

Cryptographic Implementation Attacks

Check Point

June 25, 2010

Joseph Bonneau

Security Group

jcb82@cl.cam.ac.uk

Insecure MAC checking routine #1

```
int check_MAC(u_char * test, u_char * correct){  
    return (strcmp(test, correct) == 0);  
}
```

Insecure MAC checking routine #1

```
int check_MAC(u_char * test, u_char * correct){  
    return (strcmp(test, correct) == 0);  
}
```

```
int strcmp(u_char *s1, u_char *s2){  
    while (*s1 != '\0' && *s1 == *s2) {  
        s1++;  
        s2++;  
    }  
  
    return ((*s1 < *s2) ? -1 : (*s1 > *s2));  
}
```

Insecure MAC checking routine #1

```
int check_MAC(u_char * test, u_char * correct){  
    return (strcmp(test, correct) == 0);  
}
```

```
int strcmp(s1, s2){  
    while(*s1 & *s2){  
        s1++;  
        s2++;  
    }  
  
    return ((*s1 < *s2) ? -1 : (*s1 > *s2));  
}
```

≈ |A| queries ☹

Insecure MAC checking routine #2

```
int check_MAC(u_char * test, u_char * correct){  
    return (strcmp(test, correct, LEN) == 0);  
}
```

```
int memcmp(u_char *s1, u_char *s2, size_t n){  
    while (n-- != 0) {  
        if (*s1 != *s2)  
            return 1;  
  
        s1++;  
        s2++;  
    }  
  
    return 0;  
}
```

Insecure MAC checking routine #2

```
int check_MAC(u_char * test, u_char * correct){  
    return (strcmp(test, correct, LEN) == 0);  
}
```

```
int memcmp(u_char *s1, u_char *s2, size_t n){  
    while (  
        if (*s1 != *s2)  
            return 1;  
        s1++;  
        s2++;  
    }  
  
    return 0;  
}
```

$\approx n \cdot |A|$ queries

MAC timing attack against Xbox 360

```
C:\Documents and Settings\Administrator\Desktop\360\DGTool>DGTool.exe 1 Infectus
_5759#4_1888_build2.bin
```

```
Pairing Data 0x6DF3B8 01
```

```
Turn on your Xbox, press any key when the RRoD starts
```

```
H[0 00XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17821 A 17821 D 0 : 0 NEXT
H[0 01XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17817 A 17819 D -3 : 0 NEXT
H[0 02XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17822 A 17820 D 3 : 0 NEXT
...
H[0 1AXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17822 A 17819 D 3 : 0 NEXT
H[0 1BXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17831 A 17819 D 12 : 11 RPT
H[0 1BXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17830 A 17819 D 11 : 0 HIT!
H[1 1B00XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17830 A 17830 D 0 : 0 NEXT
H[1 1B01XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17829 A 17829 D -1 : 0 NEXT
...
H[1 1BFDXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17844 A 17830 D 14 : 8 RPT
H[1 1BFDXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17839 A 17830 D 9 : 0 HIT!
H[2 1BFD00XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17839 A 17838 D 1 : 0 NEXT
H[2 1BFD01XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17844 A 17841 D 4 : 0 NEXT
...
H[2 1BFDF0XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17856 A 17841 D 15 : 11 RPT
H[2 1BFDF0XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17851 A 17841 D 10 : 0 HIT!
...
H[15 1BFDF0625C214F67CD94DCA3FC47CA55] M 18014 A 17988 D 16 : 0 HIT!
```

```
Correct hash: 1BFDF0625C214F67CD94DCA3FC47CA55
```

```
Result: BOOT
```

MAC timing attack against Xbox 360

```
C:\Documents and Settings\Administrator\Desktop\360\DGTool>DGTool.exe 1 Infectus
_5759#4_1888_build2.bin
```

```
Pairing Data 0x6DF3B8 01
```

```
Turn on your Xbox, press any key when the RRoD starts
```

```
H[0 00XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17821 A 17821 D 0 : 0 NEXT
```

```
H[0 01XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17817 A 17819 D -3 : 0 NEXT
```

```
H[0 02XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17822 A 17820 D 3 : 0 NEXT
```

```
...
```

```
H[0 1AXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17822 A 17819 D 3 : 0 NEXT
```

```
H[0 1BXXXXXXXXXX
```

```
H[0 1BXXXXXXXXXX
```

```
H[1 1B00XXXXXXX
```

```
H[1 1B01XXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17829 A 17829 D -1 : 0 NEXT
```

```
...
```

```
H[1 1BFDXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17844 A 17830 D 14 : 8 RPT
```

```
H[1 1BFDXXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17839 A 17830 D 9 : 0 HIT!
```

```
H[2 1BFD00XXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17839 A 17838 D 1 : 0 NEXT
```

```
H[2 1BFD01XXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17844 A 17841 D 4 : 0 NEXT
```

```
...
```

```
H[2 1BFDF0XXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17856 A 17841 D 15 : 11 RPT
```

```
H[2 1BFDF0XXXXXXXXXXXXXXXXXXXXXXXXXXXXX] M 17851 A 17841 D 10 : 0 HIT!
```

```
...
```

```
H[15 1BFDF0625C214F67CD94DCA3FC47CA55] M 18014 A 17988 D 16 : 0 HIT!
```

```
Correct hash: 1BFDF0625C214F67CD94DCA3FC47CA55
```

```
Result: BOOT
```

≈ 2,000 queries

Insecure MAC checking routine #3

```
int check_MAC(u_char * test, u_char * correct){  
    return (strcmp(test, correct, LEN) == 0);  
}
```

```
int safecmp(u_char *s1, u_char *s2, size_t n){  
    int result = 0;  
    while (n-- != 0) {  
        if (*s1 != *s2)  
            result = 1;  
        s1++;  
        s2++;  
    }  
  
    return result;  
}
```

Insecure MAC checking routine #3

```
int check_MAC(u_char * test, u_char * correct){  
    return (strcmp(test, correct, LEN) == 0);  
}
```

```
int safecmp(u_char *s1, u_char *s2, size_t n){  
    int  
    while
```

$\approx n \cdot |A|$ queries

```
        if (*s1 != *s2)  
            result = 1;
```

```
        s1++;
```

```
        s2++;
```

```
    }
```

```
    return result;
```

```
}
```

Secure MAC checking routine?

```
int check_MAC(u_char * test, u_char * correct){  
    return (strcmp(test, correct, LEN) == 0);  
}
```

```
int safercmp(u_char *s1, u_char *s2, size_t n){  
    int result = 0;  
    while (n-- != 0) {  
        result &= (*s1 ^ *s2);  
        s1++;  
        s2++;  
    }  
  
    return !result;  
}
```

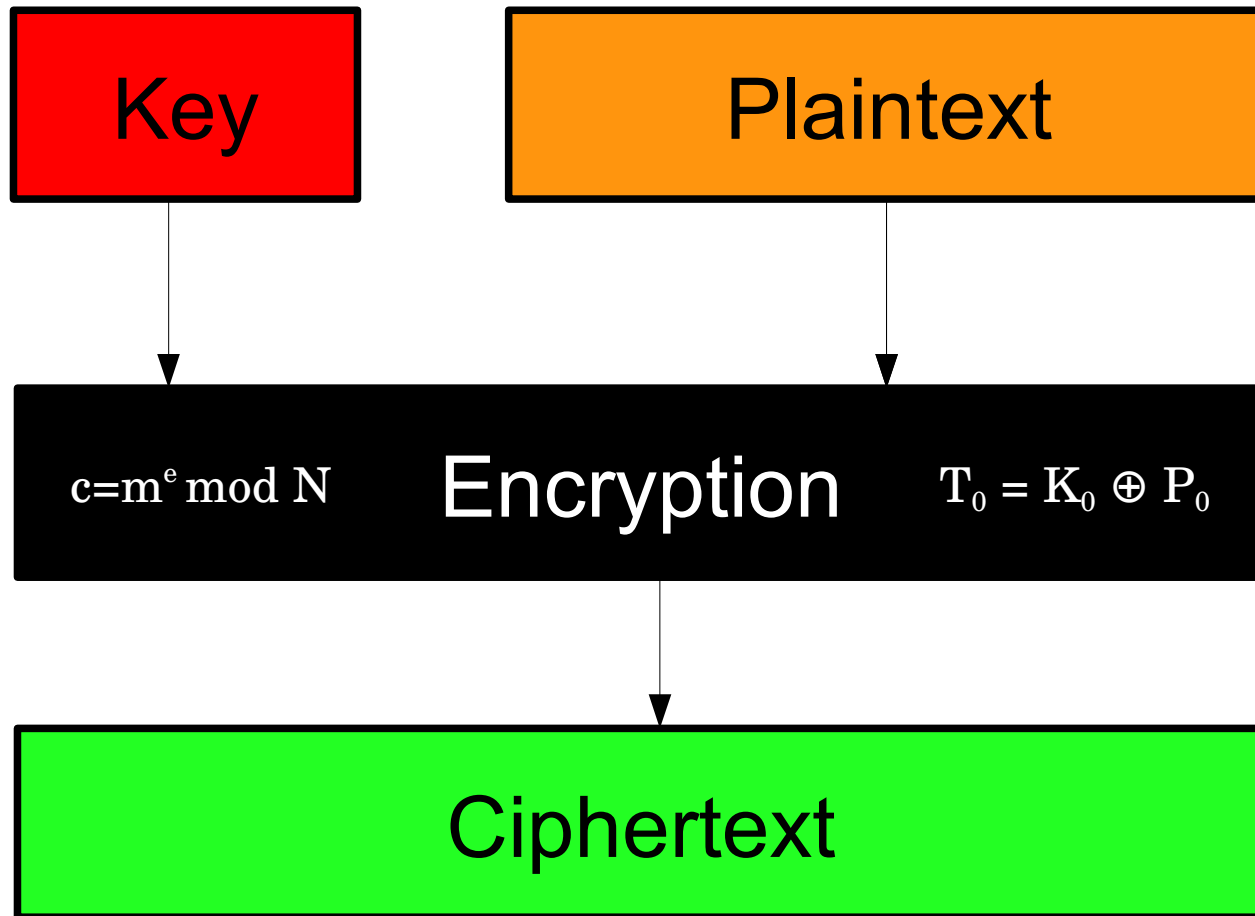
Rule #0: Attackers will **always** cheat



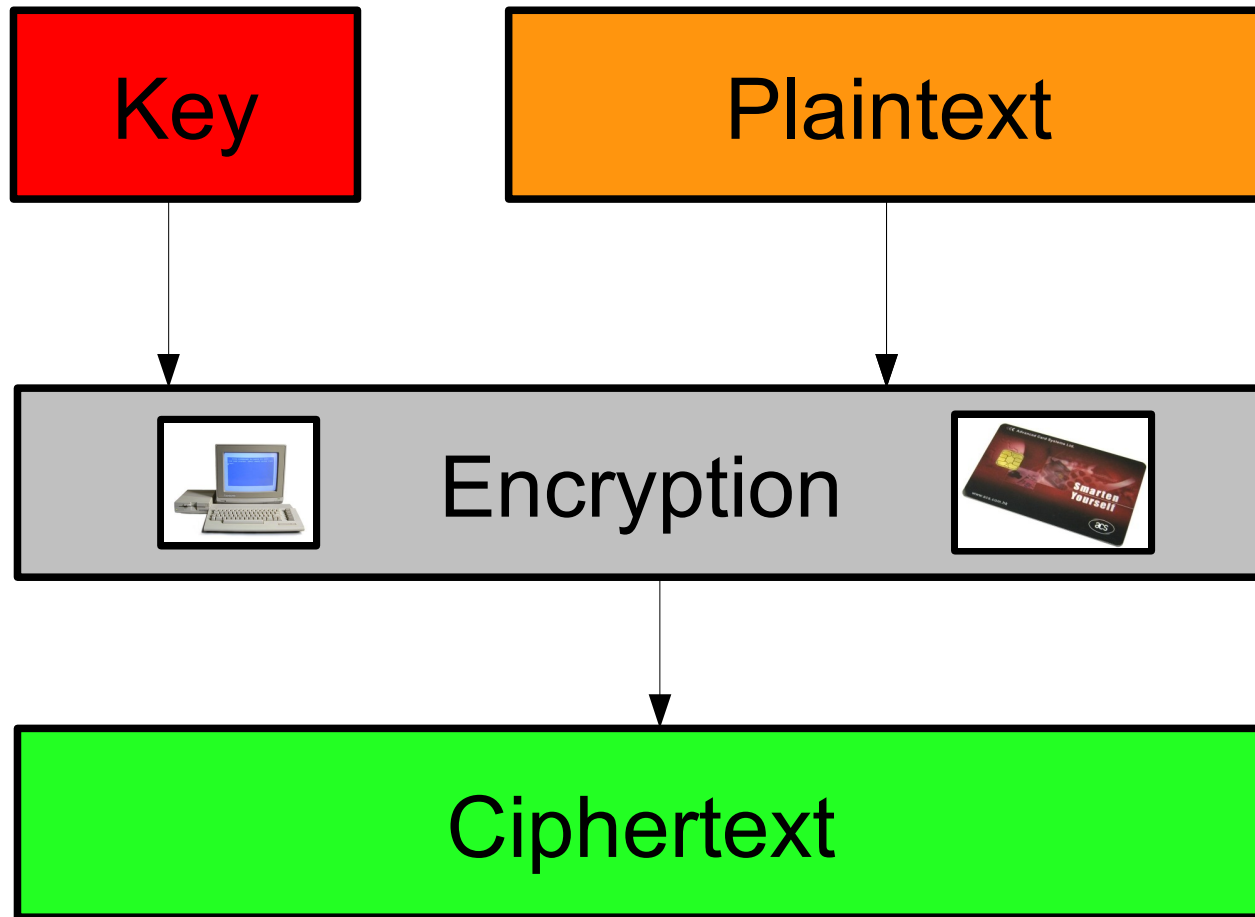
Crypto design processes ignore(d) cheating



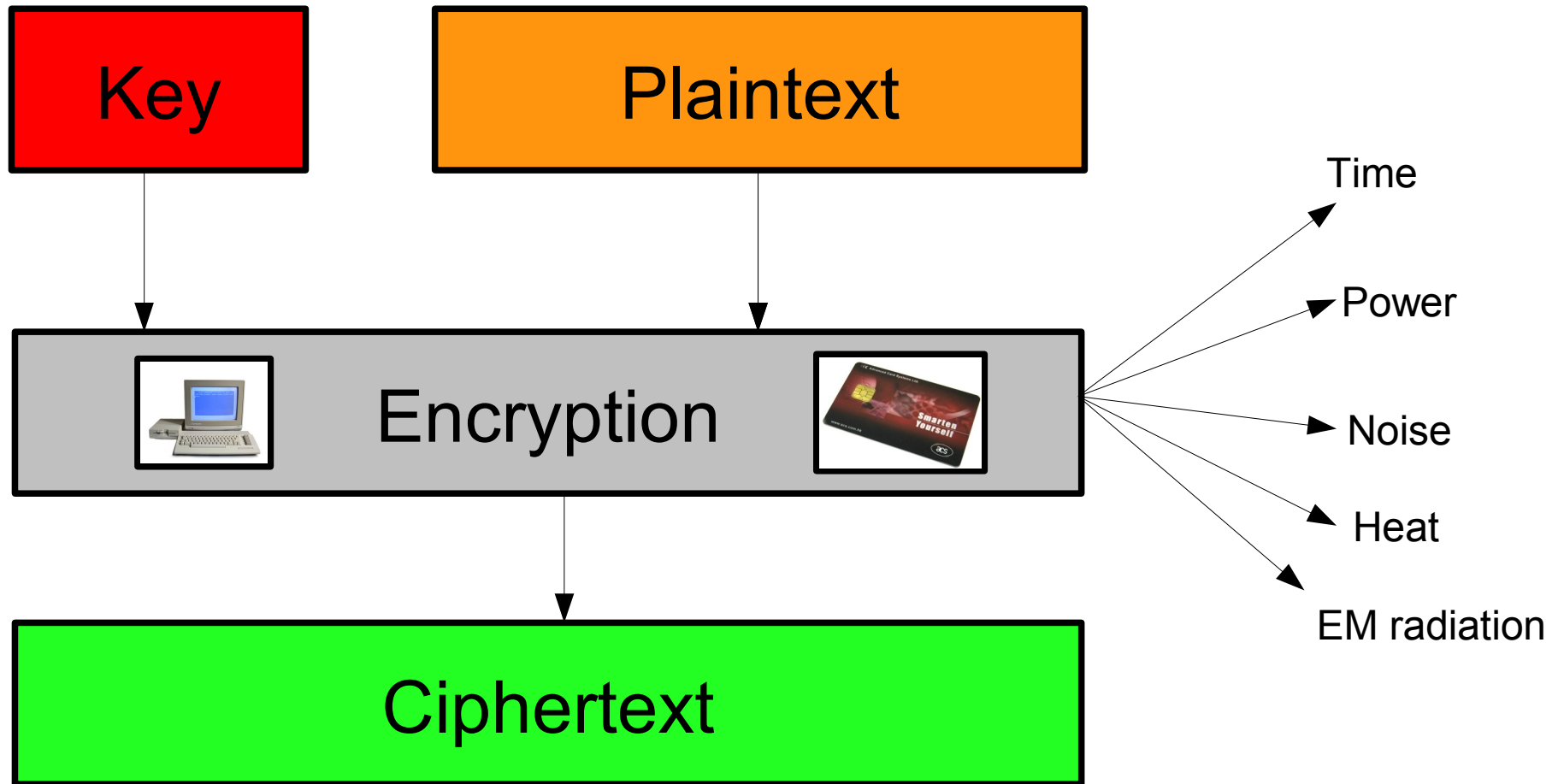
Side Channels: whatever the designers ignored



