## Examples of symmetric primitives

**D. J. Bernstein**

<table>
<thead>
<tr>
<th>Function</th>
<th>message len</th>
</tr>
</thead>
<tbody>
<tr>
<td>Permutation</td>
<td>fixed</td>
</tr>
<tr>
<td>Compression function</td>
<td>fixed</td>
</tr>
<tr>
<td>Block cipher</td>
<td>fixed</td>
</tr>
<tr>
<td>Tweakable block cipher</td>
<td>fixed</td>
</tr>
<tr>
<td>Hash function</td>
<td>variable</td>
</tr>
<tr>
<td>MAC (without nonce)</td>
<td>variable</td>
</tr>
<tr>
<td>MAC (using nonce)</td>
<td>variable</td>
</tr>
<tr>
<td>Stream cipher</td>
<td>variable</td>
</tr>
<tr>
<td>Authenticated cipher</td>
<td>variable</td>
</tr>
<tr>
<td>tweak</td>
<td>key</td>
</tr>
<tr>
<td>-------</td>
<td>-----</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>no</td>
<td>no</td>
</tr>
<tr>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>
1994 Wheeler–Needham "TEA, a tiny encryption algorithm":

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Can efficiently **decrypt**: $(key, ciphertext) \mapsto plaintext$. 
Wait, how can we decrypt?

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

```c
void decrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 32 * 0x9e3779b9;
    for (r = 0; r < 32; r += 1) {
        y -= x+c ^ (x<<4)+k[2]
            ^ (x>>5)+k[3];
        x -= y+c ^ (y<<4)+k[0]
            ^ (y>>5)+k[1];
        c -= 0x9e3779b9;
    }
    b[0] = x; b[1] = y;
}
```
Generalization, **Feistel network** (used in, e.g., “Lucifer” from 1973 Feistel–Coppersmith):

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

Decryption, inverting each step:

\[
\ldots
\]
\[
y \xrightarrow{-} \text{function4}(x,k);
\]
\[
x \xrightarrow{-} \text{function3}(y,k);
\]
\[
y \xrightarrow{-} \text{function2}(x,k);
\]
\[
x \xrightarrow{-} \text{function1}(y,k);
\]
Higher-level functionality

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

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

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

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

TEA-CTR produces ciphertext

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

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

CTR is a mode of operation that converts block cipher TEA into stream cipher TEA-CTR.
User also wants to recognize forged/modified ciphertexts.
User also wants to recognize forged/modified ciphertxts.

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

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

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

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

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

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

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

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

Specify how nonce is chosen: message number. (Stateless alternative: uniform random.)
Is this secure?

Step 1: Define security for authenticated ciphers.
Is this secure?

Step 1: Define security for authenticated ciphers.

This is not easy to do!
Is this secure?

Step 1: Define security for authenticated ciphers.

This is not easy to do!

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

Too weak. Many ciphers leak plaintext or allow forgeries without leaking key.
Is this secure?

Step 1: Define security for authenticated ciphers.

This is not easy to do!

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

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

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

Hard to define clearly.

Everything seems “attackable”.
Step 2: After settling on target security definition, prove that security follows from simpler properties.
Step 2: After settling on target security definition, prove that security follows from simpler properties.

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

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

I.e. Prove that any PRF attack against
\[ n \mapsto \text{TEA}_k(n, 0), \text{TEA}_k(n, 1), \ldots \]
implies PRF attack against
\[ b \mapsto \text{TEA}_k(b). \]
privacy of TEA-CTR-XCBC-MAC

privacy of TEA-CTR

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

PRF security of TEA

PRP security of TEA
authenticity of TEA-CTR-XCBC-MAC

authenticity of TEA-XCBC-MAC

PRF security of TEA-XCBC-MAC

PRF security of TEA

PRP security of TEA

PRP security of TEA

PRP security of TEA
Many things can go wrong here:

