Cryptography in D

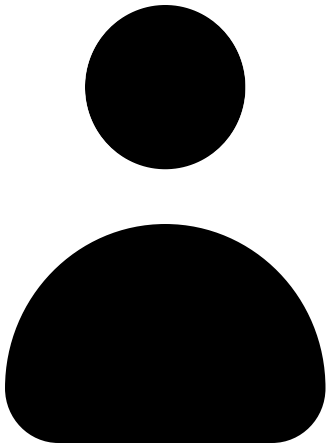
Amaury SÉCHET
@deadalnix

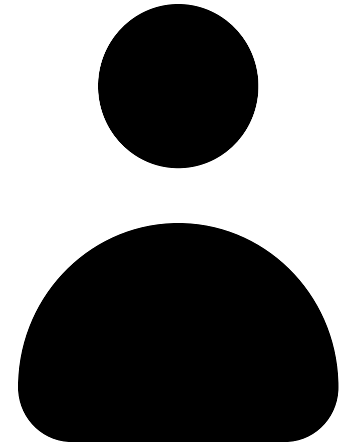
Wikipedia

Cryptography or **cryptology** is the practice and study of techniques for secure communication in the presence of third parties called adversaries.

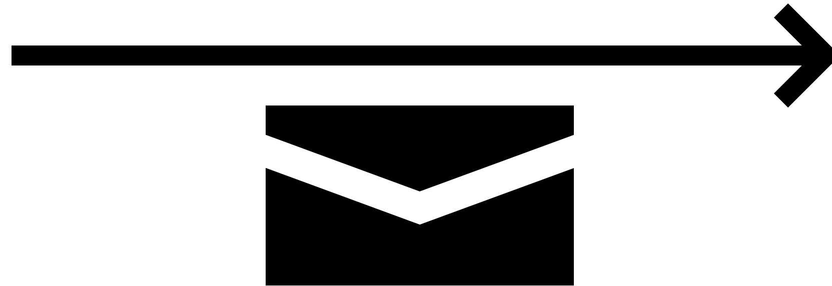
Myself

Cryptography is a set of techniques ensuring the **confidentiality**, **authenticity** and **integrity** of messages.


Alice


Bob

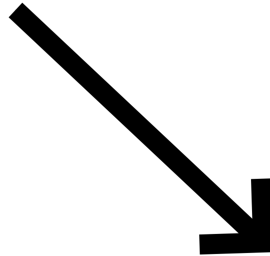
Andrei

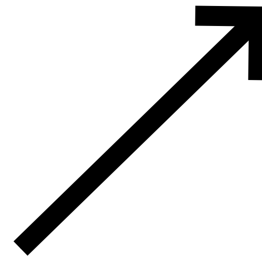


Walter

Confidentiality

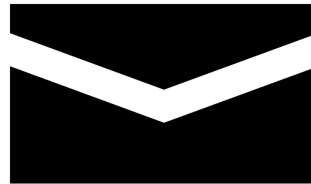
Andrei





Walter

Authenticity

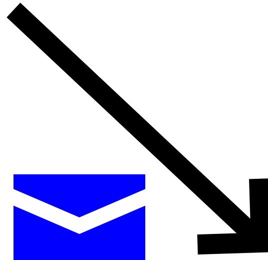


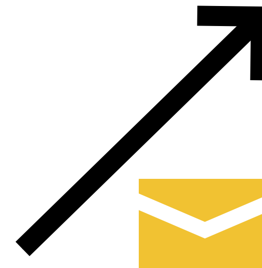
Walter

Hey Walter,
Andrei here...

Integrity

Andrei





Walter

I hope Walter will believe it...

Cryptography

Symetric

Participants share a secret.

Use the shared secret to ensure **confidentiality**, **authenticity** and **integrity**.

Asymetric

Participants have a secret private key and a shared public key.

Use private key to generate proofs which can be verified using the public key.

Information wants to be FREE

John phoned a clinic doing paternity test and then a divorce lawyer.

We don't know what was said...

Use a secret usually known as "key".

The secret must be:

- Small.
- Maximally entropic.
- Ensure the secret is reusable.
- Protected against side channel leaks.

Anything secret is part of your key

Any secret algorithm is part of the key.

Use **public** algorithms.

Using a secret algorithm is known as **security by obscurity**.

"Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break."

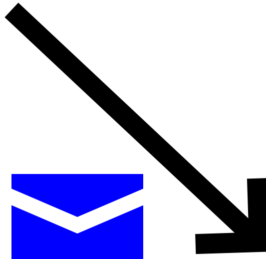
Bruce Schneier

Use **peer reviewed** algorithms.

Integrity

Integrity

Andrei



Walter

I hope Walter will believe it...

Hash functions

Definition

A **hash function** is any function that can be used to map data of arbitrary size to data of fixed size.

In crypto ?

A **cryptographic hash function** is a special class of hash function that has certain properties which make it suitable for use in cryptography.

Thanks wikipedia....

Confusion

It is not possible to get information about the secret key from the output.

It isn't possible to deduce anything related to the input from the output.

Diffusion

Changing the input even slightly drastically changes the output.

One bit flip in the input will flip half the bit of the output on average.

Collision

It must be hard to find 2 inputs sharing the same hash.

Because of the **anniversary paradox** it takes $\mathbf{2^{n/2}}$ trials on average to find a **collision**.

Pre-image

It must be hard, given a hash, to find an input that produces this hash.

It takes $\mathbf{2^{n-1}}$ trials on average to find a **pre-image**.