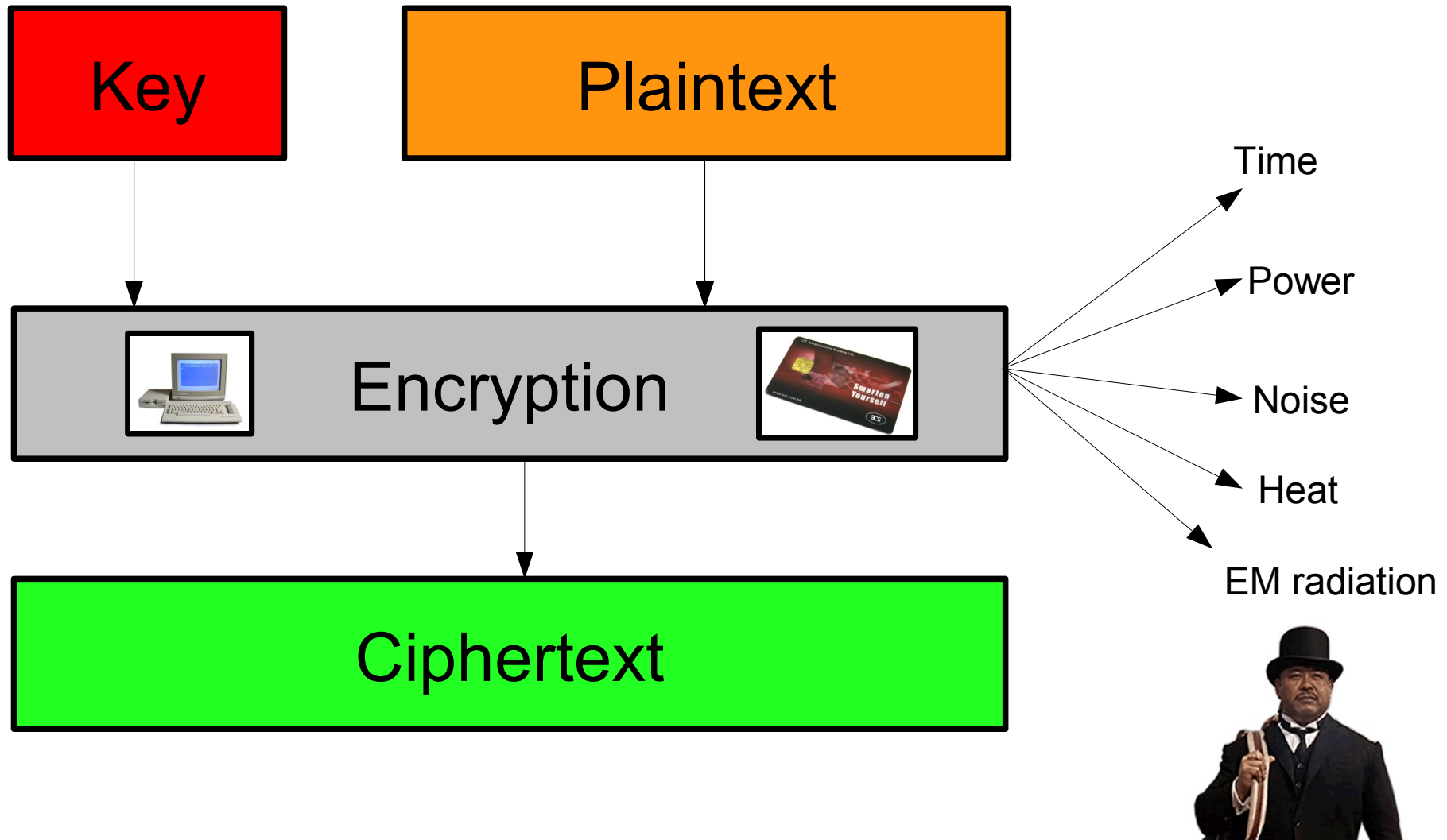
Side Channels: whatever the designers ignored



Side Channels: whatever the designers ignored



Side Channels: whatever the designers ignored

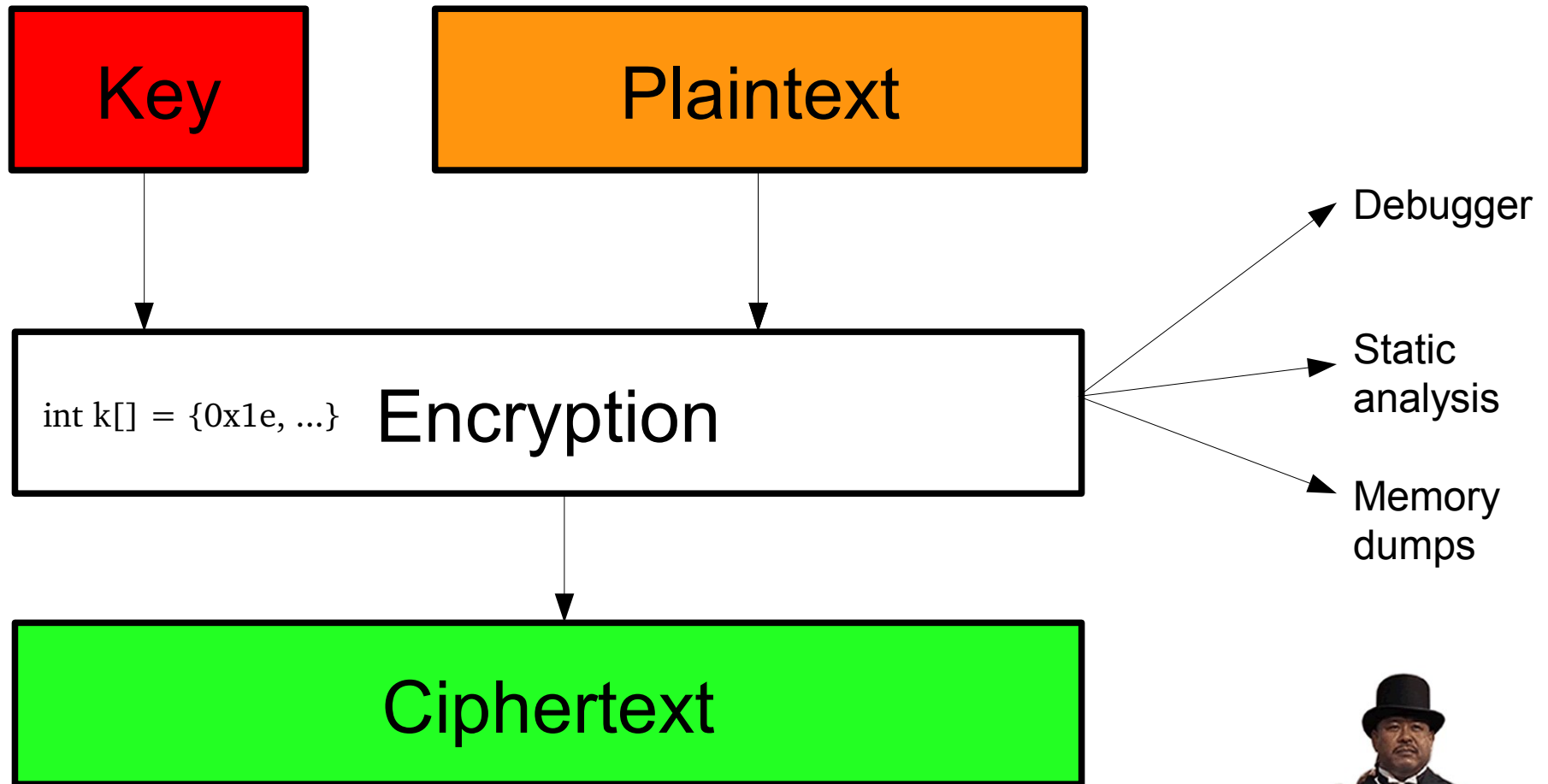


Definition

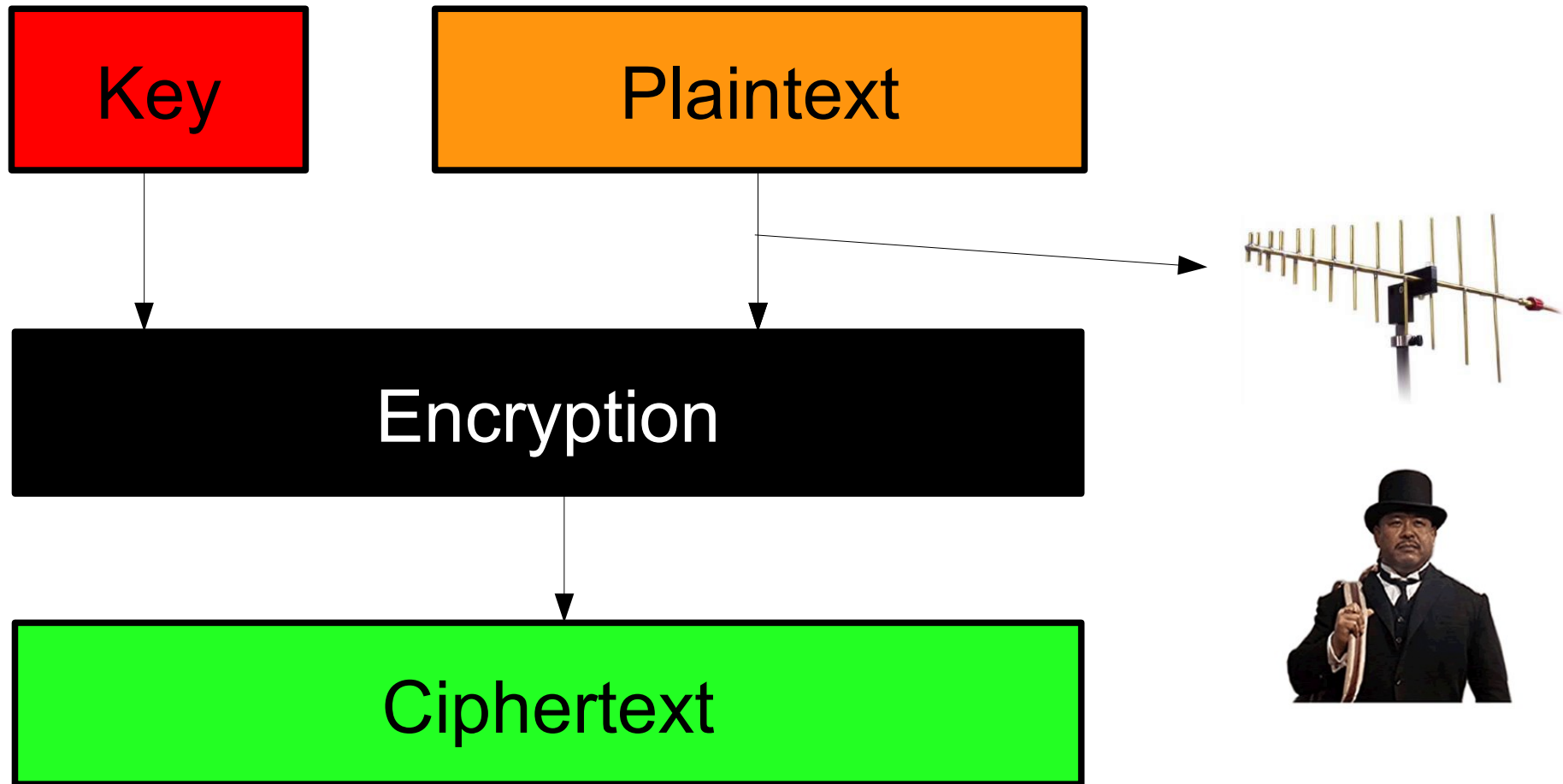
Side-channel cryptanalysis is any attack on a cryptosystem requiring information emitted as a byproduct of the physical implementation.

Related attacks

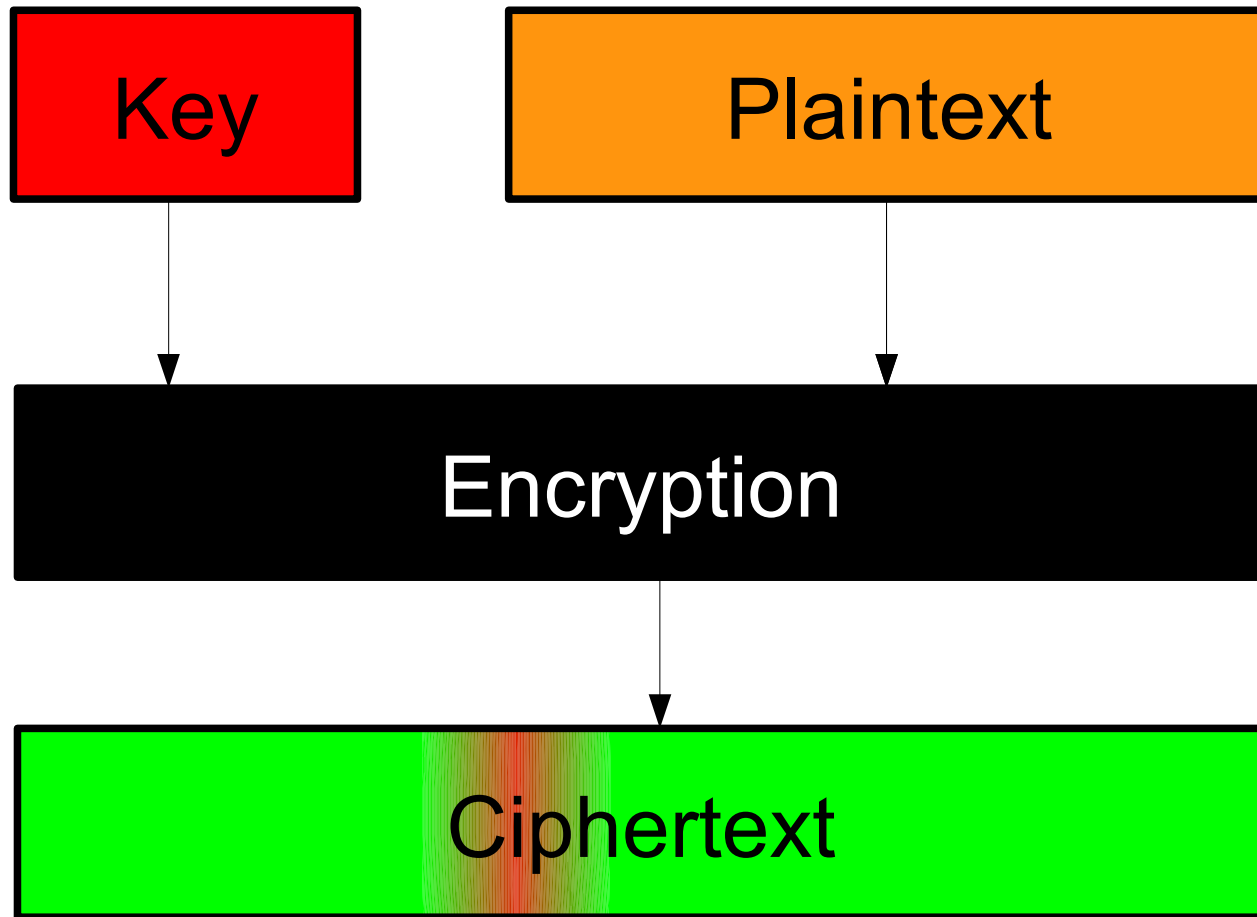
White box cryptanalysis: nothing is hidden



