati
secrets r

echanism:
nessages.
ver know
mbers
age 1;
age 2;
age 3; etc.

od 10^{30};
od 10^{30}; etc.
identiality.

a20-Poly1305
led up to
$\mathbf{Z}/10^{30}$.

4

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.

5

AES expands 256-
into $F(k,1), F(k,$
simulating many i
secrets $r, s_1, t_1, \ldots$

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
into $F(k,1), F(k,2), F(k,3)$
simulating many independen
secrets $r, s_1, t_1, \ldots$.

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 $r, s_1, t_1, \ldots$.

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 $r, s_1, t_1, \ldots$.

ChaCha20 also does this, using a different function F.

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 $r, s_1, t_1, \ldots$.

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.

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 $r, s_1, t_1, \ldots$.

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.

e: For each message
e **authenticator**
other secret number.

attaches authenticator
age before sending it.
checks authenticator.
tects integrity.

use multiplications.
M, ChaCha20-Poly1305
s way, again scaled up.

uld be the whole picture
, server started with
secret random numbers.

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

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 ("p
Attacker
$F(k,1)$,
independ
blocks, g
that retu
Server is

ch message
icator
et number.

uthenticator
sending it.
uthenticator.
grity.

lications.
aa20-Poly1305
ain scaled up.

whole picture
arted with
dom numbers.

5

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.

6

PRF ("pseudoranc
Attacker can't dist
$F(k,1), F(k,2), F$
independent unifor
blocks, given acce
that returns $F(k,$
Server is called an

e

r.

or

t.

or.

1305

up.

cture

bers.

5

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.

6

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

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**.

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**.

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

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**.

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.

pands 256-bit secret k
$k,1), F(k,2), F(k,3), \ldots$
ng many independent
$r, s_1, t_1, \ldots$.

20 also does this,
different function F.

on of **PRG**
prandom generator"):
r can't distinguish
$F(k,2), F(k,3), \ldots$
ing of independent
random blocks.

: "pseudorandom"
y 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**.

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 bl
PRF pro
applicati

-bit secret k
$2), F(k,3), \ldots$
ndependent
.

es this,
unction F.
;
generator"):
tinguish
$(k,3), \ldots$
ependent
locks.

random"
eanings.

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 a
PRF property fails
application securit

k
$), \ldots$
nt
):

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 danger
PRF property fails, and ofte
application security fails.

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.

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.

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

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.

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?

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.

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.

pseudorandom function”):
r can't distinguish
$F(k,2), F(k,3), \ldots$ from
dent uniform random
given access to a server
urns $F(k,i)$ given i.
s called an **oracle**.

... permutation”):
r can't distinguish
$F(k,2), F(k,3), \ldots$
lependent uniform random
blocks, given oracle.

size is big then
PRF $\Rightarrow$ PRG.

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 prov
integrity
ChaCha2
AES and

dom function"):
tinguish
$(k, 3), \ldots$ from
rm random
ss to a server
i) given i.
oracle.

tation"):
tinguish
$(k, 3), \ldots$
uniform random
ven oracle.

then
PRG.

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 confide
integrity of AES-G
ChaCha20-Poly130
AES and ChaCha2

ion"):

from

n

rver

andom

e.

7

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.

8

Can prove confidentiality an
integrity of AES-GCM and
ChaCha20-Poly1305 *assumi*
AES and ChaCha20 are PR

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.

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.

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

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.

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.

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.

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!

ock sizes are dangerous.
pperty fails, and often
ion security fails.

6 Bhargavan–Leurent
2.info: Triple-DES
n TLS. Same attack
aks small block sizes
s Simon, Speck.

ck size: 128 bits.
ack chance $\approx q^2/2^{129}$
s used for q blocks.
afe? How big is q?

20 block size: 512 bits.

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
ChaCha2

are dangerous.
, and often
y fails.

an–Leurent
Triple-DES
ame attack
block sizes
Speck.

28 bits.
e $\approx q^2/2^{129}$
q blocks.
big is q?

ize: 512 bits.

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
ChaCha20 are PR

ous.

n

t

k

s

29

its.

8

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!

9

How do we know that AES ChaCha20 are PRFs? **We d**

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.**

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.**

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

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.**

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.

ve confidentiality and
of AES-GCM and
20-Poly1305 *assuming*
d ChaCha20 are PRFs.

zation: Prove security
) assuming cipher F is a
is a **mode of use** of F.

odes: CTR ("counter
CBC, OFB, many more.

des: ECB, many more.

hat claimed proof
recently broken: OCB2.
check proofs carefully!

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 W
a tiny en

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


    y +=


  }
  b[0] =
}
```

ntiality and
GCM and
05 *assuming*
20 are PRFs.

rove security
g cipher F is a
e of use of F.

R ("counter
B, many more.

many more.

d proof
broken: OCB2.
ofs carefully!

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–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]
}
```

d

ng

Fs.

ty

is a

of F.

er

more.

ore.