Exciting

- 80 bits security
- MD5
- SHA1

Boring

- 128 bits security
- SHA256
- SHA3

D hash function API

```
struct Hasher {
    // Initialize the hasher
    void start();

    // Hash some data
    void put(scope const(ubyte)[] data);

    // Get the hash
    ubyte[N] finish();
}
```

Hash function: SHA256

- Internal state of 32B.
- Process message by blocks of 64B.
- Each block is mixed with internal state using **ARX** operations.
- Last round use a special padding to make sure the last block is 64B.

```
struct SHA256 {
    uint[8] state = [
        0x6a09e667, 0xbb67ae85,
        0x3c6ef372, 0xa54ff53a,
        0x510e527f, 0x9b05688c,
        0x1f83d9ab, 0x5be0cd19,
    ];

    ubyte[64] buffer;
    ulong byteCount;

    void start() {
        this = typeof(this).init;
    }

    void put(...) { ... }
    ubyte[32] finish() { ... }
}
```

Hash function: SHA256

```
struct SHA256 {
    uint[8] state;
    ubyte[64] buffer;
    ulong byteCount;

    void start() { ... }
    ubyte[32] finish() { ... }

    void put(const(ubyte)[] input) {
        auto byteIndex = byteCount % 64;

        // If we can fill the buffer, do one round.
        if (byteIndex && ((byteIndex + input.length) >= 64)) {
            // [...]
        }

        // Now we don't need to bufferise.
        while (input.length >= 64) {
                assert(byteIndex == 0, "unexpected buffer position");
                transform(*(cast(ubyte[64]*) input.ptr));
                input = input[64 .. $];
                byteCount += 64;
        }

        // Put the remaining bytes in the buffer.
        if (input.length > 0) {
                memcpy(buffer.ptr + byteIndex, input.ptr, input.length);
                byteCount += input.length;
        }
    }

private:
    void transform(ref ubyte[64] chunk) { ... }
}
```

Hash function: SHA256

```
struct SHA256 {
    uint[8] state;
    ubyte[64] buffer;
    ulong byteCount;

    void start() { ... }
    void put(const(ubyte)[] input) { ... }

    ubyte[32] finish() {
        auto count = byteCount;

        // We want to pad up to 56 bytes mod 64.
        auto paddingSize = 64 - ((byteCount + 8) % 64);
        put(Padding[0 .. paddingSize]);

        // SHA-256 append the size in bits to the last round.
        buffer.byUlong[7] = bswap(count * 8);
        transform(buffer);

        uint[8] ret;
        foreach (i; 0 .. 8) {
                ret[i] = bswap(state[i]);
        }

        // Same player play again.
        start();
        return *(cast(ubyte[32]*) &ret);
    }

private:
    void transform(ref ubyte[64] chunk) { ... }
}
```

Hash function: SHA256

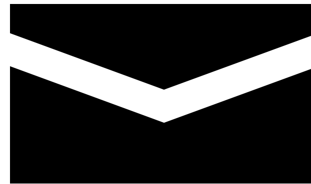
```
struct SHA256 {
    uint[8] state;
    ubyte[64] buffer;
    ulong byteCount;

    void start() { ... }
    void put(const(ubyte)[] input) { ... }
    ubyte[32] finish() { ... }

private:
    void transform(ref ubyte[64] chunk) {
        auto s = state;
        uint[16] w;

        foreach (i; 0 .. 16) {
            Round(
                s[(0 - i) & 0x07],
                s[(1 - i) & 0x07],
                s[(2 - i) & 0x07],
                s[(3 - i) & 0x07],
                s[(4 - i) & 0x07],
                s[(5 - i) & 0x07],
                s[(6 - i) & 0x07],
                s[(7 - i) & 0x07],
                Constants[i],
                w[i] = get(chunk, i),
            );
        }

        foreach (i; 16 .. 64) {
            w[i & 0x0f] += SmallSigma1(
                w[(i + 14) & 0x0f]);
            w[i & 0x0f] +=
                w[(i + 9) & 0x0f];
            w[i & 0x0f] += SmallSigma0(
                w[(i + 1) & 0x0f]);
            Round(
                s[(0 - i) & 0x07],
                s[(1 - i) & 0x07],
                s[(2 - i) & 0x07],
                s[(3 - i) & 0x07],
                s[(4 - i) & 0x07],
                s[(5 - i) & 0x07],
                s[(6 - i) & 0x07],
                s[(7 - i) & 0x07],
                Constants[i],
                w[i & 0x0f],
            );
        }

        foreach (i; 0 .. 8) {
            state[i] += s[i];
        }
    }
}
```

Integrity

- Provided by a **cryptographic hash** of the message
- Require either **confidentiality** or **authenticity** of the hash
- Fancier schemes can be used such as error correcting code.
  - Solomon Reed for instance.

Authenticity

Authenticity



Walter

Hey Walter,
Andrei here…

HMAC

HMAC

- If the key is too large hash the key.
- Compute i = $H(K0||m)$ with
  - $K0$ the key with each byte XORed with 0x5c
  - m the message.
  - H the hash function.
- Compute h = $H(K1||i)$ with
  - $K0$ the key with each byte XORed with 0x36
- The double hashing protects against length extension attacks. It is not required with modern hash functions.

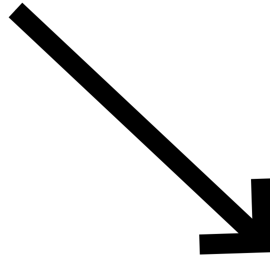
HMAC

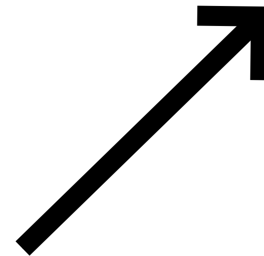
It's in the standard lib!