TEMPEST: EM signals containing full secrets

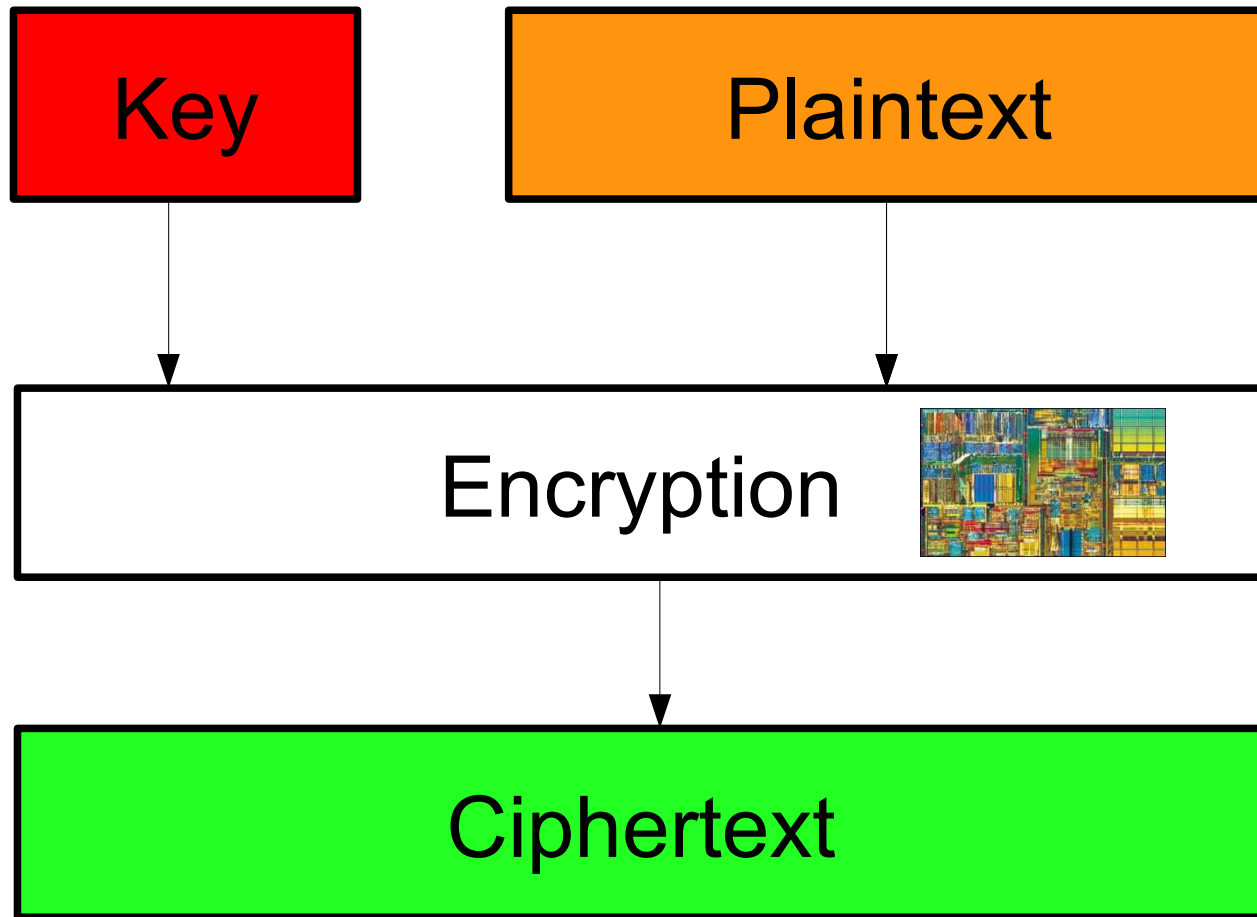


Fault injection: inducing a telling error

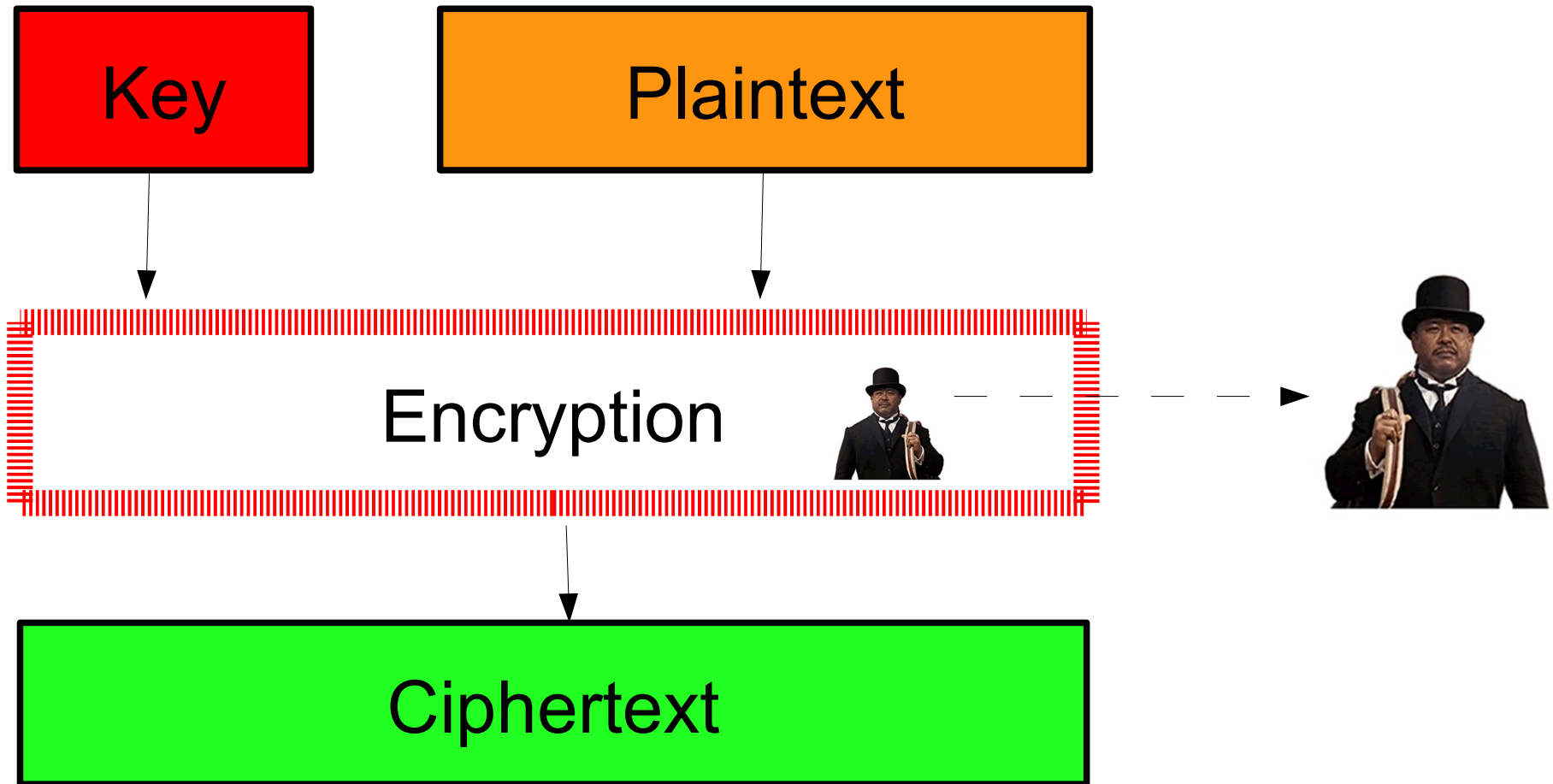


MORE

Hardware attacks: breaking the box open



Covert channels: attack code running from within



Attacks against RSA

RSA: The original public key algorithm



RSA: The original public key algorithm

Private Key: p, q (random primes)
 $d \equiv e^{-1} \pmod{\phi(N)}$ (exponent)

Public Key: $N = p \cdot q$ (modulus)
 e (exponent)

Encryption: $c = m^e \pmod{N}$

Verification: $m = c^d \pmod{N}$

Signing: $s = m^d \pmod{N}$

Verification: $m = s^e \pmod{N}$

RSA is implemented via square-and-multiply

$$b^{65553} \pmod{N}$$

$$\equiv b^{0x10011} \pmod{N}$$

$$\equiv b^{(10000000000010001)b} \pmod{N}$$

$$\equiv b^{65536} \cdot b^{16} \cdot b^1 \pmod{N}$$

n digit exponentiation requires **log *n*** squares,
up to **log *n*** multiplications

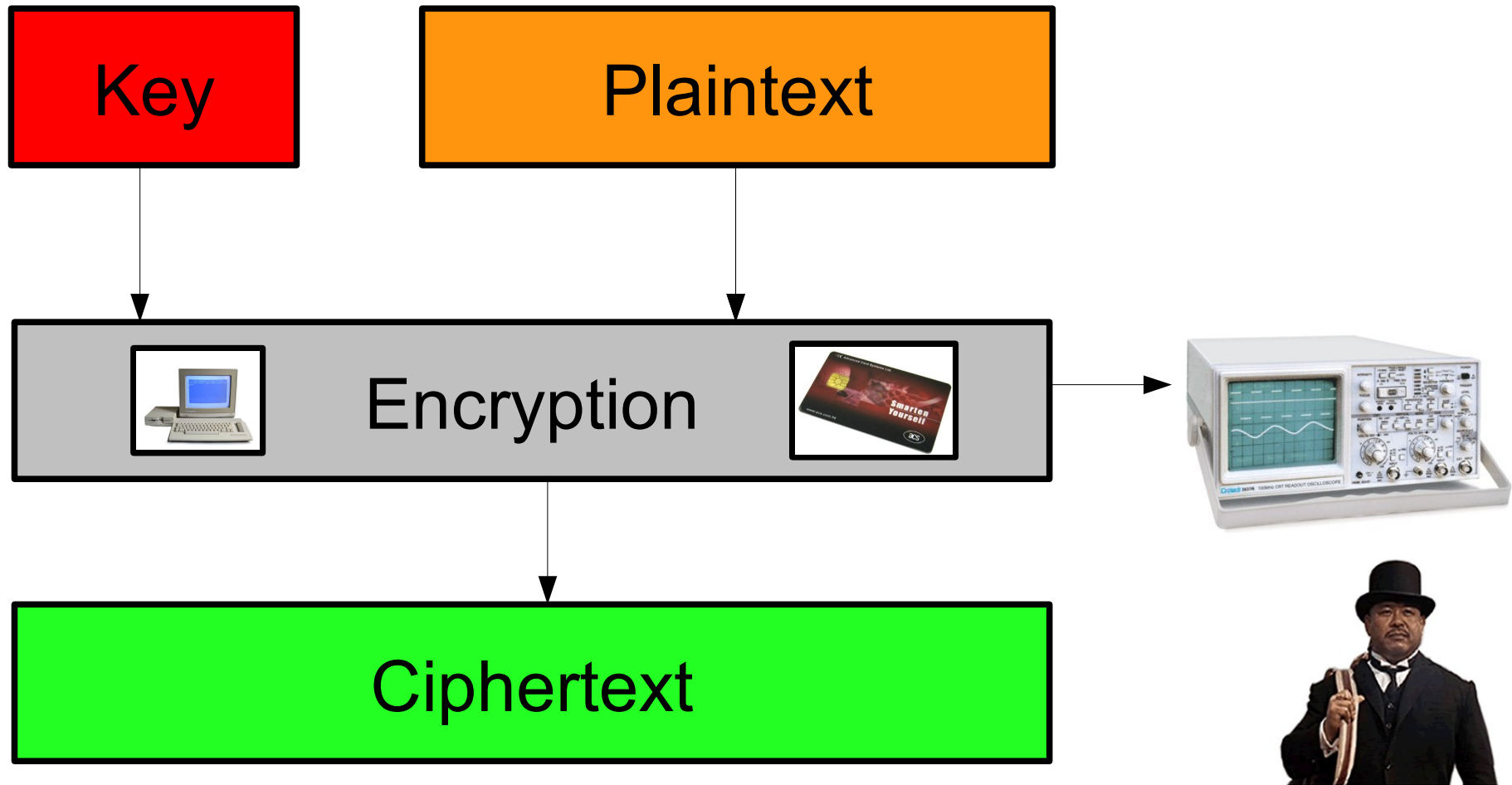
RSA is implemented via square-and-multiply

```
def power(b, e, N):  
    result = 1  
  
    for i in range(len(e) - 1, -1):  
        result = square(result, N)  
        if bit_set(e, i):  
            result = mult(result, b, N)  
  
    return result
```

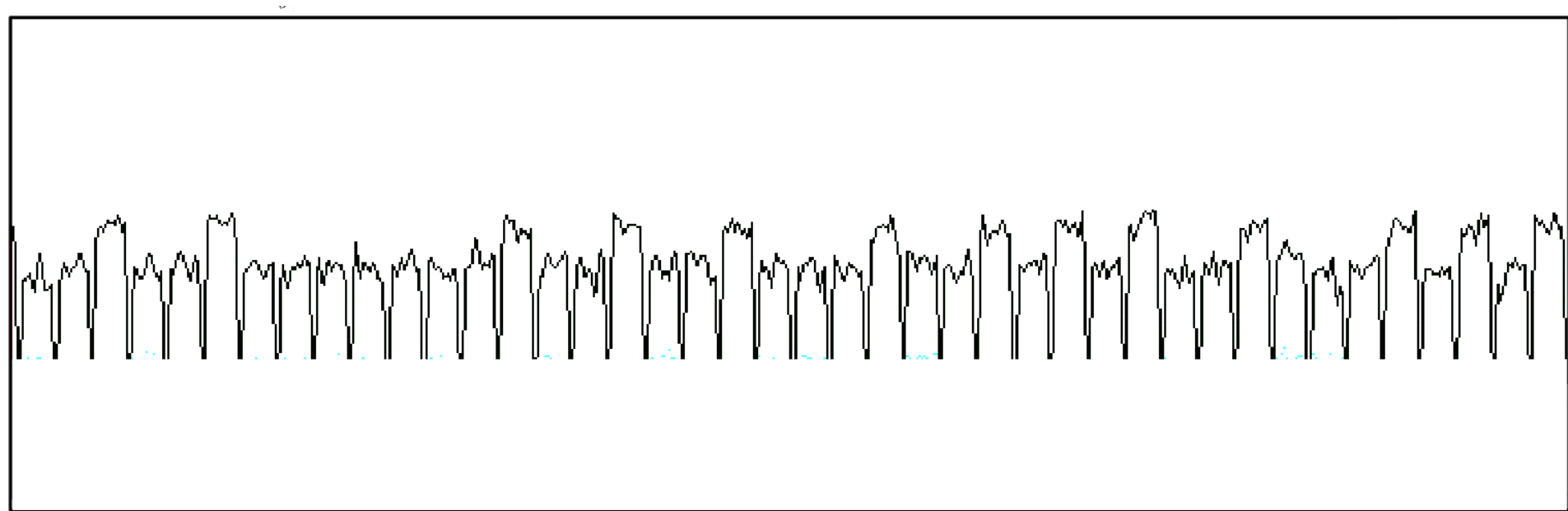
Simple power analysis of RSA

(Paul Kocher et. al 1999)

Simple power analysis setup



What does power consumption reveal?