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

1. Security definition too weak.

2. Internal mismatch between hypotheses and conclusions.
Many things can go wrong here:

1. Security definition too weak.

2. Internal mismatch between hypotheses and conclusions.

3. Errors in proofs.
Did anyone write full proofs?
Did anyone check all details?
Many things can go wrong here:

1. Security definition too weak.

2. Internal mismatch between hypotheses and conclusions.

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

4. Quantitative problems. e.g. 2016 Bhargavan–Leurent

sweet32.info: Triple-DES broken in TLS; PRP-PRF switch too weak for 64-bit block ciphers.
Many things can go wrong here:

1. Security definition too weak.

2. Internal mismatch between hypotheses and conclusions.

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

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

5. Is TEA PRP-secure?
One-time pad has complete proof of privacy, but key must be as long as total of all messages.
One-time pad has complete proof of privacy, but key must be as long as total of all messages.

Wegman–Carter authenticator has complete proof of authenticity, but key length is proportional to number of messages.
One-time pad has complete proof of privacy, but key must be as long as total of all messages.

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

Short-key cipher handling many messages: no complete proofs.
One-time pad has complete proof of privacy, but key must be as long as total of all messages.

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

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

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

void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 32; r += 1) {
        c += 0x9e3779b9;
        x ^= y^c ^ (y<<4)^k[0]
            ^ (y>>5)^k[1];
        y ^= x^c ^ (x<<4)^k[2]
            ^ (x>>5)^k[3];
    }
    b[0] = x; b[1] = y;
}
“Hardware-friendlier” cipher, since xor circuit is cheaper than add.
“Hardware-friendlier” cipher, since xor circuit is cheaper than add. But output bits are linear functions of input bits!
“Hardware-friendlier” cipher, since xor circuit is cheaper than add.

But output bits are linear functions of input bits!

e.g. First output bit is

$$1 \oplus k_0 \oplus k_1 \oplus k_3 \oplus k_{10} \oplus k_{11} \oplus k_{12} \oplus k_{20} \oplus k_{21} \oplus k_{30} \oplus k_{32} \oplus k_{33} \oplus k_{35} \oplus k_{42} \oplus k_{43} \oplus k_{44} \oplus k_{52} \oplus k_{53} \oplus k_{62} \oplus k_{64} \oplus k_{67} \oplus k_{69} \oplus k_{76} \oplus k_{85} \oplus k_{94} \oplus k_{96} \oplus k_{99} \oplus k_{101} \oplus k_{108} \oplus k_{117} \oplus k_{126} \oplus b_1 \oplus b_3 \oplus b_{10} \oplus b_{12} \oplus b_{21} \oplus b_{30} \oplus b_{32} \oplus b_{33} \oplus b_{35} \oplus b_{37} \oplus b_{39} \oplus b_{42} \oplus b_{43} \oplus b_{44} \oplus b_{47} \oplus b_{52} \oplus b_{53} \oplus b_{57} \oplus b_{62}.$$
There is a matrix $M$ with coefficients in $\mathbb{F}_2$ such that, for all $(k, b)$, $\text{XORTEA}_k(b) = (1, k, b)M$. 
There is a matrix $M$ with coefficients in $\mathbb{F}_2$ such that, for all $(k, b)$,

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

$$\text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = (0, 0, b_1 \oplus b_2)M.$$
There is a matrix $M$ with coefficients in $\mathbb{F}_2$ such that, for all $(k, b)$,
$\text{XORTEA}_k(b) = (1, k, b)M$.

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

Very fast attack:
if $b_4 = b_1 \oplus b_2 \oplus b_3$ then
$\text{XORTEA}_k(b_1) \oplus \text{XORTEA}_k(b_2) = \text{XORTEA}_k(b_3) \oplus \text{XORTEA}_k(b_4)$. 
There is a matrix $M$ with coefficients in $\mathbb{F}_2$ such that, for all $(k, b)$, $\text{XORTEA}_k(b) = (1, k, b)M$.