Confidentiality

Confidentiality

Andrei



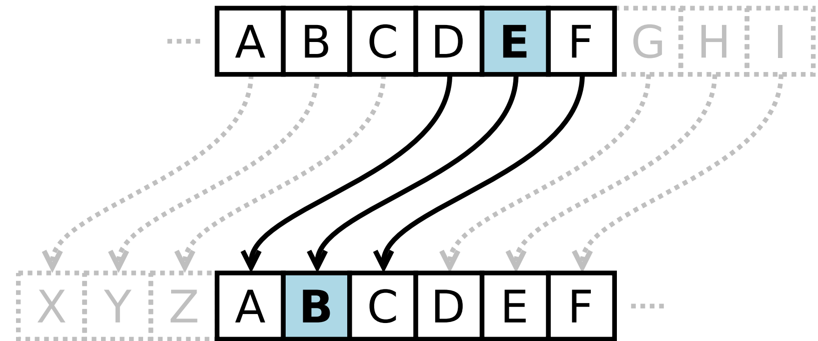


Walter

CAESAR cipher

Shift each letter of a message by some amount.

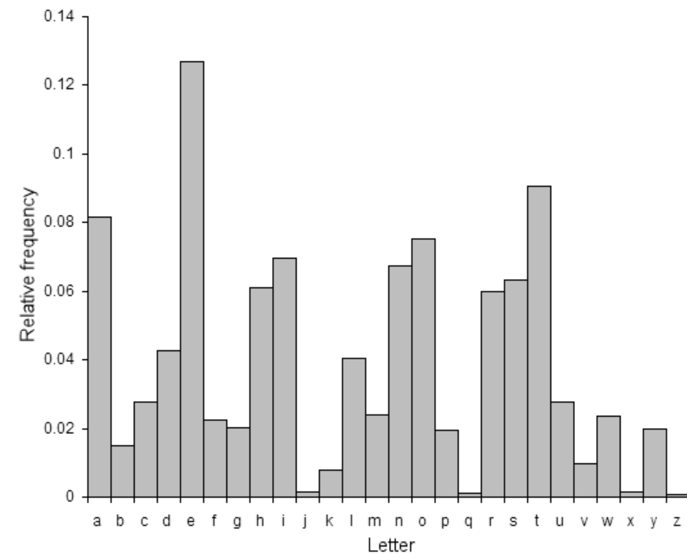
This amount is the key.



CAESAR cipher

Easy to try all the keys, less than 5 bits of entropy.

Easy to break via statistical analysis.



Vigenère cipher

Use multiple characters as a key. Repeat the key to match message length.

Solve the entropy problem.

Do not solve the statistical analysis problem.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

Stream cipher

PRGN

Instead of using the key to encrypt, use the key to seed a PRNG.

Message is XORed with the stream to encrypt.

PRGN must do **confusion** and **diffusion**.

IV

Reusing a pseudo random stream reintroduce statistical attack vectors.

A new **initialization vector** is used for each messages to generate a new stream.

Can't regenerate the stream without the **secret key**. The IV is not secret.

Using a hash function as PRNG

```
ubyte[32] getPseudoRandom(
    ubyte[32] key,
    ubyte[16] iv,
    ulong i,
) {
    H hasher;

    hasher.start();
    hasher.put(key);
    hasher.put(iv);
    hasher.put(i);

    return hasher.finish();
}
```

But just use ChaCha20

Block cipher

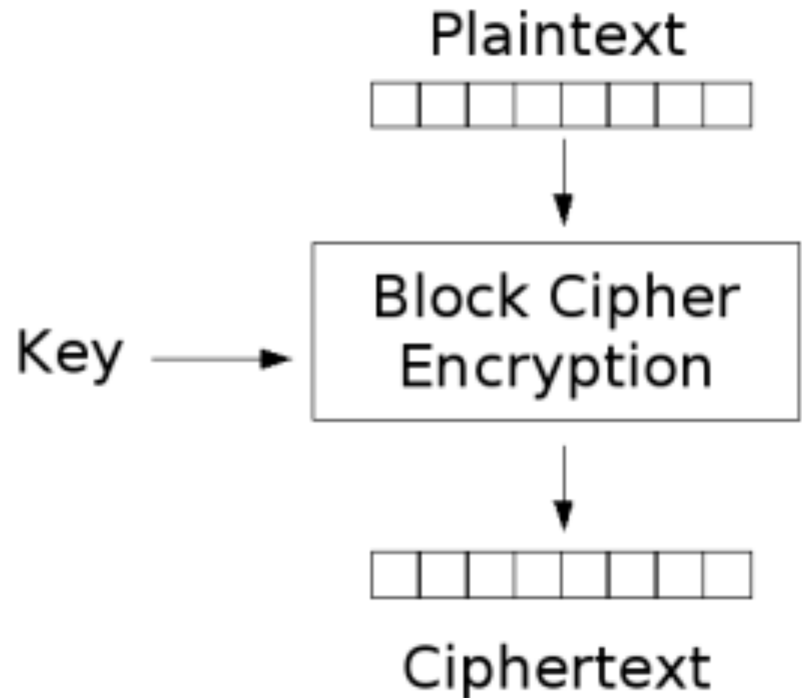
Block cipher

It is a useful crypto **building block**.