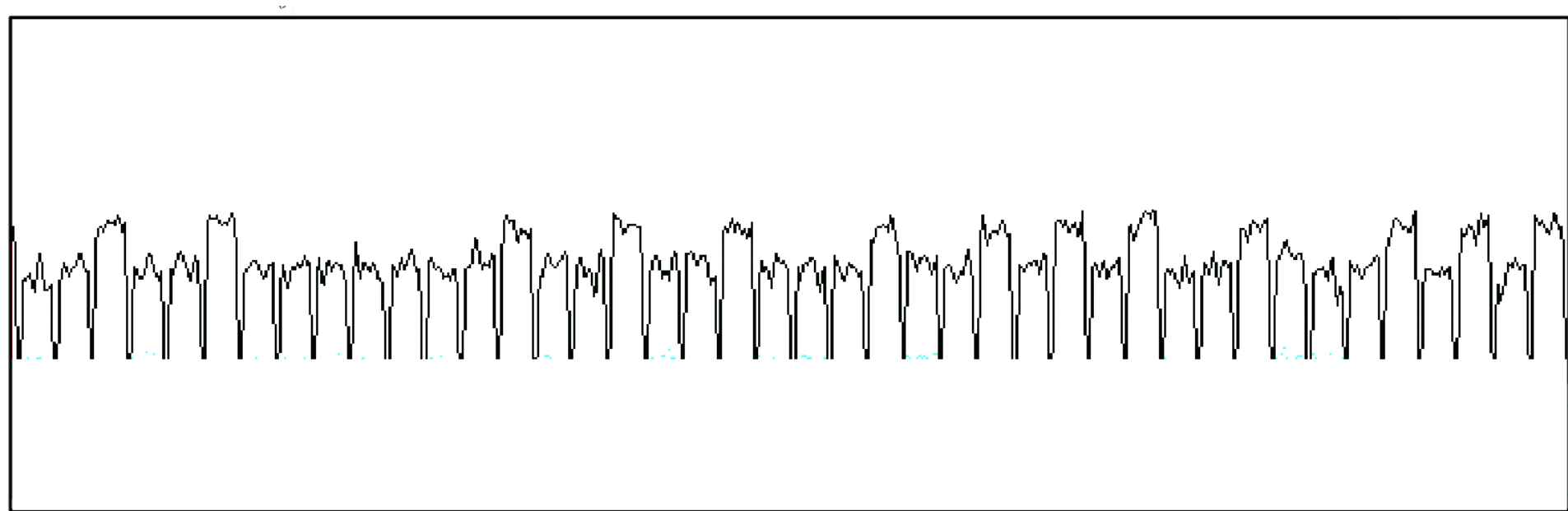


Trace courtesy of Cryptography Research, Inc.

Each set exponent bit inserts a multiplication

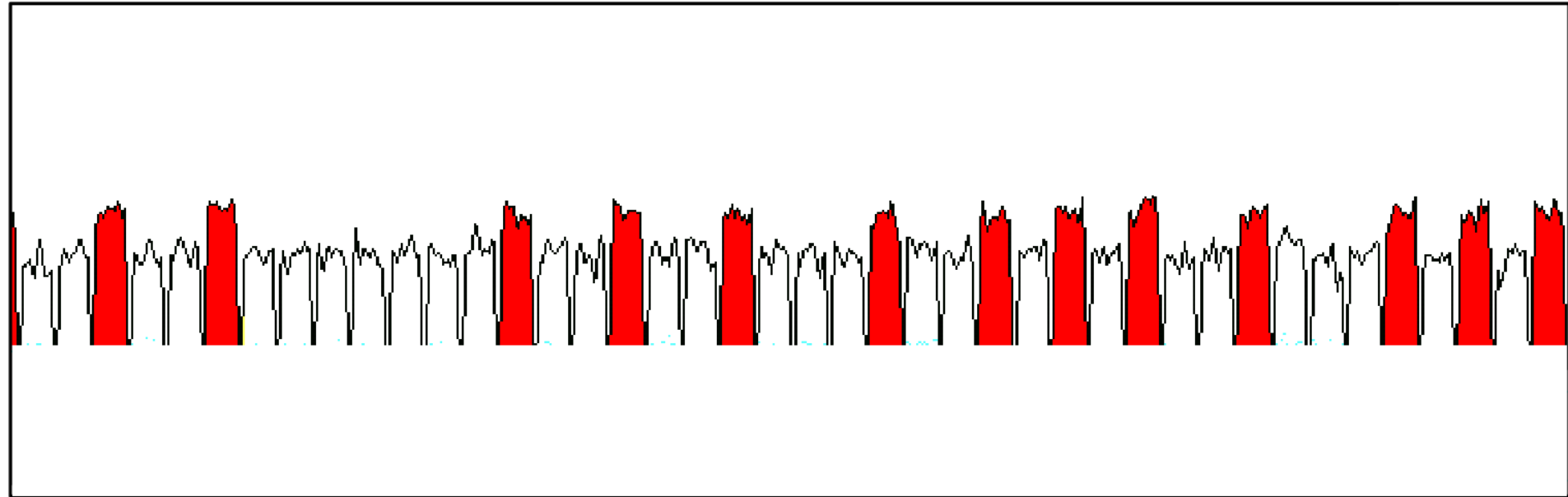
```
def power(b, e, N):  
    result = 1  
  
    for i in range(len(e) - 1, -1):  
        result = square(result, N)  
        if bit_set(e, i):  
            result = mult(result, b, N)  
  
    return result
```

Multiplies can often be visually detected



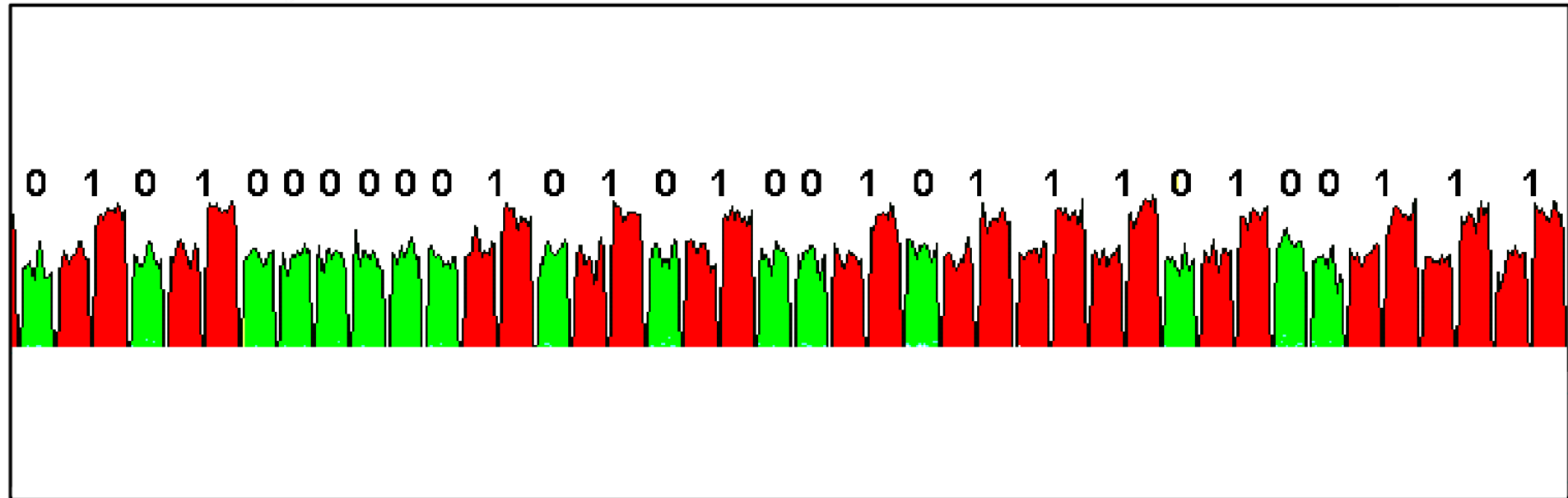
Trace courtesy of Cryptography Research, Inc.

Multiplies can often be visually detected

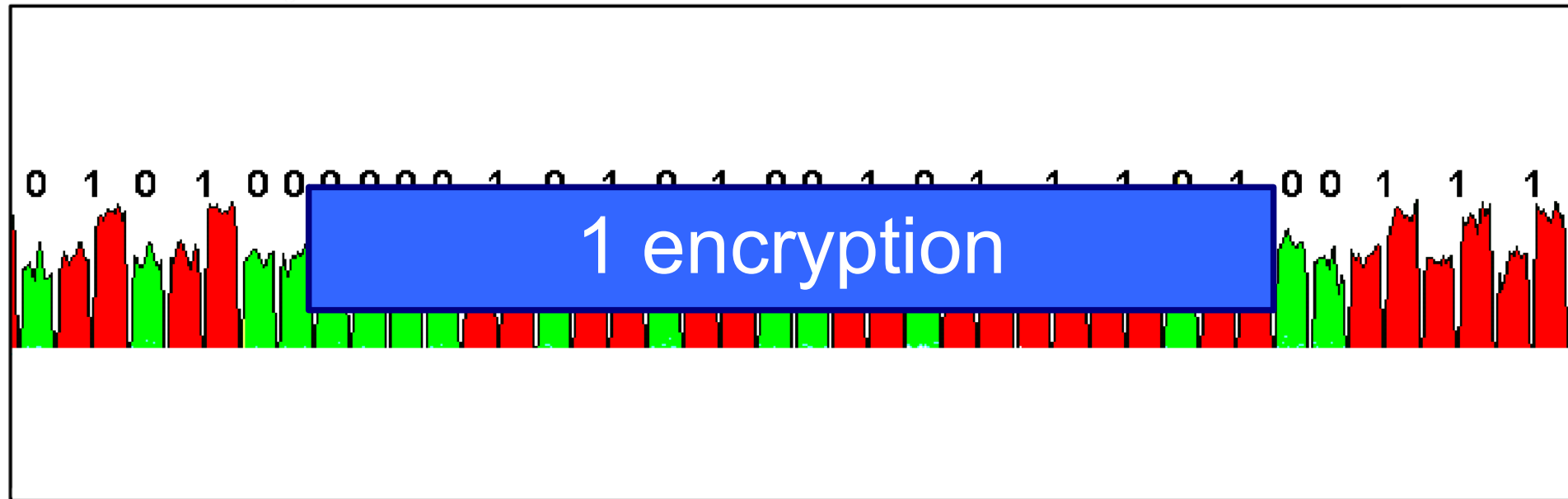


Trace courtesy of Cryptography Research, Inc.

Secret exponent can be easily read out



Secret exponent can be easily read out



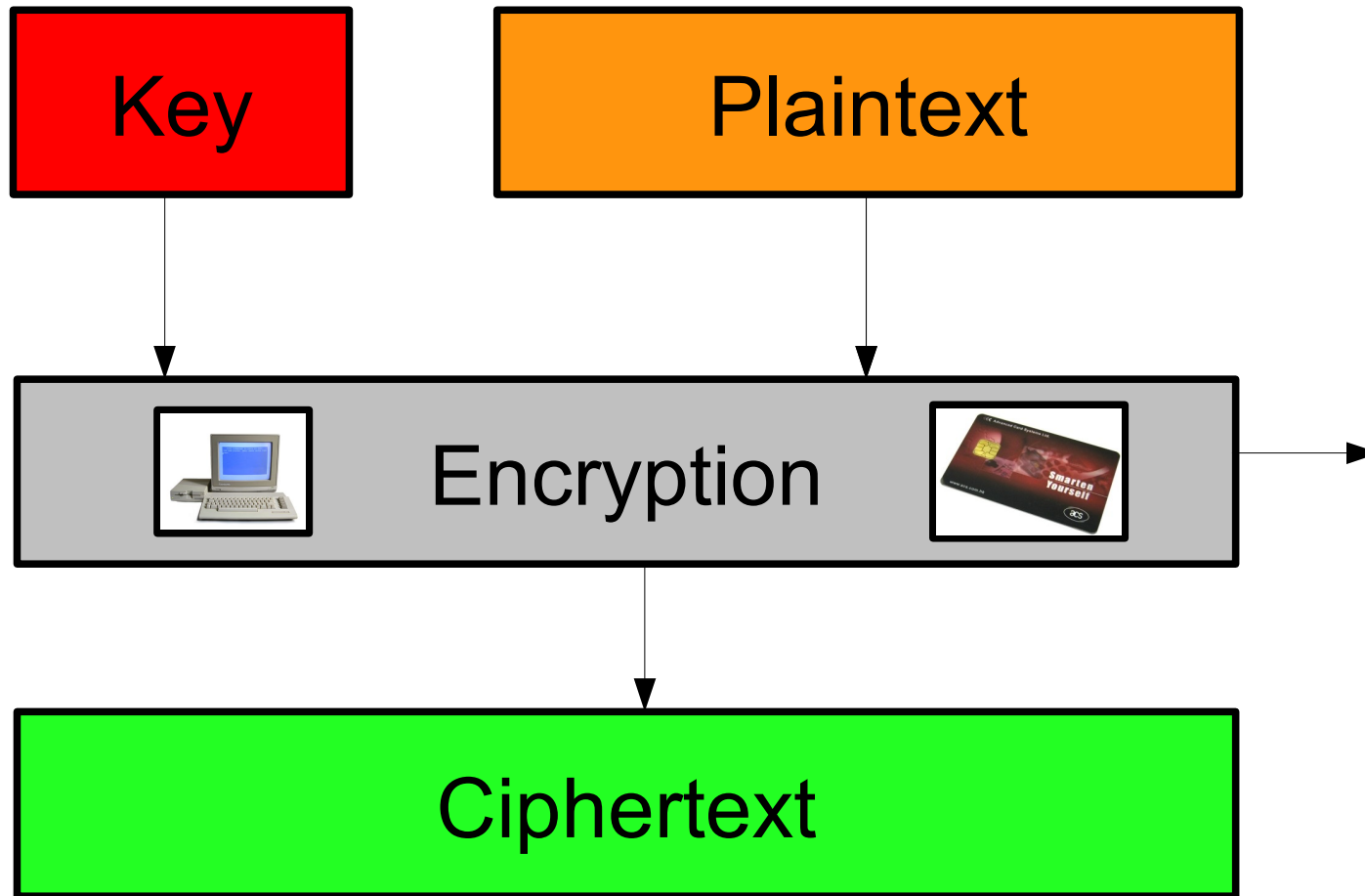
Algorithmic patch: square and **always** multiply

```
def power(b, e, N):  
    result = 1  
    b = mult(b, r, N)  
    for i in range(len(e) - 1, -1):  
        result = square(result, N)  
        if bit_set(e, i):  
            result = mult(result, b, N)  
        else:  
            result = mult(result, r, N)  
    return mult(result, r_inverse, N)
```

Timing attack against RSA

(Paul Kocher 1996)