CB2.

lly!

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 "T
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;
}
```

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;
}
```

we know that AES and
20 are PRFs? **We don't.**

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

ng slides today:

e example of block cipher.
to be a good cipher,
block size is too small.

ts of this block cipher
ook similar but
e 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:
represen
integer $\mathbb{Z}$

+: addit

c += d:

^: xor; $\oplus$
each bit

Lower pr
so spacin

<<4: mu
$(0, 0, 0, 0$

>>5: div
$(b_5, b_6, \ldots$

that AES and
Fs? **We don't.**

curity
d attack efforts.
cks fail and we
attack ideas."

oday:

of block cipher.
good cipher,
e is too small.

block cipher
r but
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 (
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

<<4: multiplicatio
$(0, 0, 0, 0, b_0, b_1, \ldots$

>>5: division by 3
$(b_5, b_6, \ldots, b_{31}, 0,$

and
lon't.

fforts.
d we
as."

cipher.
r,
nall.

er

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

Function

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) {
0]
];
2]
3];
```


`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.,

$\ldots, b_{31}, 0, 0, 0, 0, 0$).

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.

Wait, ho

```
void en
{
  uint3
  uint3
  for (
    c +=
    x +=
    y +=
  }
  b[0] =
}
```

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

unsigned"

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

2^{32}.

= c + d.

n of

y mod 2.

than + in C,

misleading.

n by 16, i.e.,

$\ldots, b_{27}$).

2, i.e.,

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, 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) $\mapsto$ ciphertext.

ciently **decrypt**:
hertext) $\mapsto$ 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:

```
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

]).

13

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;
}
```

14

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 Fe

```
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) {

]

];

2]

3];
```


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);
```

TEA aga

```
void en
{
  uint3
  uint3
  for (
    c +=
    x +=
      y +=
  }
  b[0] =
}
```

b 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);
```

TEA again for cor

```
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]
}
```

```
ble.

nt32 *k)

1];
e3779b9;
 1) {
2]
3];
0]
];
```


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,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;
}
```

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;
}
```

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);
```


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;
}
```

XORTE

```
void en
{
  uint3
  uint3
  for (
    c +
    x ^
    y ^
  }
  b[0] =
}
```

istel network

cifer" from
ersmith):

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

ing each step:

```
,k);
,k);
,k);
,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 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]
}
```

work

n

tep:

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,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;
}
```

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;
}
```

ain for comparison

```
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;
```


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

mparison

```
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;
```


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

```
nt32 *k)

1];

 1) {

0]

];

2]

3];
```


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

```
nt32 *k)

1];

1) {

]

];

2]

3];
```


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

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

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

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

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

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

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

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

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

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

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

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

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

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 aga

```
void en
{
  uint3
  uint3
  for (
    c +=
    x +=
    y +=
  }
  b[0] =
}
```

er" cipher, since
per than add.

e linear
bits!

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

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

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

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 cor

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

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

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

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

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

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

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;
}
```

a matrix M

efficients in $\mathbf{F}_2$

t, for all (k, b),

$\mathrm{A}_k(b) = (1, k, b)M$.

$\mathrm{A}_k(b_1) \oplus \mathrm{XORTEA}_k(b_2)$

$b_1 \oplus b_2)M$.

t attack:

$b_1 \oplus b_2 \oplus b_3$ then

$\mathrm{A}_k(b_1) \oplus \mathrm{XORTEA}_k(b_2) =$

$\mathrm{A}_k(b_3) \oplus \mathrm{XORTEA}_k(b_4)$.

aks PRP (and PRF):

random permutation

tion) F almost never has

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

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

```
void en
{
  uint3
  uint3
  for (
    c +
    x +
    y +
  }
  b[0] =
}
```

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

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

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

```
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]
}
```

$_k(b_2)$

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

):
n
r has
(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 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;
}
```

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;
}
```

ain for comparison

```
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;
```


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

$1 \oplus k_0 \oplus$

nparison

```
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;
```


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

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

```
nt32 *k)

1];

1) {

0]

];

2]

3];
```


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.)

TEA aga

```
void en
{
  uint3
  uint3
  for (
    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.)

TEA again for cor

```
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) {


0]

];
2]
3];
```

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,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;
}
```

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;
}
```

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.)

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: a

```
void en
{
  uint3
  uint3
  for (
    c +=
    x +=
      y +=
  }
  b[0] =
}
```

-linear,

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

$_4 \oplus k_{96} \oplus b_{32}$.

s

onlinear

ect first bit.

this problem:

near changes

ow bits.

w bits to high

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 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]
}
```

ear.

b_{32}.

t.

m:

ges

nigh

n.)