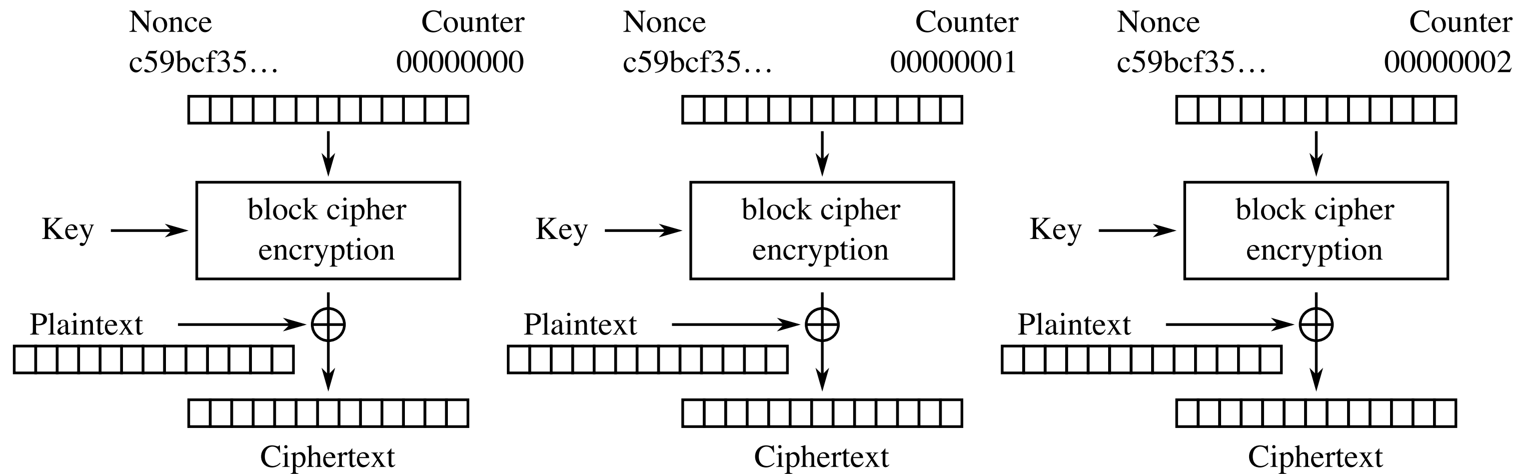
It takes a **secret key** and a **message** as input.

It output random looking **ciphertext**.

ciphertext can be decrypted using the **secret key**.