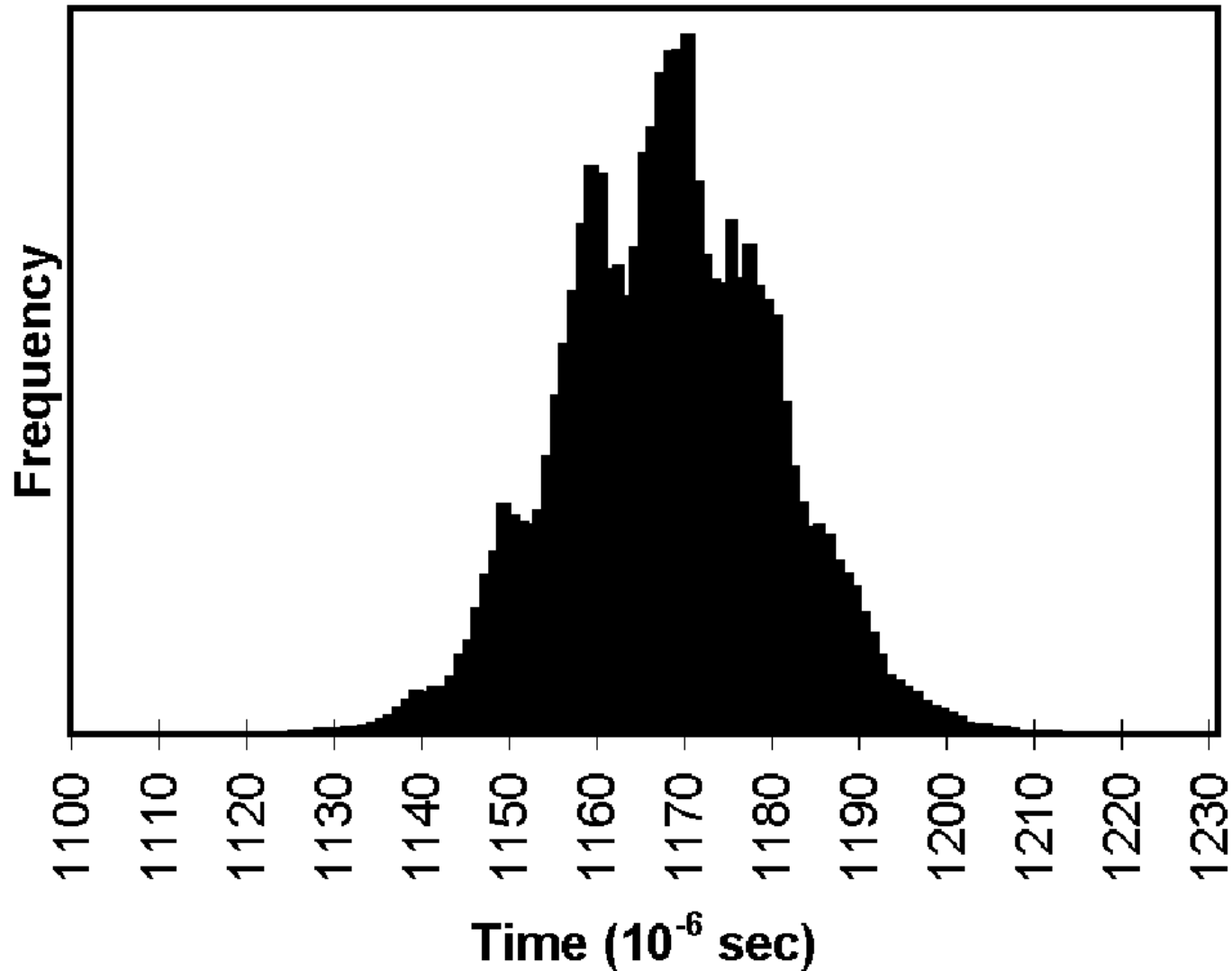
Timing attack setup



Timing leaks Hamming weight of exponent

```
def power(b, e, N):  
    result = 1  
  
    for i in range(len(e) - 1, -1):  
        result = square(result, N)  
        if bit_set(e, i):  
            result = mult(result, b, N)  
  
    return result
```

Timing of individual multiplies varies significantly



Need a model relating multiplication inputs to time

Example:

$2345 \cdot 6789 \pmod{9997}$

Multiply: $2345 \cdot 6826 = 16006970$

Reduce: $16006970 - 9997 \cdot 1 \cdot 1000$
 $= 6009970 - 9997 \cdot 6 \cdot 100$
 $= 11770 - 9997 \cdot 0 \cdot 10$
 $= 11770 - 9997 \cdot 1 \cdot 1$
 $= 1773$

Attack exponent one bit at a time

```
def power(b, e, N):  
    result = 1  
  
    for i in range(len(e) - 1, -1):  
        result = square(result, N)  
        if bit_set(e, i):  
            result = mult(result, b, N)  
  
    return result
```

Attack exponent one bit at a time

T = observed timing of entire algorithm

M = model for time of one multiplication

Bit $n-1$: always 1

Attack exponent one bit at a time

T = observed timing of entire algorithm

M = model for time of one multiplication

Bit n-2: Is **T**(power(r, e, N)) $\propto$
 M(mult($1, r, N$)) +
 M(square(r, N)) +
 M(mult(r^2, r, N)) +
 M(square(r^3, N))?

Attack exponent one bit at a time

T = observed timing of entire algorithm

M = model for time of one multiplication

Bit n-2: Is **T**(power(r, e, N)) $\propto$

~~**M**(mult($1, r, N$)) +~~

~~**M**(square(r, N)) +~~

M(mult(r^2, r, N)) +

M(square(r^3, N))?

Attack exponent one bit at a time

T = observed timing of entire algorithm

M = model for time of one multiplication

Bit n-3: Is $T(\text{power}(r, e, N)) \propto$
 $M(\text{mult}(1, r, N)) \cdot e[n-1] +$
 $M(\text{square}(r, N)) +$
 $M(\text{mult}(r^{2 \cdot e[n-1:n-1]}, r, N)) \cdot e[n-2] +$
 $M(\text{square}(r^{e[n-1:n-2]}, N))$
 $M(\text{mult}(r^{2 \cdot e[n-1:n-2]}, r, N)) +$
 $M(\text{square}(r^{e[n-1:n-2] || 1}, N))?$

Attack exponent one bit at a time

T = observed timing of entire algorithm

M = model for time of one multiplication

Bit n-3: Is $T(\text{power}(r, e, N)) \propto$

$$\begin{aligned}
 & \cancel{M(\text{mult}(1, r, N)) \cdot e[n-1]} + \\
 & \cancel{M(\text{square}(r, N))} + \\
 & \cancel{M(\text{mult}(r^{2 \cdot e[n-1:n-1]}, r, N)) \cdot e[n-2]} + \\
 & \cancel{M(\text{square}(r^{e[n-1:n-2]}, r, N))} \\
 & M(\text{mult}(r^{2 \cdot e[n-1:n-2]}, r, N)) + \\
 & M(\text{square}(r^{e[n-1:n-2] || 1}, r, N)) ?
 \end{aligned}$$

Attack exponent one bit at a time

T = observed timing of entire algorithm

M = model for time of one multiplication

Bit $n-i$: Is $T(\text{power}(r, e, N)) \propto$
 $M(\text{mult}(r^{2 \cdot e_{[n-1:n-i]}}, r, N)) +$
 $M(\text{square}(r^{e_{[n-1:n-i]} || 1}}, N))?$

Attack exponent one bit at a time

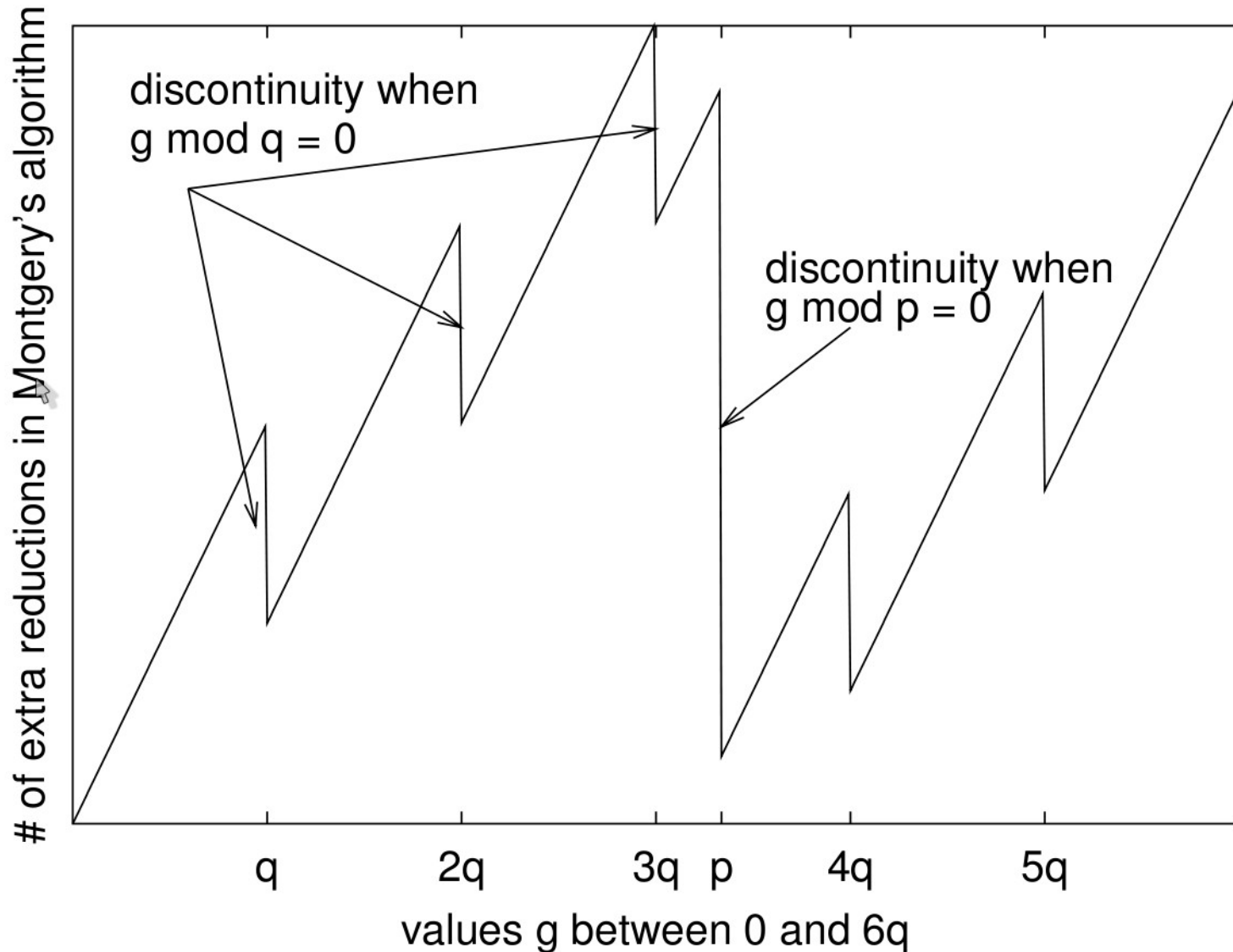
T = observed timing of entire algorithm

M = model for time of one multiplication

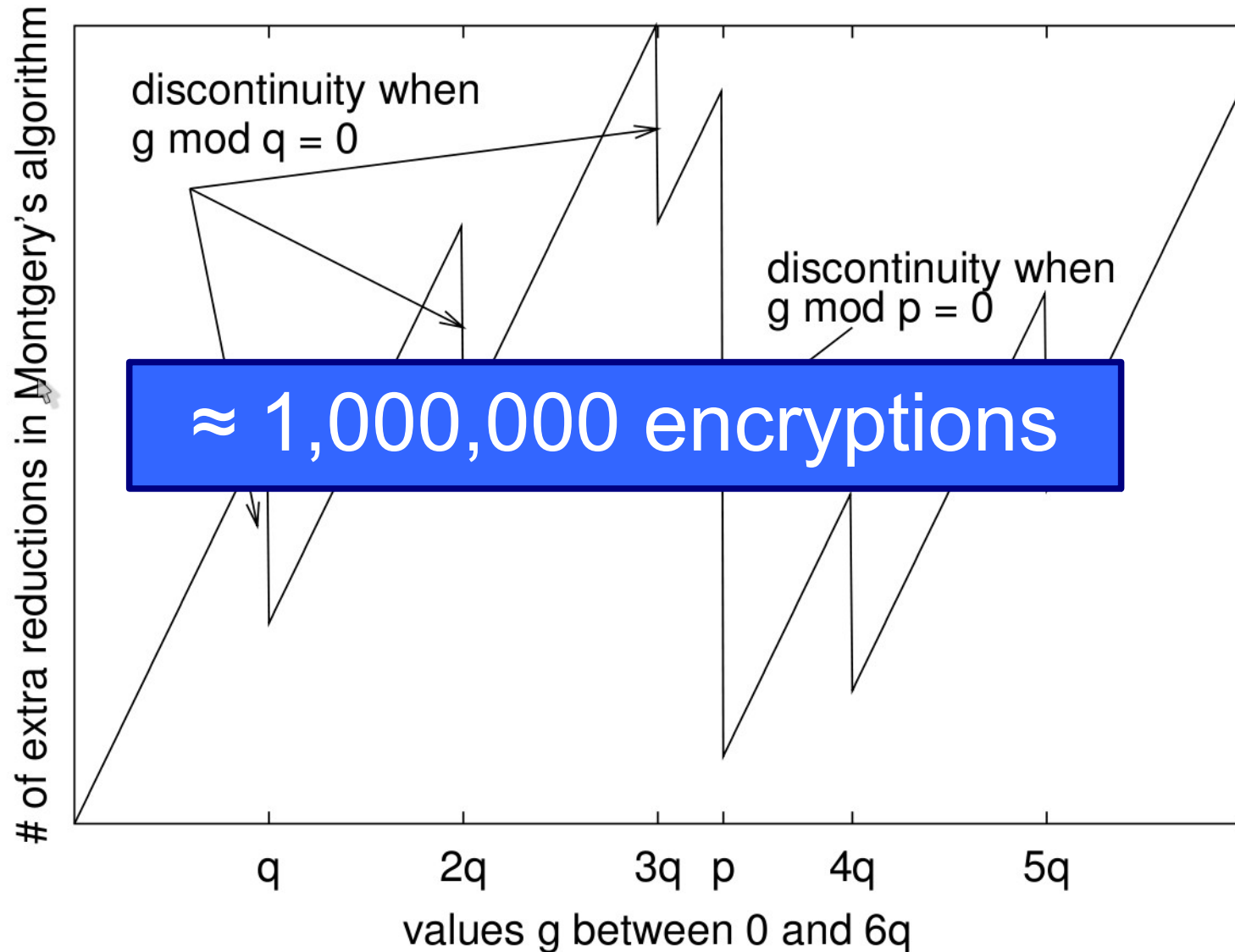
Bit $n-i$: Is $T(\text{power}(r, e, N)) \propto$

$\approx 2,500$ encryptions

More complicated attacks work across a LAN



More complicated attacks work across a LAN



Blinded RSA provides generic defense

Private Key: p, q (random primes)
 $d \equiv e^{-1} \pmod{\phi(N)}$ (exponent)

Public Key: $N = p \cdot q$ (modulus)
 e (exponent)

Signing: $s = m^d \pmod{N}$

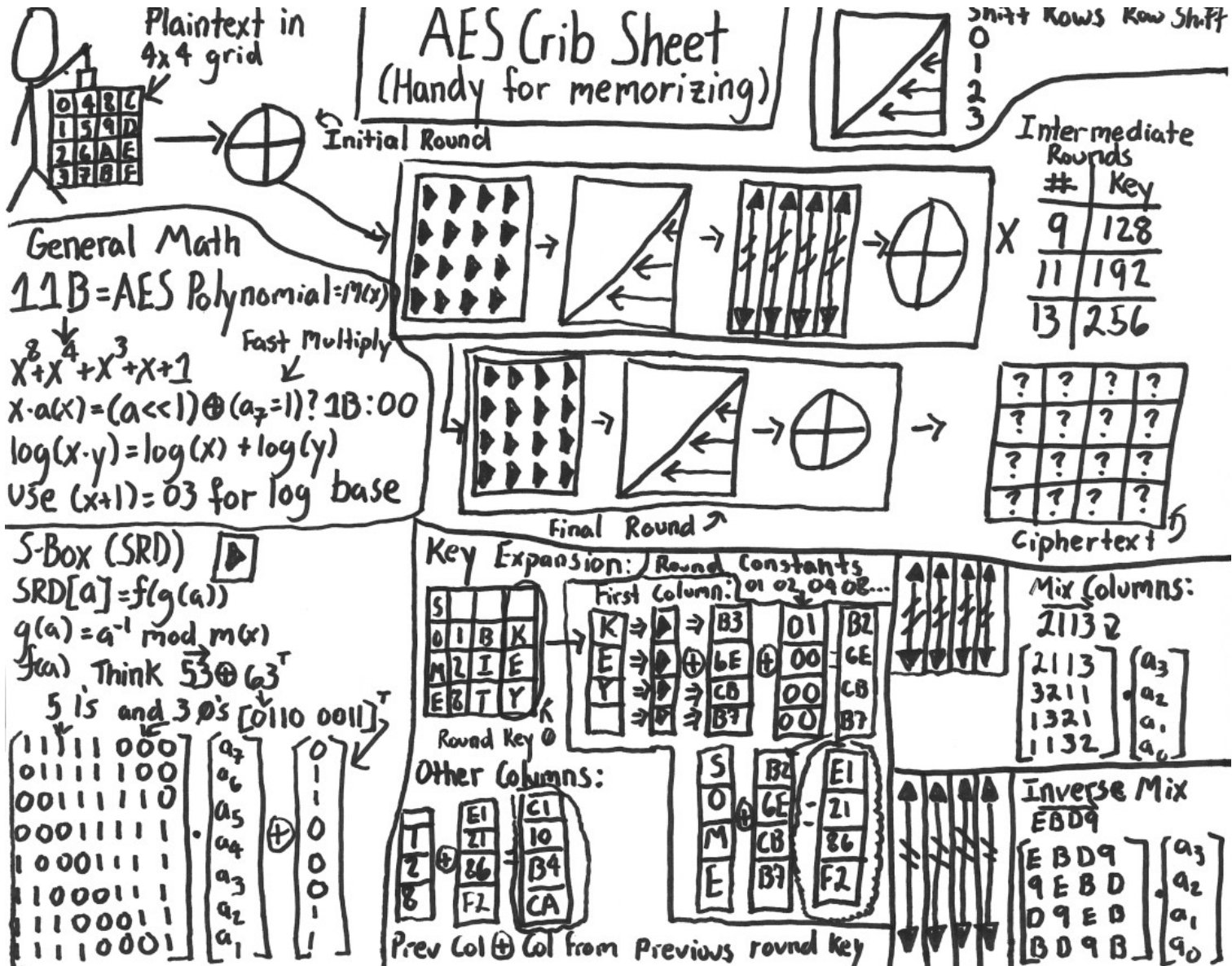
Blind Signing: $r_1 = r_0^e \pmod{N}$
 $s = r_0^{-1} (r_1 \cdot m)^d \pmod{N}$

