

Small high-security  
encryption, authentication,  
and hashing

D. J. Bernstein

University of Illinois at Chicago

# Main goals of cryptography

Alice and Bob are communicating.  
Eve is eavesdropping.

Alice and Bob have several  
standard security goals:

**Confidentiality** despite espionage.  
Maybe Eve wants to acquire data.

**Integrity** despite corruption.  
Maybe Eve wants to change data.

**Availability** despite sabotage.  
Maybe Eve wants to destroy data.

The cold-boot example:

Alice is your laptop's CPU.

Bob is also your laptop's CPU.

Alice sends messages to Bob  
by copying data to DRAM,  
copying data to disk, etc.

Eve steals your laptop,  
*freezes the DRAM,*  
installs the DRAM and disk  
inside Eve's computer,  
reads the entire DRAM,  
and reads the entire disk.

Eve now has all your laptop data,  
violating confidentiality.

[secure-medicine.org](https://secure-medicine.org) example:

Alice is an M7278 pacemaker sitting next to a patient's heart. Bob is a pacemaker control unit communicating by 175 kHz radio.

By imitating Bob, Eve can

- verify Alice's presence,
- see patient's name,
- see heart-stability history,

[secure-medicine.org](https://secure-medicine.org) example:

Alice is an M7278 pacemaker sitting next to a patient's heart. Bob is a pacemaker control unit communicating by 175 kHz radio.

By imitating Bob, Eve can

- verify Alice's presence,
- see patient's name,
- see heart-stability history,
- disable Alice's defibrillation, or
- tell Alice to give a 1.0J 138V shock to the patient's heart.

The cryptographic solution:

Alice scrambles messages to Bob.

Bob unscrambles them.

Same for Bob→Alice messages.

Eve isn't told the secret  
scrambling/unscrambling methods  
used by Alice and Bob.

Alice and Bob hope that  
Eve cannot understand what  
the scrambled messages mean,  
and cannot forge a message  
that survives the unscrambling.

## Publicly designed ciphers

Typical scrambling method is a public cipher using a shared secret key.

Example: Alice and Bob use the “AES” cipher with a secret 128-bit key.

AES is a public standard.

The only secret is the key.

Alice+Bob choose a random key.

How do Alice and Bob share this secret key? See the next talk.

Cryptographic research literature includes hundreds of papers analyzing various cipher proposals.

These papers have shown that many proposals are unsafe:

Eve can unscramble messages or forge scrambled messages.

Designers have to watch out for “differential attacks” and “slide attacks” and more.

But the research community has built confidence in the security of some ciphers.



Why do we publish ciphers?

Why not use secret ciphers?

Several reasons:

1. Protecting a short secret key is easier than protecting a secret scrambling algorithm.

2. *Sharing* a short secret key is easier than sharing a secret scrambling algorithm.

3. If there are many users then one of them is likely to leak the cipher to Eve.

4. Cipher publication for review helps us eliminate bad ciphers.

1997: United States NIST called for proposals of an Advanced Encryption Standard.

1998: 15 proposals were submitted from around the world. Several of them were broken.

1999: NIST selected 5 finalists.

2000: NIST selected "Rijndael" as AES because it was fast.

2004: eSTREAM project called for stream ciphers providing higher throughput *or* smaller size than AES.

2005: 34 proposals were submitted from around the world. Some of them were broken.

2006: Second round, 28 ciphers.

2007: Third round, 16 ciphers.

2008: eSTREAM selected a final portfolio of 4 “software” ciphers and 4 low-security “hardware” ciphers.

1 “hardware” cipher was broken.

2007: United States NIST called for SHA-3 proposals.

2008: 64 proposals were submitted from around the world. Many of them were broken.

2009: NIST selected 14 proposals for the second round.

2010: NIST is expected to select 5 finalists.

2012?: NIST will select SHA-3.

## And now the bad news

It is extremely unusual for a cryptographic designer to be an expert in attack methods *and* software efficiency *and* hardware efficiency.

Some cipher proposals are from efficiency experts, but most of these proposals are quickly broken.

Example: Intel's Vortex.

Most surviving proposals are from attack experts.

Most of these proposals are quite inefficient in software and even worse in hardware.

Example: Rivest's MD6.

Huge number of gates,  
huge power consumption,  
huge latency, etc.

Occasionally a design team has expertise in attacks and in software efficiency, but hardware efficiency is still bad.

Example: HC-128.

Occasionally a design team has expertise in attacks and in *hardware* efficiency, but something always goes wrong.

The #2 AES candidate, Serpent, was very fast in hardware, but slow software performance prevented standardization.

New generation of “lightweight” ciphers (mCrypton, DESL, PRESENT, KATAN, KTANTAN) are even smaller in hardware and sometimes fast in software, but have a low security level.

## How to authenticate data

Alice hashes a message  $m$ ,  
obtaining an  $h$ -bit hash  $H(m)$ .

Alice encrypts the hash,  
obtaining  $a = E(H(m))$ .

Alice sends  $m, a$ .

Maybe Eve changes  $m, a$   
to a forgery  $m', a'$ .

Bob discards  $m', a'$   
if  $a' \neq E(H(m'))$ .

Random-guessing attack:

Eve changes  $m$  to random  $m'$ ;  
chance  $1/2^h$  of  $H(m) = H(m')$ .