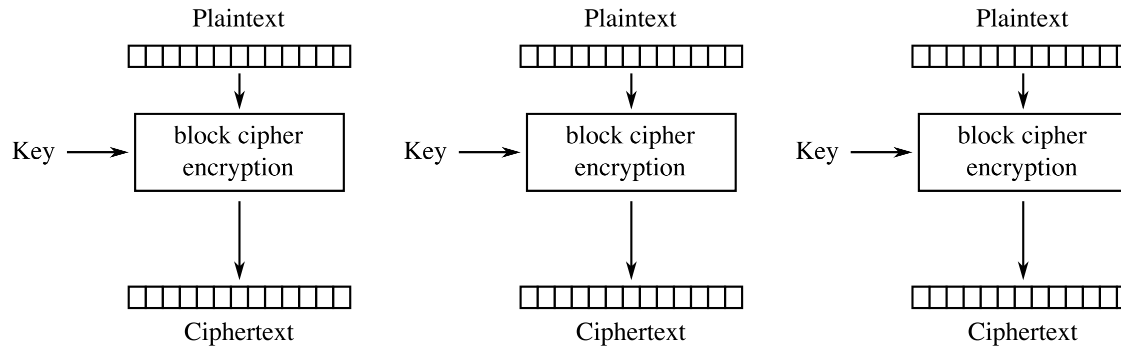


CTR: Stream cipher

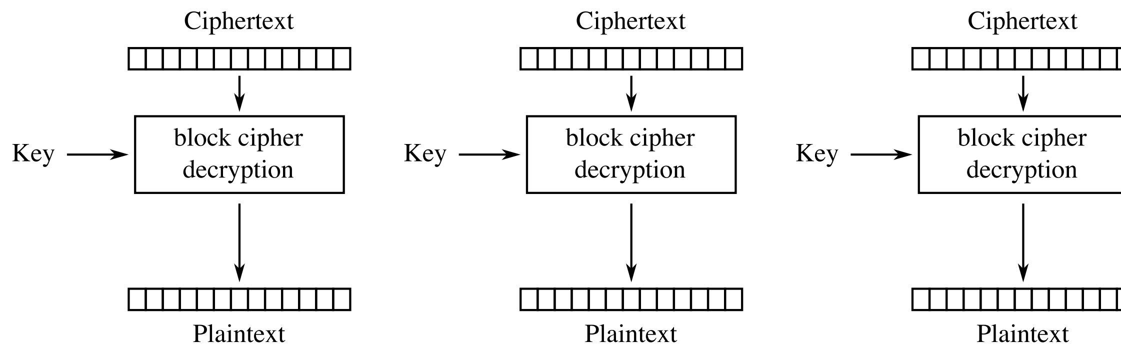


Counter (CTR) mode encryption

ECB: insecure cipher

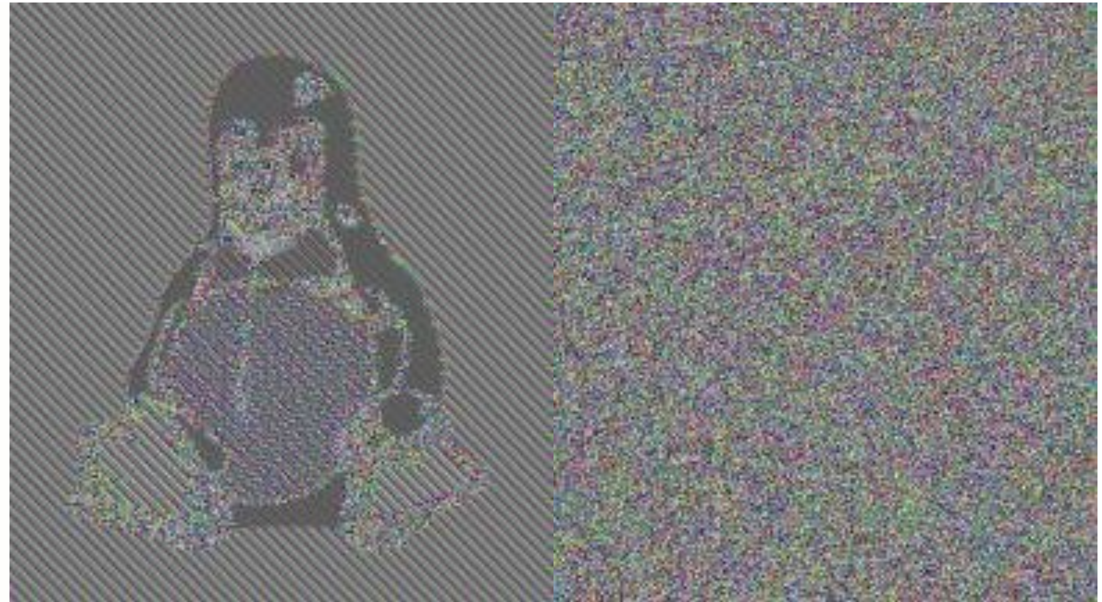


Electronic Codebook (ECB) mode encryption



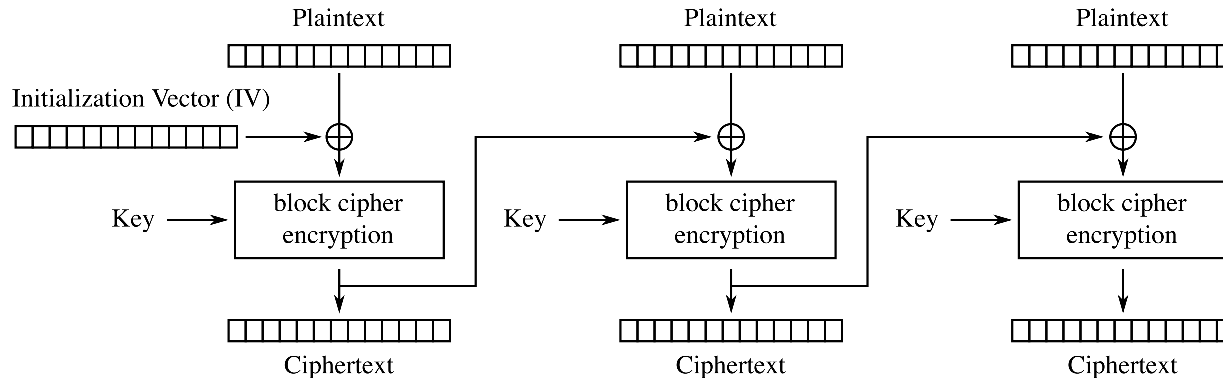
Electronic Codebook (ECB) mode decryption

ECB: insecure cipher

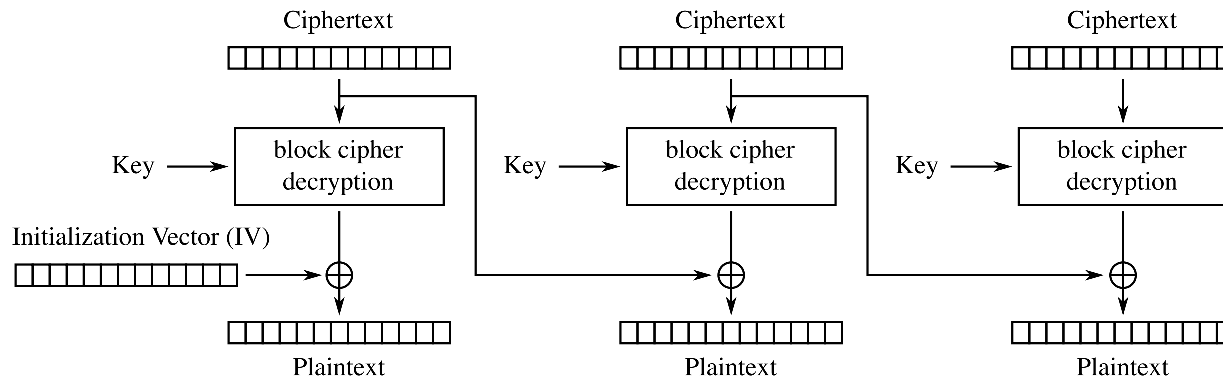


Tux Tux ECB Tux encrypted

CBC: most common cipher



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

Advanced Encryption Standard

S box

A map of all possible 8bits value to a new 8bits value.

Contribute to **confusion** and **diffusion**.