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

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

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


void encrypt(uint32 *b,uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0;r < 32;r += 1) {
        c += 0x9e3779b9;
        x += y+c ^ (y<<4)+k[0]
            ^ (y<<5)+k[1];
        y += x+c ^ (x<<4)+k[2]
            ^ (x<<5)+k[3];
    }
    b[0] = x; b[1] = y;
}
Addition is not $\mathbb{F}_2$-linear, but addition mod 2 is $\mathbb{F}_2$-linear.

First output bit is

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

First output bit is

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

Higher output bits are increasingly nonlinear but they never affect first bit.
Addition is not $\mathbb{F}_2$-linear, but addition mod 2 is $\mathbb{F}_2$-linear.

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

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

How TEA avoids this problem: $\gg 5$ **diffuses** nonlinear changes from high bits to low bits.
Addition is not $\mathbb{F}_2$-linear, but addition mod 2 is $\mathbb{F}_2$-linear.

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

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

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

(Diffusion from low bits to high bits: $\ll 4$; carries in addition.)
void encrypt(uint32 *b, uint32 *k)
{
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0;
    for (r = 0; r < 4; r += 1) {
        c += 0x9e3779b9;
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
Fast attack:
\[ \text{TEA}_k(x + 2^{31}, y) \text{ and } \text{TEA}_k(x, y) \text{ have same first bit.} \]
Fast attack:
$\text{TEA4}_k(x + 2^{31}, y)$ and $\text{TEA4}_k(x, y)$ have same first bit.

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

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

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

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

$r = 3$: multiples of $2^1, 2^0$. 
Fast attack:
TEA4\(_k\)(x + 2^{31}, y) and
TEA4\(_k\)(x, y) have same first bit.

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

Uniform random function \(F\):
\(F(x + 2^{31}, y)\) and \(F(x, y)\) have same first bit with probability 1/2.
Fast attack:
$TEA4_k(x + 2^{31}, y)$ and $TEA4_k(x, y)$ have same first bit.

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

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

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

PRF advantage $1/2$.

Two pairs $(x, y)$: advantage $3/4$. 
More sophisticated attacks: trace *probabilities* of differences; probabilities of linear equations; probabilities of higher-order differences
\[ C(x + \delta + \epsilon) - C(x + \delta) - C(x + \epsilon) + C(x); \] etc. Use algebra+statistics to exploit non-randomness in probabilities.
More sophisticated attacks: trace *probabilities* of differences; probabilities of linear equations; probabilities of higher-order differences $C(x + \delta + \epsilon) - C(x + \delta) - C(x + \epsilon) + C(x)$; etc. Use algebra+statistics to exploit non-randomness in probabilities.

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

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

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

Hard question in cipher design: How many “rounds” are really needed for security?
void encrypt(uint32 *b, uint32 *k) {
    uint32 x = b[0], y = b[1];
    uint32 r, c = 0x9e3779b9;
    for (r = 0; r < 1000; r += 1) {
        x += y + c ^ (y << 4) + k[0]
            ^ (y >> 5) + k[1];
        y += x + c ^ (x << 4) + k[2]
            ^ (x >> 5) + k[3];
    }
    b[0] = x; b[1] = y;
}
REPTEA$_k(b) = I_k^{1000}(b)$
where $I_k$ does $x+=\ldots; y+=\ldots$. 
$\text{REPTEA}_k(b) = I_k^{1000}(b)$
where $I_k$ does $x+=\ldots;y+=\ldots$.

Try list of $2^{32}$ inputs $b$.
Collect outputs $\text{REPTEA}_k(b)$. 
REPTEA_k(b) = I_k^{1000}(b)
where I_k does x+=...; y+=... .

Try list of $2^{32}$ inputs b.
Collect outputs REPTEA_k(b).
Good chance that some b in list also has a = I_k(b) in list. Then
REPTEA_k(a) = I_k(REPTEA_k(b)).
\( \text{REPTEA}_k(b) = I_k^{1000}(b) \)

where \( I_k \) does \( x+=\ldots;y+=\ldots \).