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,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;
}
```

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;
}
```

ain for comparison

```
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;
```


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

$\mathrm{TEA4}_k($

$\mathrm{TEA4}_k($

nparison

```
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;
```


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

```
nt32 *k)

1];

 1) {

0]

];

2]

3];
```


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

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:

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

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.

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

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 *pr*
probabil
probabil
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:
$\mathrm{TEA4}_k(x + 2^{31}, y)$ 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 sophisticated
trace *probabilities*
probabilities of lin
probabilities of hig
differences $C(x +$
$C(x + \delta) - C(x +$
Use algebra+statis
non-randomness in

```
nt32 *k)

1];

) {

0]

];

2]

3];
```

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 sophisticated attacks:
trace *probabilities* of differen
probabilities of linear equatio
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:
$\mathrm{TEA4}_k(x + 2^{31}, y)$ 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 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:
$\mathrm{TEA4}_k(x + 2^{31}, y)$ 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 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:
$\mathrm{TEA4}_k(x + 2^{31}, y)$ 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 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?

TEA aga

```
void en
{
  uint32
  uint32
  for (i
    c +=
    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?

TEA again for cor

```
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]
}
```

t bit.

on.

:

have

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?

TEA again for comparison

```
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;
}
```

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;
}
```

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?

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

```
void en
{
  uint3
  uint3
  for (
    x +=
      y +=
  }
  b[0] =
}
```

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?

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

```
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]
}
```

nces;
ons;

); etc.
ploit
ties.

s.

gn:

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 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;
}
```

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;
}
```

ain for comparison

```
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;
```


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_k

nparison

```
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;
```


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;
}
```

$\mathrm{REPTEA}_k(b) = I$

where I_k does x+=

```
nt32 *k)

1];

1) {

0]

];

2]

3];
```


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;
}
```


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

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;
}
```


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

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

Try list of 2^{32} inputs b.

Collect outputs $\mathrm{REPTEA}_k(b)$.

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+=...`.

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

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+=...`.

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.)

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 en
{
  uint3
  uint3
  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) {
]
];
2]
3];
```


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

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,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;
}
```

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

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

of 2^{32} inputs b.

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

hance that some b in list

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

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

n (b, a) from list:

ing equations $a = I_k(b)$,

$\mathrm{A}_k(a)$=$I_k(\mathrm{REPTEA}_k(b))$

e out k. (More equations:

ncrypting these outputs.)

n 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

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

where $(k$

$(k[0] + 2$

$_k^{1000}(b)$

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

uts b.

$\mathrm{EPTEA}_k(b)$.

some b in list

) in list. Then

$(\mathrm{REPTEA}_k(b))$.

om list:

ons $a = I_k(b)$,

$(\mathrm{REPTEA}_k(b))$

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.

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

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.,

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

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

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

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.,

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

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

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

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.,
$\mathsf{TEA}_{k'}(b) = \mathsf{TEA}_k(b)$
where $(k'[0], k'[1], k'[2], k'[3]) =$
$(k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])$.

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.,
$\mathsf{TEA}_{k'}(b) = \mathsf{TEA}_k(b)$
where $(k'[0], k'[1], k'[2], k'[3]) =$
$(k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])$.

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.,
$\mathsf{TEA}_{k'}(b) = \mathsf{TEA}_k(b)$
where $(k'[0], k'[1], k'[2], k'[3]) =$
$(k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])$.

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.,
$\mathrm{TEA}_{k'}(b) = \mathrm{TEA}_k(b)$
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

nal 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.,
$\mathsf{TEA}_{k'}(b) = \mathsf{TEA}_k(b)$
where $(k'[0], k'[1], k'[2], k'[3]) =$
$(k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])$.

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–Schn
Fancier relationshi
has chance 2^{-11} o
a particular output

```
nt32 *k)

1];

1) {

0]

];

2]

3];
```

Related keys: e.g.,
$\mathrm{TEA}_{k'}(b) = \mathrm{TEA}_k(b)$
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–Wagn
Fancier relationship betwee
has chance 2^{-11} of producin
a particular output equation

Related keys: e.g.,
$\mathrm{TEA}_{k'}(b) = \mathrm{TEA}_k(b)$
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.,
$\mathrm{TEA}_{k'}(b) = \mathrm{TEA}_k(b)$
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 modes using TEA.

Related keys: e.g.,
$\mathrm{TEA}_{k'}(b) = \mathrm{TEA}_k(b)$
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 modes using TEA.

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

keys: e.g.,

$b) = \mathrm{TEA}_k(b)$

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

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

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 modes using TEA.

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 modes using TEA.

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

])) =
$k[3]$).

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 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 cryptolog
Includes symmetric attacks.

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