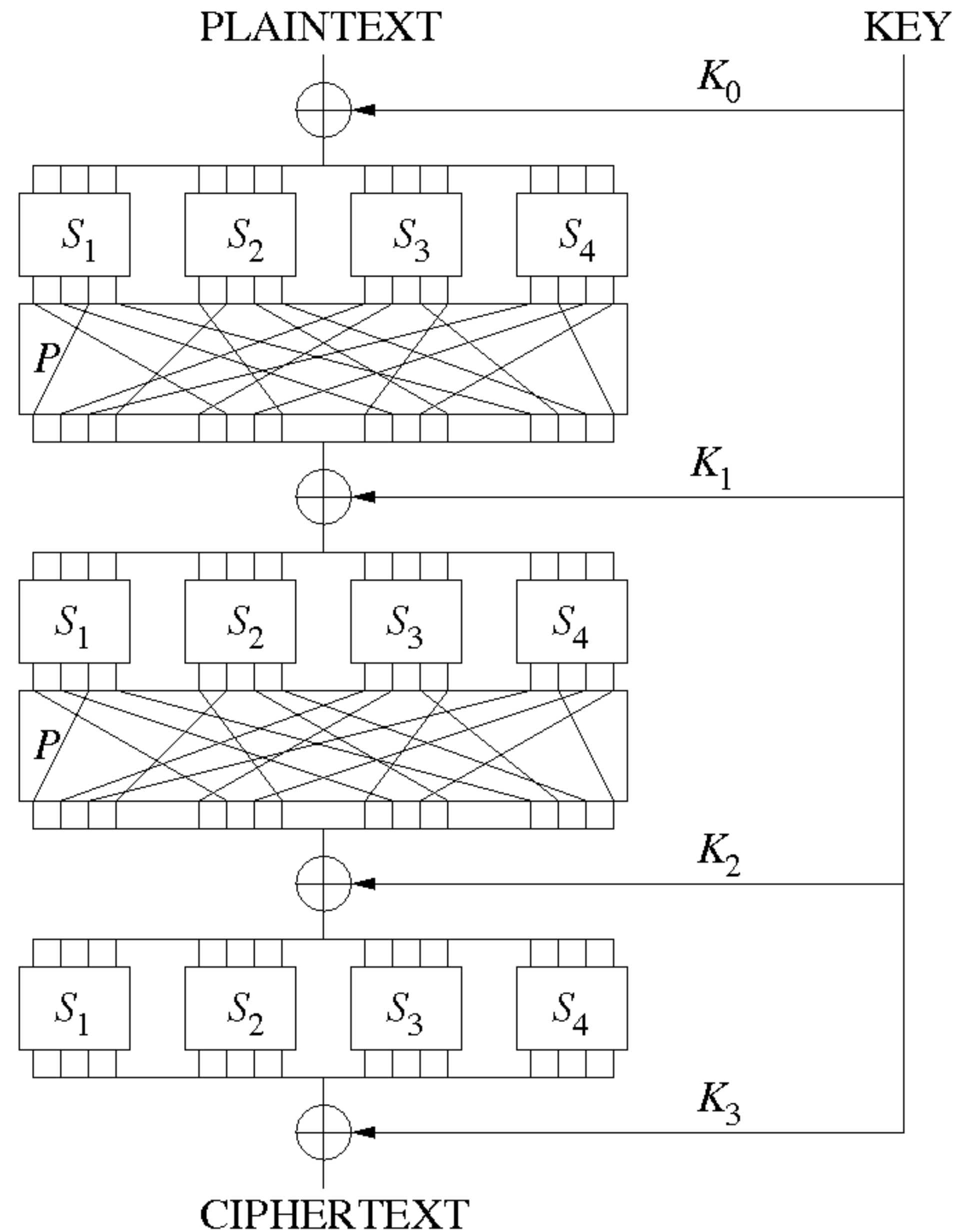
	00	01	02	03	04	05	06	07	08	09	0a	0b	0c	0d	0e	0f
00	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
10	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
20	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
30	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
40	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
50	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
60	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
70	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
80	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
90	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
a0	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
b0	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
c0	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
d0	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
e0	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
f0	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

P box

Shuffle bits across S boxes.

After several rounds, **confusion** and **diffusion** are total.

Between each round, the result is XORed with data generated from the key.



Implementation

- **P box** and **key expansion** are straightforward.
- Naive **S box** implementation is a lookup table.
  - **Not secure !**

"Running this on the Athlon64 with knowledge about address mappings, we succeeded in extracting the full key after just 800 write operations done in 65ms (including the analysis of the cache state after each write)"

http://www.cs.tau.ac.il/~tromer/papers/cache-joc-20090619.pdf

No data flow from secret to load address

Prime Field

- Operations modulo a **prime number**
  - Close to a power of 2
- Inversion is defined as the multiplicative inverse

$$x^{-1} * x = 1$$

Galois Field $GF(p^n)$

- p is a **prime number**.
- Elements of the field are a set of n elements for the prime field defined by p.
  - Similar to complex numbers.
- Addition/subtraction are done on prime field elements independently.
  - $(a, b) + (c, d) = (a + c, b + d)$

Galois Field $GF(p^n)$

- Multiplication generate $2n - 1$ elements
  - $(a, b) * (c, d) = (a * c, a * d + b * c, b * d)$
  - Reduce to n elements using an irreducible polynomial
  - $(a, b) * (c, d) = (a * d + b * c, b * d - a * c)$ for polynomial $X^2 + 1$
- Division rely on multiplicative inverse.

Galois Field $GF(2^n)$

- Addition and subtraction are just XOR
- Multiplication subtract the polynomial until the result is within bounds.
- AES uses $GF(2^8)$ for its S box and the polynomial $X^8 + X^4 + X^3 + X + 1$
  - Element of the S box are computed by taking the multiplicative inverse of the input and applying some affine transformation.
  - Computationally intensive.
  - Latest Intel offer implementation in hardware.

Confidentiality

- Tricky to implement right
  - **No load address depending on the secret.**
- Use **ChaCha20** or **AES**
  - **No ECB mode.**

Commitments

Confidentiality

Keep the message confidential.

Integrity

Can prove the message existed and its content wasn't altered.

Confidentiality Integrity

- Cannot be a simple hash because entropy of the message may not be sufficient.
- Use a blinding factor, sometime called salt.
- $c = H(R||m)$ with
  - H the hash function
  - R the blinding factor
  - m the message.
- Perfect for passwords.

Asymmetric crypto

Math

Rely on math problem which are easy in one way but hard in the other.

For instance prime factorization.

Here we'll use elliptic curve.

No shared secret

Participant do not need to share a secret.

It is easier to keep the secret secret.