Are there more effective attacks?



Replay attack:

Eve records  $m, a$  from Alice and later sends it again.

Always works:  $a = E(H(m))$ , so Bob accepts the replay.

Eve is violating integrity of the *sequence* of messages received by Bob.

Fix: Alice sends  $n, m, E(H(n, m))$

where  $n$  is a counter.

Bob remembers largest  $n$ , rejects any  $n' \leq n$ .

What if Alice doesn't have storage for a counter?

One solution:

Bob stores the counter  $n$ ,  
sends  $n, E(n)$  to Alice.

Alice checks  $n, E(n)$ ,  
sends  $n, m, E(H(n, m))$  to Bob.

Another solution:

Bob sends a random  $n$ .

Need many bits in  $n$

to avoid accidental collisions

and to avoid malicious collisions.

How does  $H$  work?

One traditional approach:

$H$  is SHA-1 or SHA-256 or another public hash function.

Another traditional approach:

$H$  is, e.g., AES-128-CBC using a secret key  $k$ .

$$H(n, m_1, m_2) = \text{AES}_k(\text{AES}_k(n) \oplus m_1) \oplus m_2.$$

Much better approach: There are smaller, faster functions  $H$  with strong security guarantees.

The basic idea:

$$H_r(m_1, m_2, m_3) = \\ (((r \oplus m_1)r \oplus m_2)r \oplus m_3)r$$

where  $r$  is a secret 128-bit key  
and  $(\cdot \cdot \cdot)r$  is multiplication  
by  $r$  in the field  $GF(2^{128})$ .

For any cipher  $E$ ,

the authenticator  $a = E_k(n) \oplus$   
 $((r \oplus m_1)r \oplus m_2)r \oplus m_3)r$   
is almost as secure as  $E$ .

Security gap is only  $N/2^{128}$   
if Alice receives  $N$  blocks.

Multiplier structure allows a tremendous variety of tradeoffs.

Can make a tiny multiplier, or a big high-throughput pipelined multiplier, or something in between.

Can use subfields etc.; many fancy optimizations.

Can do even better:

Recent research has found safe authenticators using 1 multiplication in  $GF(2^{128})$  for every 2 blocks of  $m$ .

## How to encrypt data

Need a strong cipher  $E$ .

Input to  $E$ : secret key  $k$ ,  
message number  $n$ .

Output: 128 bits  $E_k(n)$  used  
for authenticator  $E_k(n) \oplus H_r(m)$ .

Can design  $E$  to produce  
longer stream of output  
for dynamic generation of  $r$   
or for authenticated encryption:

$E_k(n) \oplus (H_r(m), m)$ .

e.g. 512-bit output of  $E$  for  
128-bit  $H$ , 128-bit  $r$ , 256-bit  $m$ .

How long should  $k$  be?

Typical assessment:

“The recent RSA-768 attack  
spent  $\approx 2^{67}$  CPU cycles.

80-bit  $k$  is big enough.”

How long should  $k$  be?

Typical assessment:

“The recent RSA-768 attack spent  $\approx 2^{67}$  CPU cycles.

80-bit  $k$  is big enough.”

Two big reasons to disagree:

1. Serious attackers have many more computers—and sometimes build ASICs!  $2^{80}$  is feasible today.

2. Attacking  $2^{30}$  targets  $E_k(n)$  is  $2^{30}$  times more efficient than attacking one target.



Typical  $E$  construction:

Pad  $n$  with constants,  
obtaining a  $b$ -bit block  
 $x[0], \dots, x[b - 1]$ .

Xor  $k$  into block.

Apply many bit operations:

$x[3] \hat{=} \sim(x[11] \& x[28])$

etc.

Xor  $k$  into block.

Apply many bit operations.

Xor  $k$  into block.

Output resulting block.

e.g. My Salsa20 stream cipher  
expands  $n$  to a 512-bit block;  
adds 256-bit  $k$  into block;  
applies many word operations;  
adds the key again.

To generate longer outputs:  
first block uses  $(n, 0)$ ;  
second block uses  $(n, 1)$ ;  
third block uses  $(n, 2)$ ; etc.

Included in eSTREAM's final  
“software” portfolio.

Some things Salsa20 does well:

Operations are very simple.

Most ciphers have bigger operations such as AES S-box.

Structure is parallel+SIMD, allowing wide range of tradeoffs.

Most ciphers are more “random”: can't do small implementations.

Very few uses of the key  $k$ .

Can even burn  $k$  into chip.

Most ciphers need extra flip-flops for “expanded key” alongside flip-flops for block.

Recent RFID implementation:

180nm process,  $255\mu\text{m} \times 530\mu\text{m}$ ,  
3468 cells, 100kHz, 202 cycles,  
1.8V,  $2.82 \mu\text{W}$  (simulated).

Some directions for improvement  
from cipher designer's perspective:

Can eliminate word additions  
in favor of ANDs or NANDs.

Can improve diffusion,  
achieving security in fewer rounds.

Can replace counters by LFSRs.

Can reduce block size somewhat  
without compromising security.

Typical DRAM controller  
reads/writes 64-byte lines.

Why not encrypt with Salsa20?

DRAM controller generates  
random key on start-up,  
uses physical address as  $n$ .

Could even use ECC DRAM  
for free authentication.