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

For each \( (b, a) \) from list:
Try solving equations \( a = I_k(b) \),
\( \text{REPTEA}_k(a) = I_k(\text{REPTEA}_k(b)) \)
to figure out \( k \). (More equations: try re-encrypting these outputs.)
\[ \text{REPTEA}_k(b) = I^1_{1000}(b) \]

where \( I_k \) does \( x+ = \ldots; y+ = \ldots \).

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

Good chance that some \( b \) in list also has \( a = I_k(b) \) in list. Then
\[ \text{REPTEA}_k(a) = I_k(\text{REPTEA}_k(b)) \]

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

This is a **slide attack**.

TEA avoids this by varying \( c \).
What about original TEA?

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

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

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

Is this an attack?
Related keys: e.g.,

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

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

Is this an attack?

PRP attack goal: distinguish \(\text{TEA}_k\), for one secret key \(k\), from uniform random permutation.
Related keys: e.g.,
\[ \text{TEA}_{k'}(b) = \text{TEA}_k(b) \]
where \((k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])\).

Is this an attack?

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

Brute-force attack:
Guess key \(g\), see if \(\text{TEA}_g\) matches \(\text{TEA}_k\) on some outputs.
Related keys: e.g.,
$$\text{TEA}_{k'}(b) = \text{TEA}_k(b)$$
where $$(k'[0], k'[1], k'[2], k'[3]) = (k[0] + 2^{31}, k[1] + 2^{31}, k[2], k[3])$$.

Is this an attack?

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

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

Related keys $$\Rightarrow g$$ succeeds with chance $$2^{-126}$$. Still very small.
1997 Kelsey–Schneier–Wagner: Fancier relationship between $k, k'$ has chance $2^{-11}$ of producing a particular output equation.
1997 Kelsey–Schneier–Wagner: Fancier relationship between $k, k'$ has chance $2^{-11}$ of producing a particular output equation.

No evidence in literature that this helps brute-force attack, or otherwise affects PRP security. No challenge to security analysis of TEA-CTR-XCBC-MAC.
1997 Kelsey–Schneier–Wagner: Fancier relationship between $k, k'$ has chance $2^{-11}$ of producing a particular output equation.

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

But advertised as “related-key cryptanalysis” and claimed to justify recommendations for designers regarding key scheduling.
Some ways to learn more about cipher attacks, hash-function attacks, etc.:

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

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

Some cipher history


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


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

1976: NSA meets Diffie and Hellman to discuss criticism. Claims “somewhere over $400,000,000” to break a DES key; “I don’t think you can tell any Congressman what’s going to be secure 25 years from now.”
1977: DES is standardized.

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

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

1978: Congressional investigation into NSA influence concludes “NSA convinced IBM that a reduced key size was sufficient”. 
1977: DES is standardized.

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

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

1977: DES is standardized.

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

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


Researchers publish new cipher proposals and security analysis.


1999: NIST selects five AES finalists: MARS, RC6, Rijndael, Serpent, Twofish.
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”
2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”

2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


2000: NIST, advised by NSA, selects Rijndael as AES.

“Security was the most important factor in the evaluation”—Really?

“Rijndael appears to offer an adequate security margin. . . . Serpent appears to offer a high security margin.”


2013–now: CAESAR competition.
Main operations in AES:
add round key to block;
apply substitution box
\( x \mapsto x^{254} \) in \( \mathbb{F}_{256} \)
to each byte in block;
linearly mix bits across block.
Main operations in AES:
add round key to block;
apply \textbf{substitution box} \[ x \mapsto x^{254} \text{ in } \mathbb{F}_{256} \]
to each byte in block;
linearly mix bits across block.

Extensive security analysis.
No serious threats to AES-256 multi-target SPRP security
(which implies PRP security),
even in a post-quantum world.
Main operations in AES:
add round key to block;
apply substitution box
\( x \mapsto x^{254} \) in \( F_{256} \)
to each byte in block;
linearly mix bits across block.

Extensive security analysis.
No serious threats to AES-256 multi-target SPRP security
(which implies PRP security),
even in a post-quantum world.