Eliptic curves

- $y^2 = x^3 + a * x + b$
- Defined over a finite field
- Can define point addition and subtraction
- Using $P + P = 2 * P$ we can define scalar point multiply
- Given $P = x * G$ it is pretty much impossible to find x given P and G

secp256k1

- NIST's "Standards for Efficient Cryptography" (SEC2)
- $y^2 = x^3 + a * x + b$
  - $a = 0$
  - $b = 9$
- Defined over a prime field
  - 0xFFFFFFFFFFFFFFFF_FFFFFFFFFFFFFFFF_FFFFFFFFFFFFFFFF_FFFFFFFEFFFFFC2F
  - close to a power of 2 so we can optimize
- A point G is chosen and called **generator**

Scalar

- Scalar are defined over a finite field
- Operation modulo the curve order
    - 0xFFFFFFFFFFFFFFFF_FFFFFFFFFFFFFFFE_BAAEDCE6AF48A03B_BFD25E8CD036414
      1
    - That's how many points do exist on the curve
- Large 256 bits integer operation required
    - Leverage ucent and compiler legalization for good quality codegen
        - Had to use SDC
        - Patched LLVM so it gets it just right

Scalar

```
static addImpl(Scalar a, Scalar b) {
        ulong[4] r;
        ucent acc;

        foreach (i; 0 .. 4) {
                acc += a.parts[i];
                acc += b.parts[i];
                r[i] = cast(ulong) acc;
                acc >>= 64;
        }

        return AddResult(r, !!(acc & 0x01));
}
```

Scalar

```
static select(bool cond, Scalar a, Scalar b) {
        auto maska = -ulong(cond);
        auto maskb = ~maska;

        ulong[4] r;
        foreach (i; 0 .. 4) {
                // NB: The compiler is still uses CMOV.
                auto ra = a.parts[i] & maska;
                auto rb = b.parts[i] & maskb;
                r[i] = ra | rb;
        }

        return Scalar(r);
}
```

Scalar

```
auto opCmp(Scalar b) const {
        auto bp = b.getParts();

        int bigger;
        int smaller;
        foreach_reverse (i; 0 .. 4) {
                // The higher ILP version require a few extra instructions.
                // TODO: Need to benchmark which one is best.
                enum WithILP = false;
                static if (WithILP) {
                        auto isBigger  = (parts[i] > bp[i]) & ~smaller;
                        auto isSmaller = (parts[i] < bp[i]) & ~bigger;

                        bigger  |= isBigger;
                        smaller |= isSmaller;
                } else {
                        bigger  |= (parts[i] > bp[i]) & ~smaller;
                        smaller |= (parts[i] < bp[i]) & ~bigger;
                }
        }

        return bigger - smaller;
}
```

Scalar

- No secret dependent control flow.
- No secret dependent addresses.
- No bound checks.

Field element

- Field elements are defined in a finite field
- Large 256 bits integer operation required
- Prime close to a power of 2
  - Define the element as 1*48bits + 4*52bits elements
  - Let carries accumulate
  - Keep track of how many carries we accumulated at worse and normalize
    - No exact tracking as it would open side channels.

Field element

```
auto normalize(ComputeElement e) {
        // We start by reducing all the MSB bits in part[4]
        // so that we will at most have one carry to reduce.
        parts = e.parts;
        ulong acc = (parts[4] >> 48) * Complement;

        // Clear the carries in part[4].
        parts[4] &= MsbMask;

        // Propagate.
        foreach (i; 0 .. 5) {
                acc += parts[i];
                parts[i] = acc & Mask;
                acc >>= 52;
        }

        assert(acc == 0, "Residual carry detected");
        return ComputeElement(parts);
}
```

Field element

```
auto add(ComputeElement b) const {
        auto a = this;
        ulong[5] parts;
        foreach (i; 0 .. 5) {
                parts[i] = a.parts[i] + b.parts[i];
        }

        auto cc = a.carryCount + b.carryCount + 1;
        auto r = ComputeElement(parts, cc);

        // We can branch on carryCount because it is only dependent on
        // control flow. If other part of the code do not branch based
        // on values, then carryCount do not depend on value.
        if (cc < 2048) {
                return r;
        }

        // We have 12bits to accumulate carries.
        // It means we can't add numbers which accumulated
        // 2048 carries or more.
        auto nr = NormalizationResult(r);
        return nr.raw;
}
```

Point multiplication

For G

- Use heavy **precomputation**.
  - All multiples of G per 4 bits blocks.
  - [0, 1, 2 … 15]
  - [16, 32, … 240]
  - …
- Then blind the tables.
- **Access all elements in the table to avoid secret dependent address loads.**

Other points

- Compute **w-NAF** representation of the scalar.
- Compute **odd multiple table**.
- Multiply via doubling and adds from the table.
- **Access all elements in the table to avoid secret dependent address loads.**

Blinding 1

- Chose H such as x is unknown in $H = x * G$
- Add $a0 * H$ to all elements in the first line.
- Add $a1 * H$ to all elements in the second line.
- ...
- Subtract $(a0 + a1 + ... + a14) * H$ to all elements in the last line.

Blinding 2

- Add $b0 * G$ to all elements in the first line.
- Add $b1 * G$ to all elements in the second line.
- ...
- Add $b15 * G$ to all elements int he last line.
- Compute $k = x - b$ with $b = b0 + b1 + ... + b15$
- Use k for table lookups.