Attacks against AES (aka Rijndael)

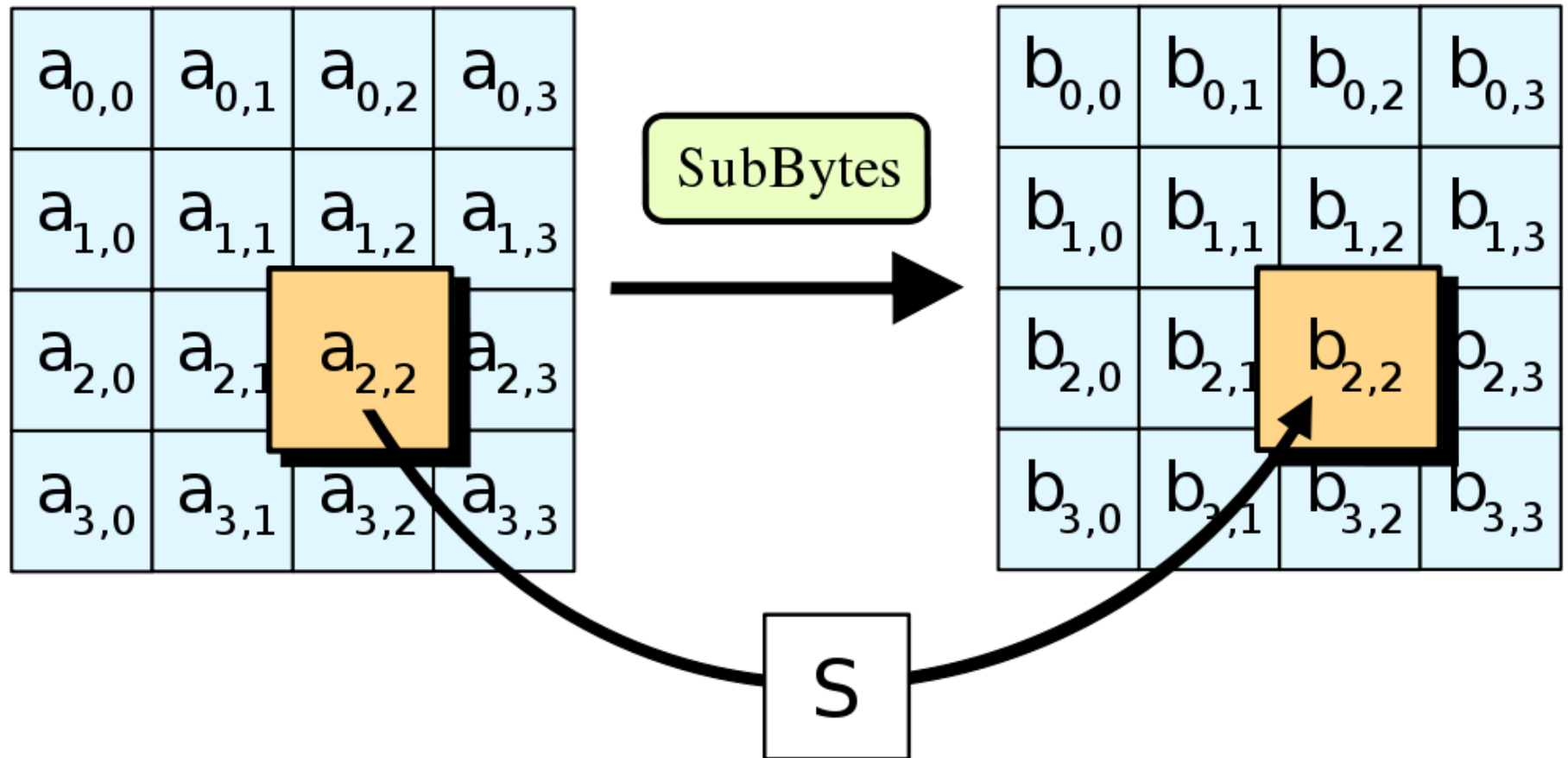
AES is cryptography's standard block cipher