So why isn’t AES-256 the end of the symmetric-crypto story?
The latest news and insights from Google on security and safety on the Internet

Speeding up and strengthening HTTPS connections for Chrome on Android
April 24, 2014

Posted by Elie Bursztein, Anti-Abuse Research Lead

Earlier this year, we deployed a new TLS cipher suite in Chrome that operates three times faster than AES-GCM on devices that don’t have AES hardware acceleration, including most Android phones, wearable devices such as Google Glass and older computers. This improves user experience, reducing latency and saving battery life by cutting down the amount of time spent encrypting and decrypting data.

To make this happen, Adam Langley, Wan-Teh Chang, Ben Laurie and I began implementing new algorithms -- ChaCha 20 for symmetric encryption and Poly1305
Hi all,

(Please note that this patchset is a tryout. It is not finalized and it to be merged quite yet!)

It was officially decided to *not* allow encryption [1]. We've been working to enable storage encryption to entry-level Android "Android Go" devices sold in developing countries, but these devices still ship with no encryption. We have to use older CPUs like ARM Cortex-A7 that don't have Cryptography Extensions, making AES-XTS too slow. As we explained in detail earlier, e.g. in this [2], it is a challenging problem due to the lack of CPUs that are suitable for practical use in dm-crypt. Speck, in this day and age the choice of encryption engine for Linux's dm-crypt, has a large political element, restricting its use on entry-level devices.

Therefore, we (well, Paul Crowley did the work for us) have implemented a new encryption mode, HPolyC. In essence, it uses the ChaCha stream cipher for disk encryption. Read more in this [3] technical paper here: https://eprint.iacr.org/2017/1252.pdf.
1-1-ebiggers () kernel ! org

...m>

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

...how Android devices to use Speck
...find an alternative way to bring
...Android devices like the inexpensive
...countries. Unfortunately, often
...ption, since for cost reasons they
...A7; and these CPUs lack the ARMv8
...S much too slow.

... in [2], this is a very
...encryption algorithms that meet
...ts, while still being secure and
...and fscrypt. And as we saw with
...of cryptographic primitives also
...ing the options even further.

...the real work) designed a new
...HPolyC makes it secure to use the
...tion. HPolyC is specified by our
...18/720.pdf ("HPolyC:
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.
AES performance seems limited in both hardware and software by small 128-bit block size, heavy S-box design strategy.

AES software ecosystem is complicated and dangerous. Fast software implementations of AES S-box often leak secrets through timing.

Picture is worse for high-security authenticated ciphers. 128-bit block size limits PRF security. Workarounds are hard to audit.
ChaCha creates safe systems with much less work than AES.
ChaCha creates safe systems with much less work than AES.

More examples of how symmetric primitives have been improving speed, simplicity, security:

PRESENT is better than DES.

Skinny is better than Simon and Speck.

Keccak, BLAKE2, Ascon are better than MD5, SHA-0, SHA-1, SHA-256, SHA-512.

Gimli permutes \( \{0, 1\}^{384} \).

Gimli permutes $\{0, 1\}^{384}$.

“Wait, where’s the key?”

Gimli permutes $\{0, 1\}^{384}$.

“Wait, where’s the key?”

Even–Mansour SPRP mode:

$E_k(m) = k \oplus \text{Gimli}(k \oplus m)$.

Salsa/ChaCha PRF mode:

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

Or: $(k, 0) \oplus \text{Gimli}(k, m)$. 
void gimli(uint32 *b)
{
    int r, c;
    uint32 x, y, z;

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

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

if ((r & 3) == 0)
    b[0] ^= (0x9e377900 | r);
}
No additions. Nonlinear carries are replaced by shifts of & , |.
(Idea stolen from NORX cipher.)

Big rotations diffuse changes quickly across bit positions.

x, y, z interaction diffuses changes quickly through columns (0, 4, 8; 1, 5, 9; 2, 6, 10; 3, 7, 11).

Other swaps diffuse changes through rows. Deliberately limited swaps per round \( \Rightarrow \) faster rounds on a wide range of platforms.