w-NAF

- Compute the multiple table of the point
  - $[1, 3, 5, \ldots, 2^w - 1]$
- Scalar is decomposed in a series of indices to lookup and a sign bit.
  - Sign bit is stored in LSB : *s = d & 0x01*
  - Index is stored in other bits: *i = d >> 1*
- Multiply by doubling and adds

w-NAF

```
// For the initial value, we can just look it up in the table.
auto first = select(table, lookup[Steps - 1]);
auto r = first.pdoublen!N();
r = r.add(select(table, lookup[Steps - 2]));

/**
 * The core multiplication routine. We double N times and
 * add the value looked up from the table each round.
 */
foreach (i; 2 .. Steps) {
        r = r.pdoublen!N();
        r = r.add(select(table, lookup[Steps - 1 - i]));
}
```

w-NAF

```
static select(ref Point[TableSize] table, ubyte n) {
        // The least significant bit is the sign. We get rid of it
        // to get the index we are interested in in the table
        auto idx = n >> 1;

        /**
         * We want to avoid side channels attacks. One of the most common
         * side channel is memory access, as it impact the cache. To avoid
         * leaking the secret, we make sure no memory access depends on the
         * secret. This is achieved by accessing all elements in the table.
         */
        auto p = table[0];
        foreach (i; 1 .. TableSize) {
                p = Point.select(i == idx, table[i], p);
        }

        // Finally we negate the point if the sign is negative.
        auto positive = (n & 0x01) != 0;
        return Point.select(positive, p, p.negate());
}
```

w-NAF

- The number need to be odd.
  - For divisor of 255 add a skew of 1 or 2
  - For other number, negate if even.
- $255 = 3 * 5 * 17$
  - 3 and 5 are especially useful
  - Allow to shave one iteration off

w-NAF

```
void buildLookup(Scalar s) {
        /**
         * w-NAF require that the scalar is odd so ScalarBuf will
         * negate even scalars.
         */
        auto buf = ScalarBuf(s, skew);
        auto flipsign = !!(skew & 0x01);

        static pack(int u, bool flipsign) {
                /**
                 * If u is positive, this is a noop. If it is negative, then
                 * all bits are flipped. Because the LSB is known to be 1,
                 * flipping the bits are the same as in the complement.
                 *
                 * The LSB is 0 for negative, 1 for positive, higher bits
                 * are the absolute value and can be used as indices.
                 */
                return ubyte(((u ^ flipsign) ^ (u >> 31)) & 0xff);
        }

        auto u = buf.extract();
        foreach (i; 1 .. Steps) {
                auto bits = buf.extract();

                /**
                 * If the current number is even, we need to correct it such as
                 * it is odd, so we create an all ones mask if even, 0 if odd.
                 */
                auto even = (bits & 0x01) - 1;
```

w-NAF

```
            /**
             * To make it odd, we can either add or remove 1. We want
             * the previous digit to stay in range, so if it is positive,
             * we produce a 1, or a -1 if it isn't.
             */
            auto sign = (u >> 31) | 0x01;

            // We add or remove 1 to make this odd.
            bits += (sign & even);

            /**
             * We compensate the addition in the previous digit by adding or
             * removing 16. We knows it stays in range because we subtract or
             * add depending on its sign, and because it is odd, so non zero.
             */
            u -= ((sign & even) << N);

            // We computed one w-NAF digit, pack it.
            lookup[i - 1] = pack(u, flipsign);

            // Get ready for the next round.
            u = bits;
        }

        // Last digit, already corrected.
        lookup[Steps - 1] = pack(u, flipsign);
}
```

ECDH

- Andrei chooses a and compute Pa = $a * G$
- Walter chooses b and compute Pb = $b * G$
- Andrei and Walter exchange Pa and Pb
- Andrei compute S = $a * Pb$
- Walter compute S = $b * Pa$
- Both get S = $a * b * G$
  - They can now use symmetric crypto securely.

Schnorr Signature

- Patented until 2009 so it is usable now but wasn't in the past.
  - ECDSA more used in the wild now
  - Schnorr is faster, can be batch validated, and can be extended to provide various features
  - It is also much simpler
- Andrei has a private key x and a public key $P = x * G$
  - Creating the signature require x but verifying only requires P

Schnorr Signature

- The signature is a pair (R, s)
- Walter compute e = H(R||P||m)
- He verify that R = $e * P$ + $s * G$
- R goes in the hash function, so Andrei must have chosen s such as it cancels e
  - Or he broke the hash function
  - How does he do it ?

Schnorr Signature

- Andrei chose a random number k
- He computes $R = k * G$ and $e = H(R||P||m)$
- He needs to finds s such as $R = e * P + s * G$
- He knows
  - $R = k * G$
  - $P = x * G$
  - So $k * G = e * x * G + s * G$
  - He deduces $s = k - e * x$
- k is a blinding factor, it needs to be kept secret.
  - Best thrown away after the signature is generated
  - Use entropy from x to avoid requiring randomness
    - $k = H(x||nonce||m)$

Ring Signature

- Any person in a group can sign.
- There are no way to know who sign.
- Signature is the tuple ($R0$, $s0$, $s1$, ..., sn)
- To verify compute
  - $R1 = H(R0||P0||m) * P0 + s0 * G$
  - $R2 = H(R1||P1||m) * P1 + s1 * G$
  - ...
- Verify that $R0 = H(Rn||Pn||m) * Pn + sn * G$
  - Someone was able to chose one of the s such as the ring can be closed.
  - Impossible to tell which one.

Ring Signature

- Signer 0 chose k such as $R1 = k * G$
- Chooses random value for $s1, s2, ..., sn$
- Compute $R2, ..., Rn, R0$
- Compute $e = H(R0 || P0 || m)$
- Compute $s0 = k - e * x$
- Produce the signature.

Homomorphic commitment

- Use a second point H on the curve
  - No x such as $G = x * H$ is known
- Andrei will prove that he has 2 numbers a and b summing up to 10
  - $Ca = a * G + ra * H$ and $Cb = b * G + rb * H$
  - The commitment is $(Ca, Cb, r = ra + rb)$
- Walter verify that $Ca + Cb = 10 * G + r * H$
  - He knows that $a + b = 10$ but has no idea of the value of a and b

Side channel

- **No load address depending on the secret.**
- **No control flow depending on the secret.**
- **Avoid CMOV depending on the secret.**
- **Disable automatic check such as bound checks.**
  - **Just for the code handling the secret.**
- **The compiler is a smart ass, inspect codegen.**

Code

https://github.com/deadalnix/schnorr