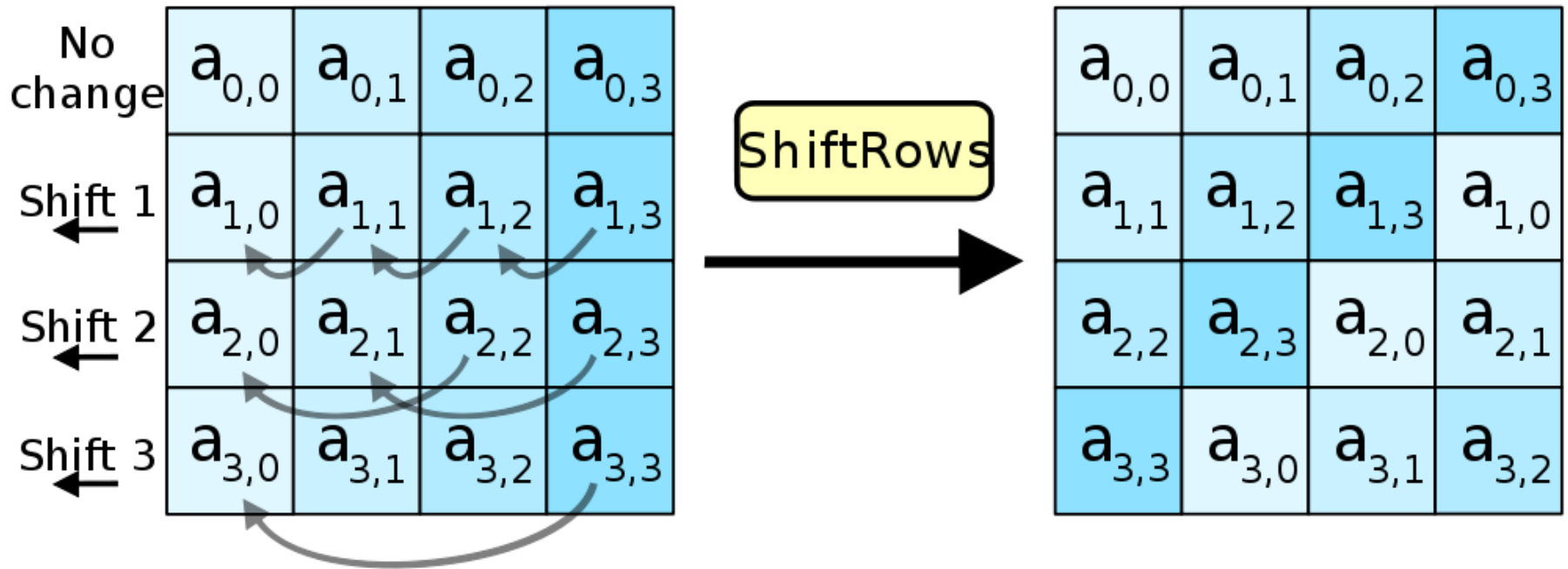
AES is very complicated



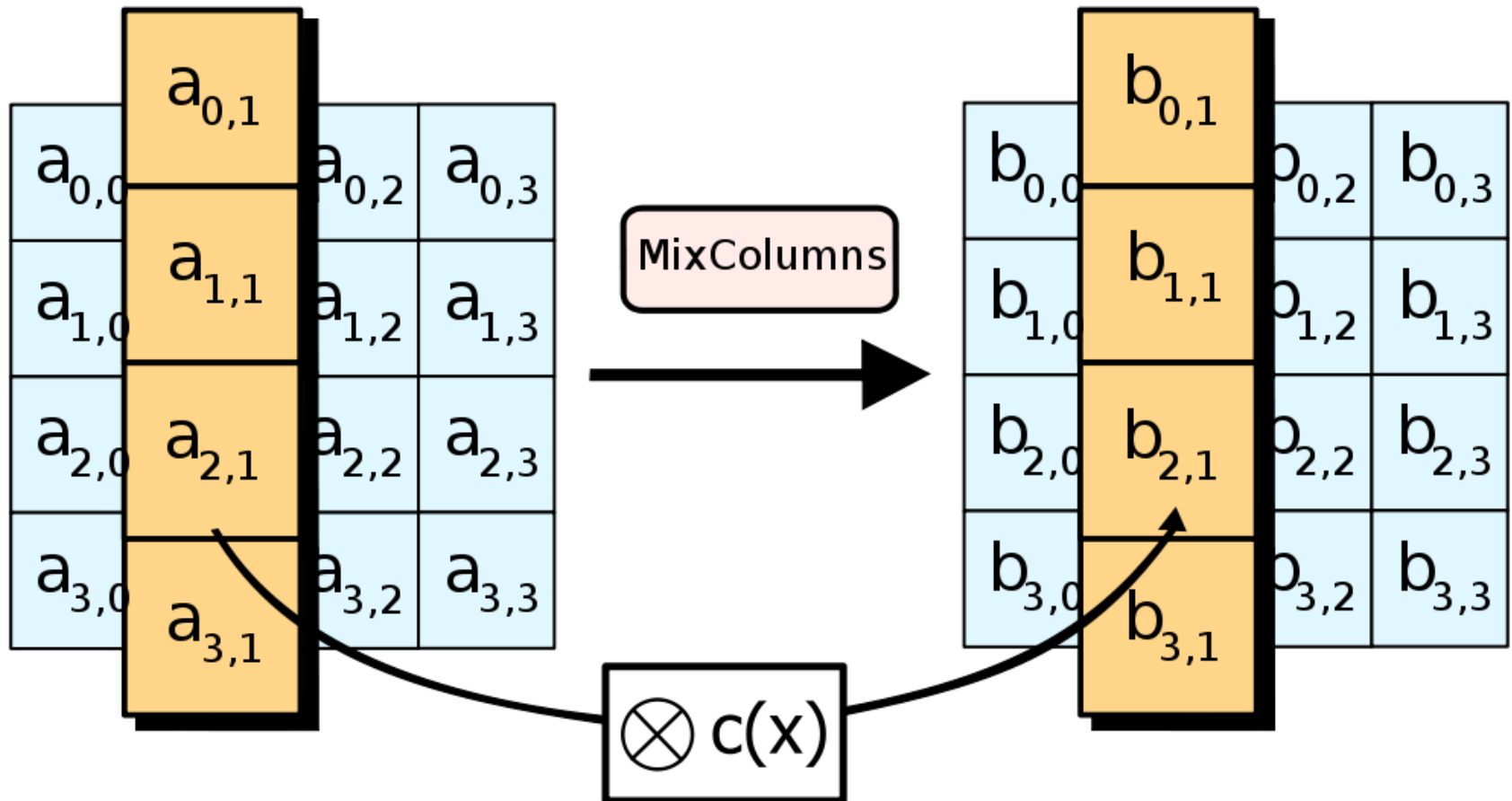
AES is very complicated



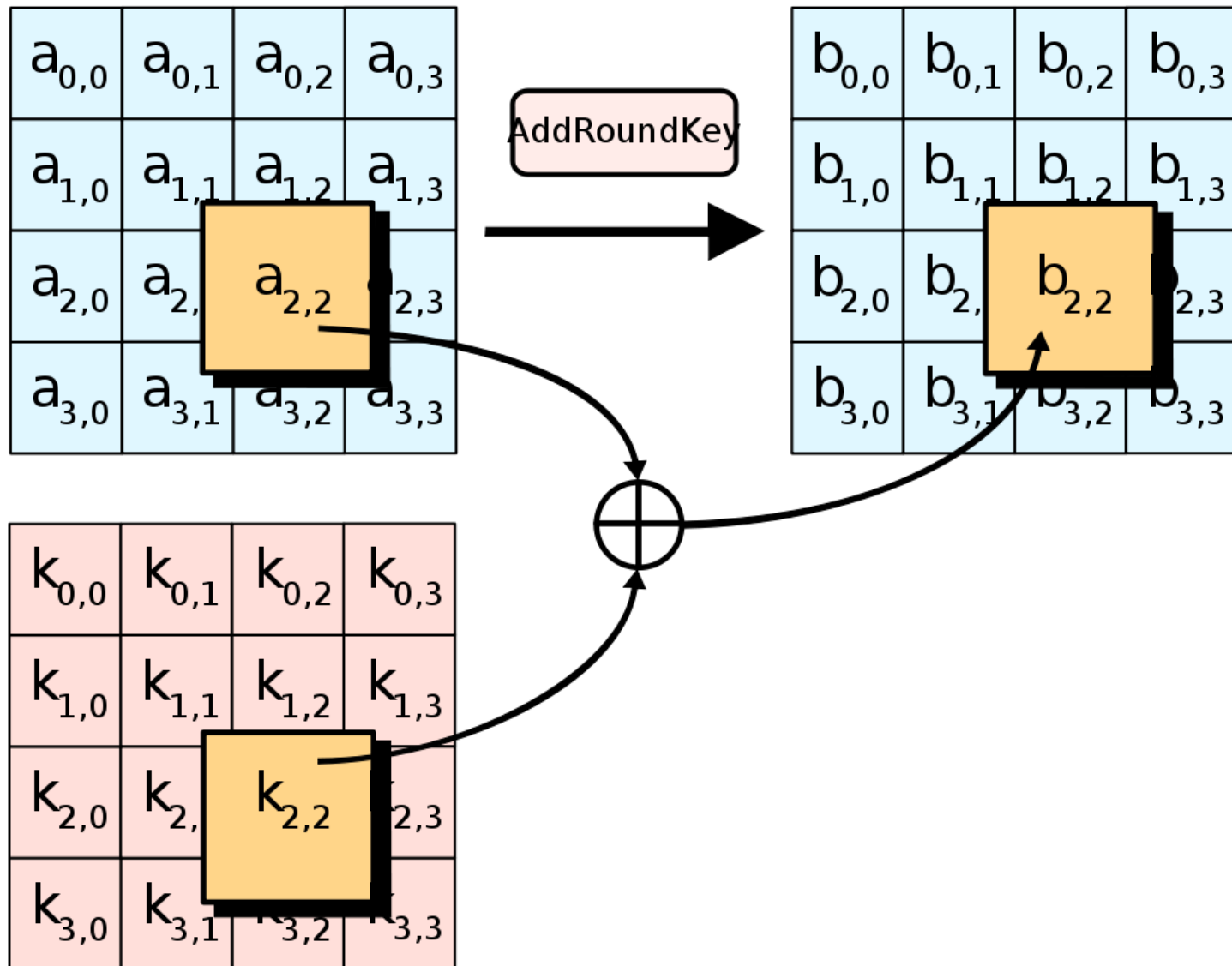
AES is very complicated



AES is very complicated



AES is very complicated



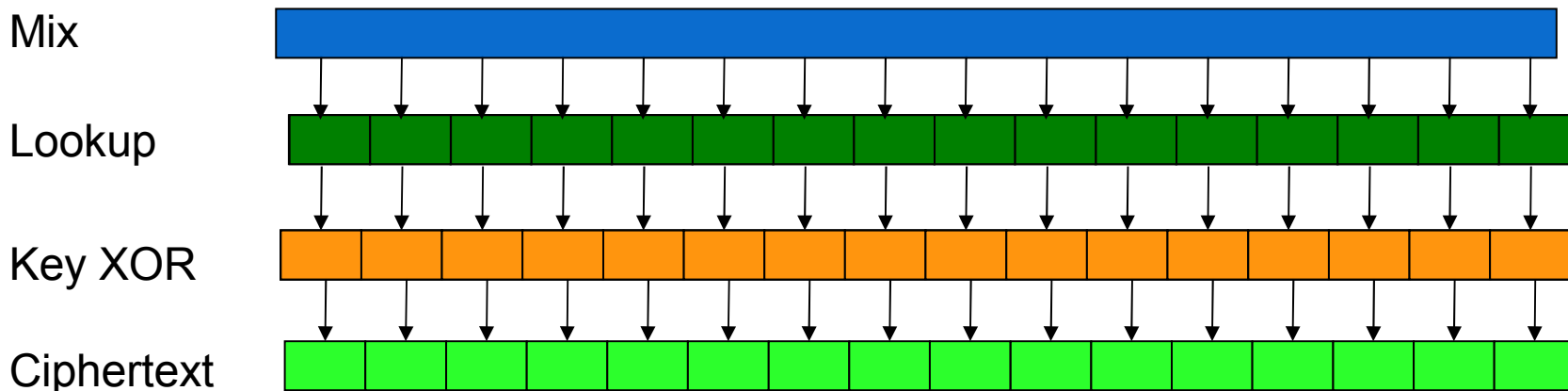
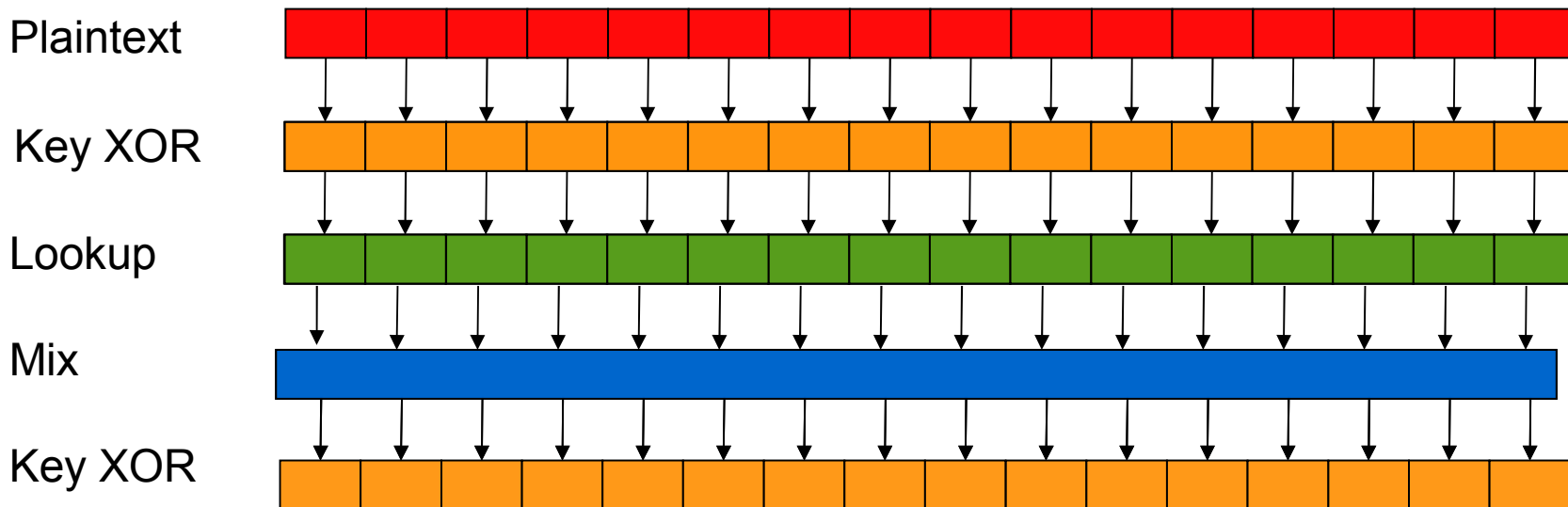
AES is designed for very efficient implementation

```
t0 = Te0[(s0 >> 24)      ] ^  
      Te1[(s1 >> 16) & 0xff] ^  
      Te2[(s2 >>  8) & 0xff] ^  
      Te3[(s3          ) & 0xff] ^  
      rk[0];  
t1 = Te0[(s1 >> 24)      ] ^  
      Te1[(s2 >> 16) & 0xff] ^  
      Te2[(s3 >>  8) & 0xff] ^  
      Te3[(s0          ) & 0xff] ^  
      rk[1];  
t2 = Te0[(s2 >> 24)      ] ^  
      Te1[(s3 >> 16) & 0xff] ^  
      Te2[(s0 >>  8) & 0xff] ^  
      Te3[(s1          ) & 0xff] ^  
      rk[2];  
t3 = Te0[(s3 >> 24)      ] ^  
      Te1[(s0 >> 16) & 0xff] ^  
      Te2[(s1 >>  8) & 0xff] ^  
      Te3[(s2          ) & 0xff] ^  
      rk[3];
```

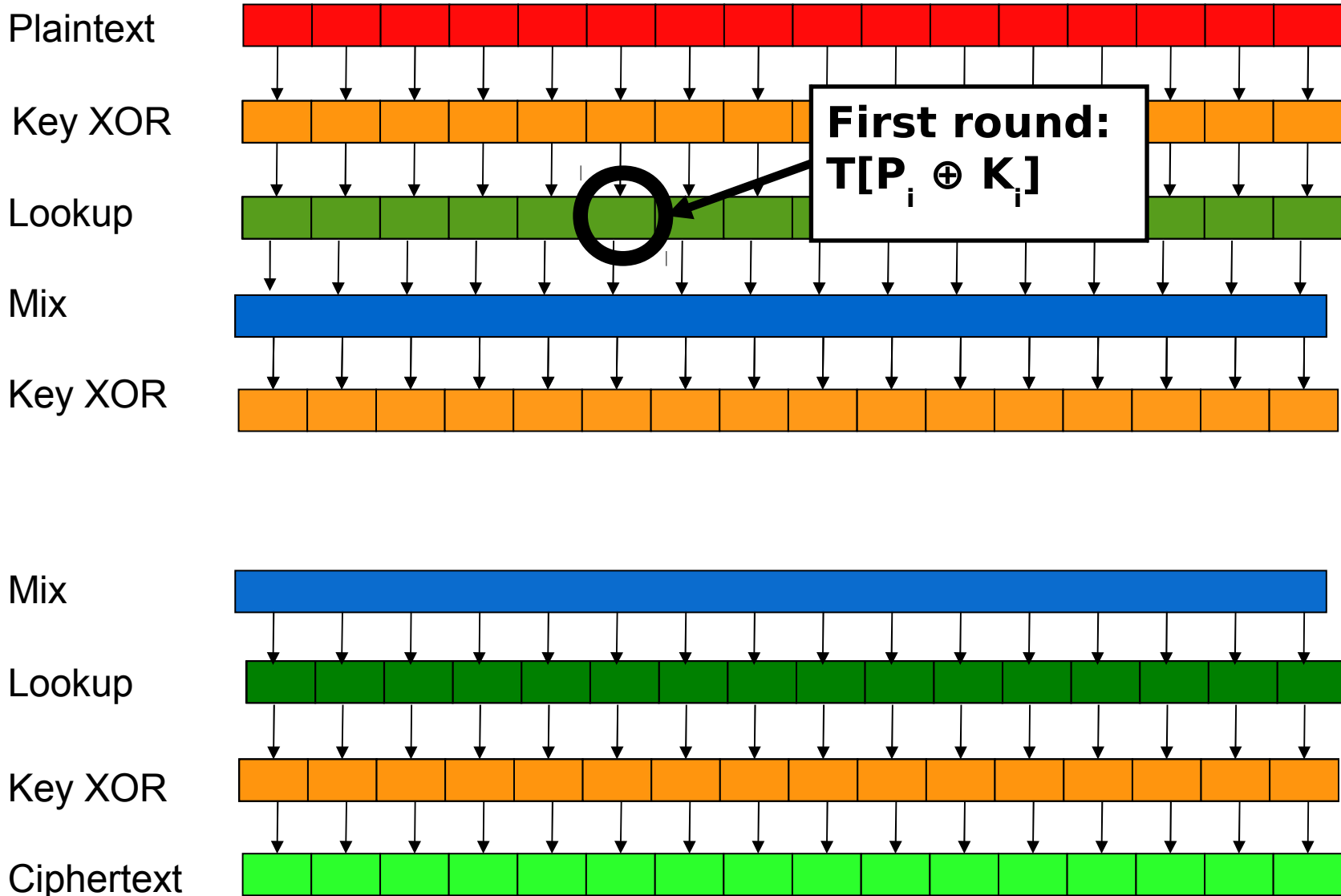
AES utilises large pre-computed lookup tables

```
static const u32 Te0[256] =  
{  
    0xc66363a5U, 0xf87c7c84U, 0xee777799U, 0xf67b7b8dU,  
    0xffff2f20dU, 0xd66b6bbdU, 0xde6f6fb1U, 0x91c5c554U,  
    0x60303050U, 0x02010103U, 0xce6767a9U, 0x562b2b7dU,  
    0xe7fefe19U, 0xb5d7d762U, 0x4dababe6U, 0xec76769aU,  
    . . .  
    0x824141c3U, 0x299999b0U, 0x5a2d2d77U, 0x1e0f0f11U,  
    0x7bb0b0cbU, 0xa85454fcU, 0x6dbbbbd6U, 0x2c16163aU,  
};
```

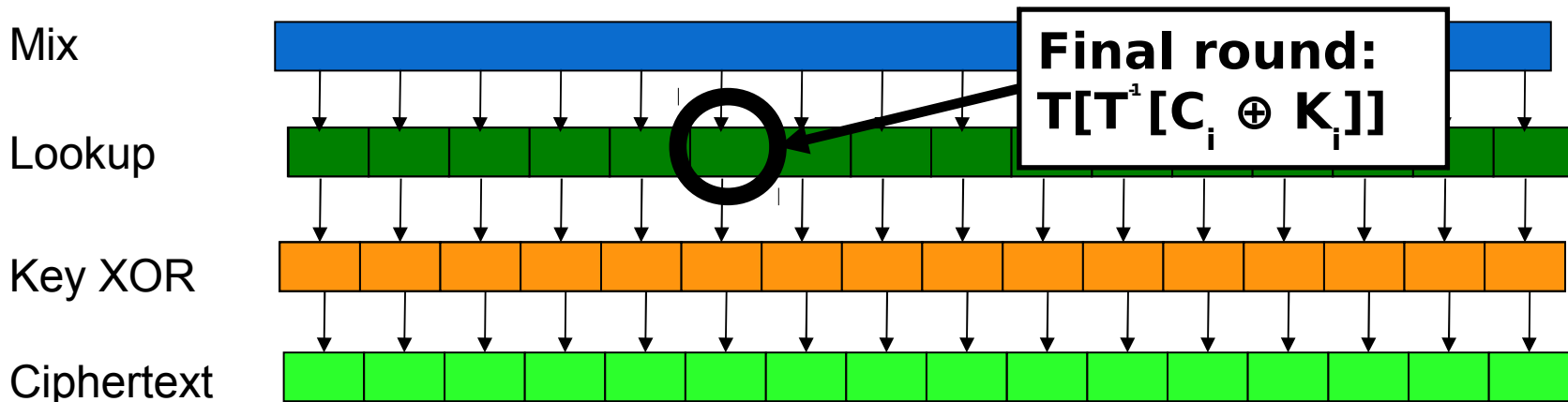
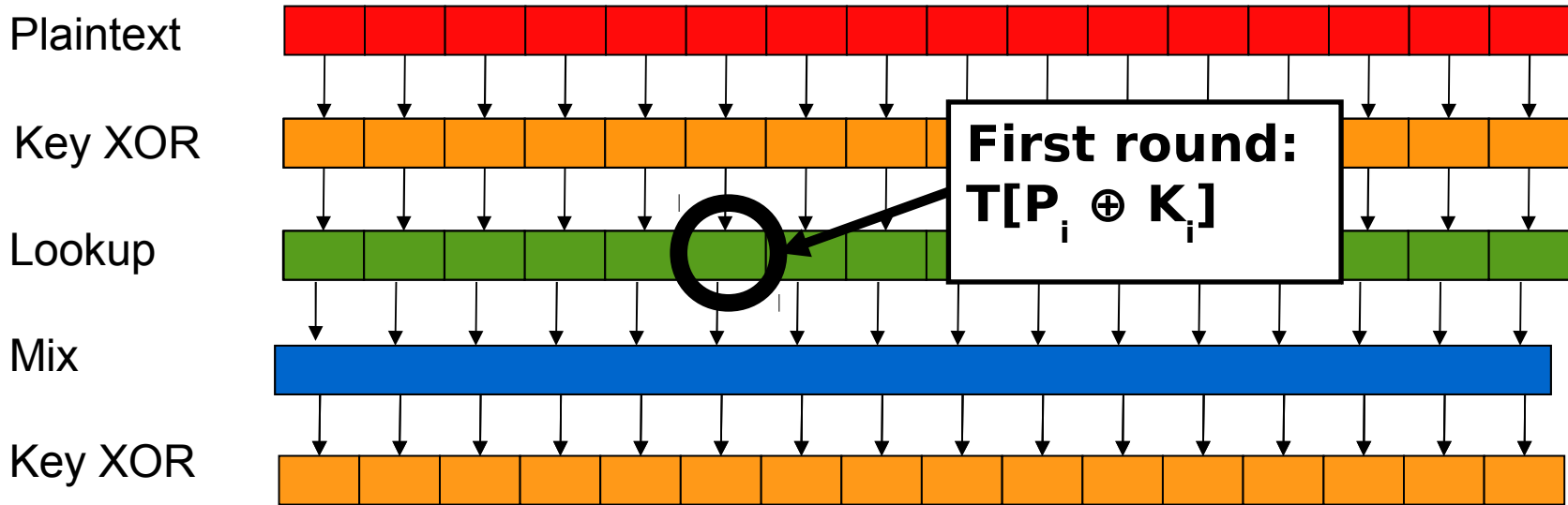
Lookups into shared cache are vulnerable



Lookups into shared cache are vulnerable



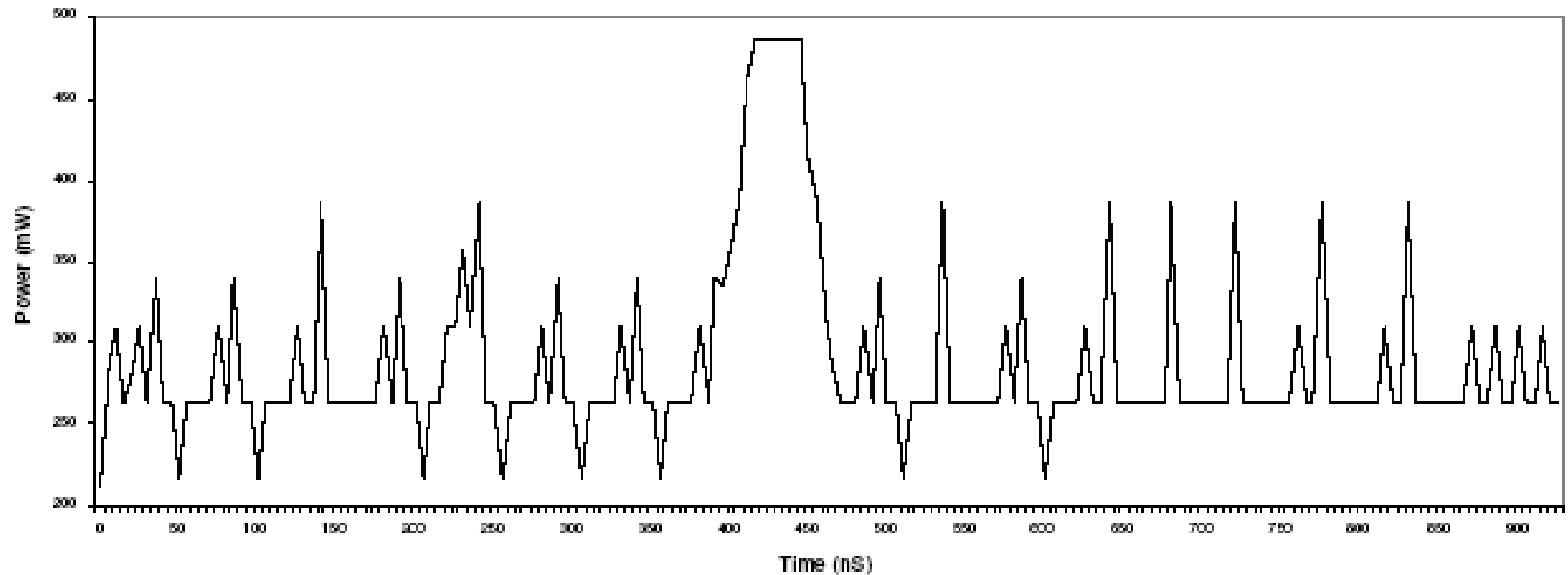
Lookups into shared cache are vulnerable



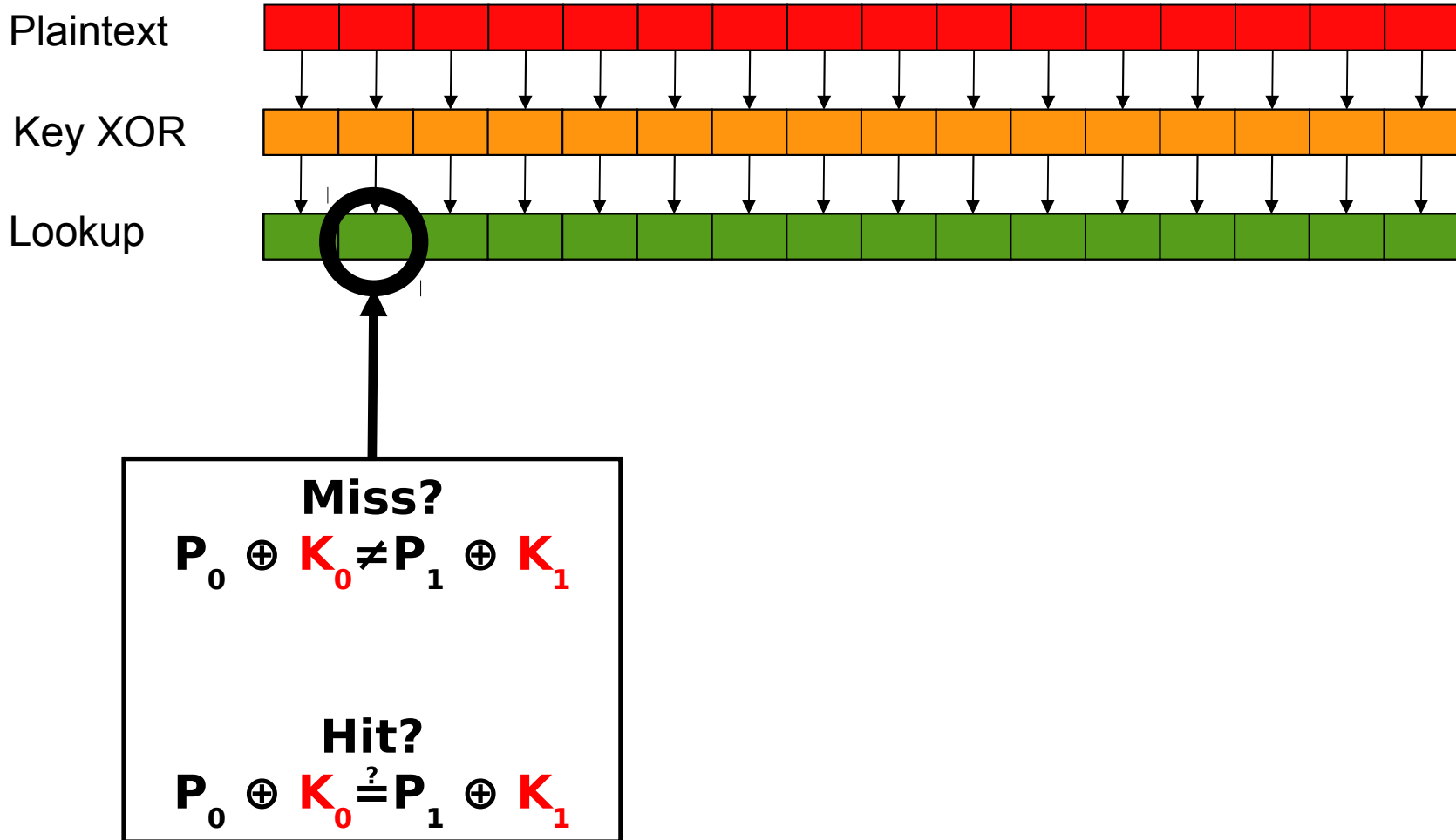
Simple power analysis of AES

(Bertoni et. Al, 2005;
Bonneau 2006)

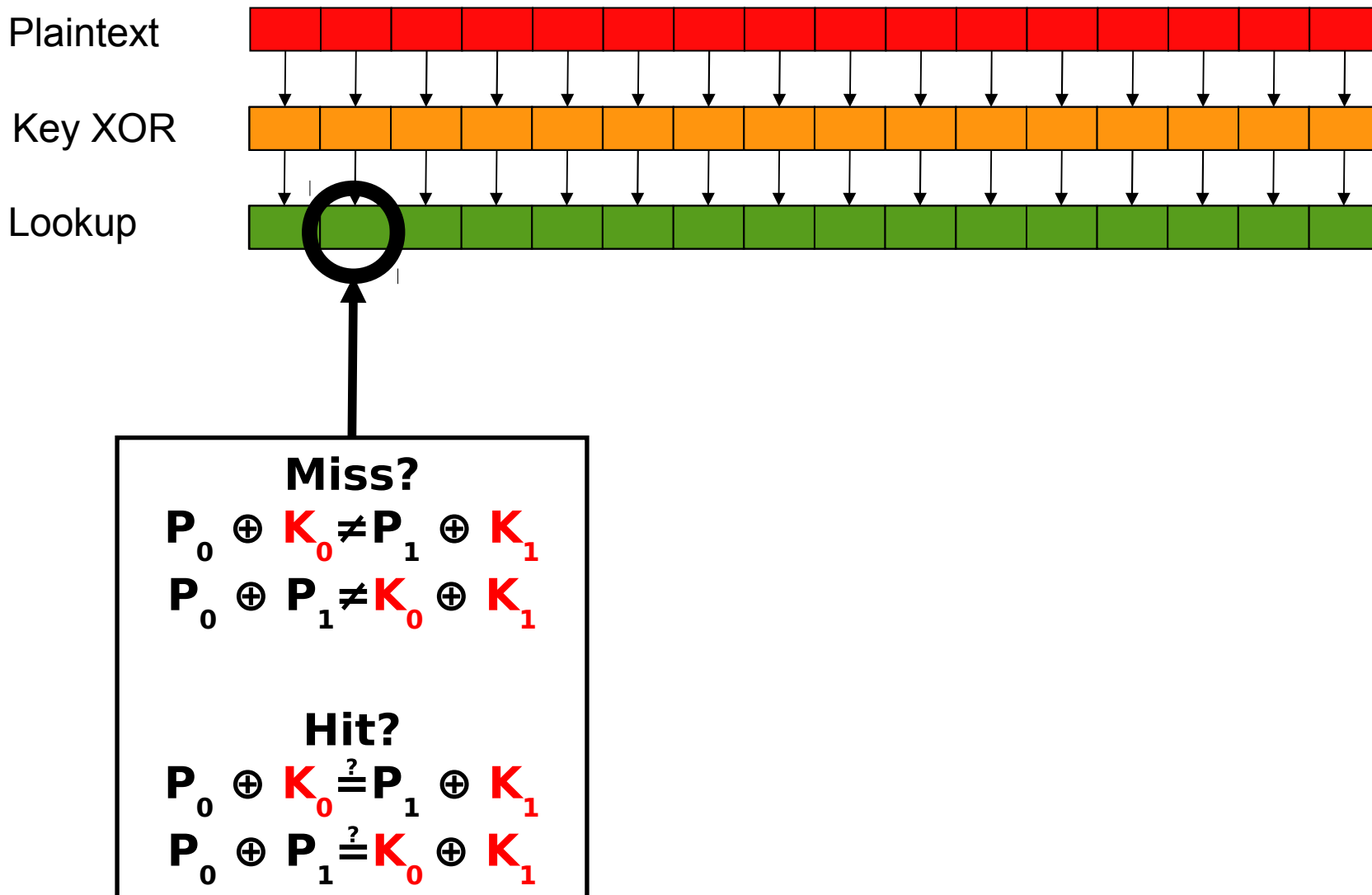
Cache hit/miss is very obvious in power trace



Every miss yields many constraints



Every miss yields many constraints



Every miss yields many constraints

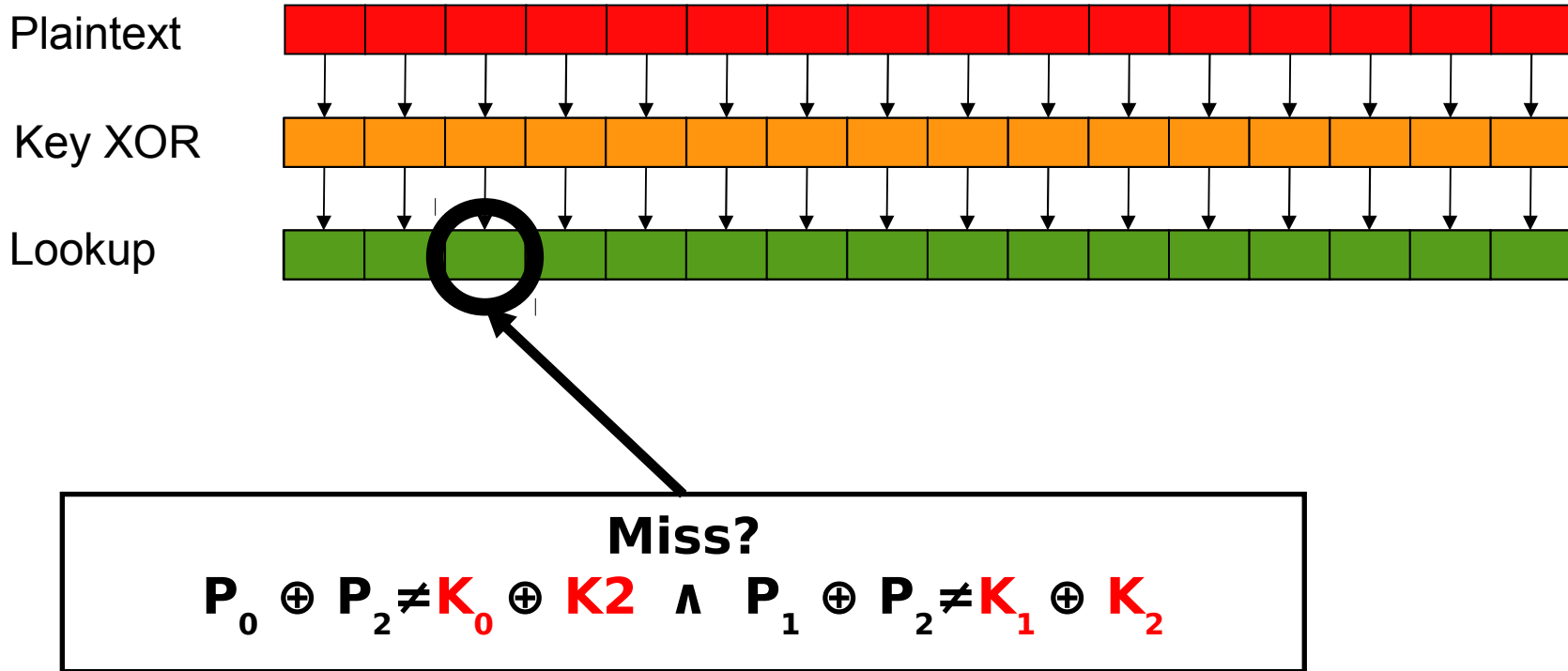


Table of possible key byte differences refined

	K0	K1	K2	...	K15
K0	00	{27,e0}	{35}	{23,70,c 4}	{65}
K1		00	{32,45,8 9}	{5f,f3}	{0a,db}
K2			00	{86}	{17,64,9 c}
...				00	{42,d5}
K15					00

Table of possible key byte differences refined

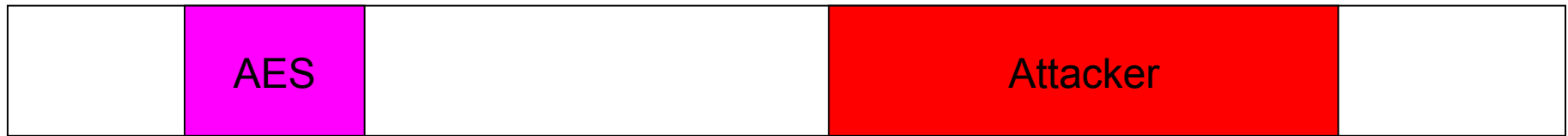
	K0	K1	K2	...	K15
K0	00	{27, 00}	{35}	{23, 70, c}	{65}
K1			9}		{0a, db}
K2			00	{86}	{17, 64, 9 c}
...				00	{42, d5}
K15					00

≈ 100 encryptions

Cache observation attack

(Osvik et. al, 2006)

1) Attacker “primes” the cache with known data



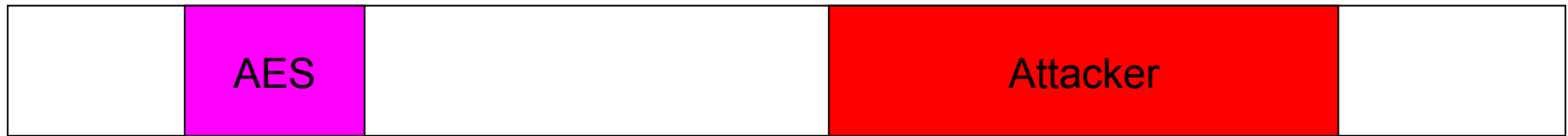
RAM



Cache

```
void * p = malloc(CACHE_SIZE);  
while(i < CACHE_SIZE)  
    p[i++]++;
```

1) Attacker “primes” the cache with known data



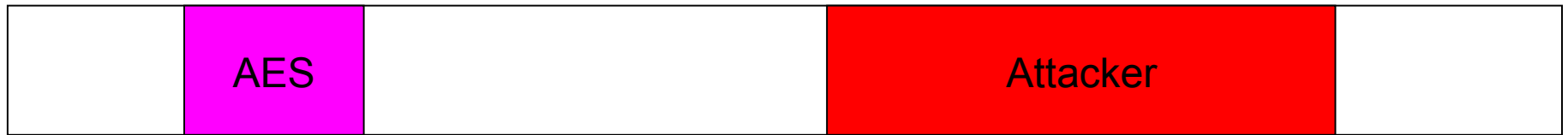
RAM



Cache

```
void * p = malloc(CACHE_SIZE);  
while(i < CACHE_SIZE)  
    p[i++]++;
```

2) Attacker triggers AES encryption



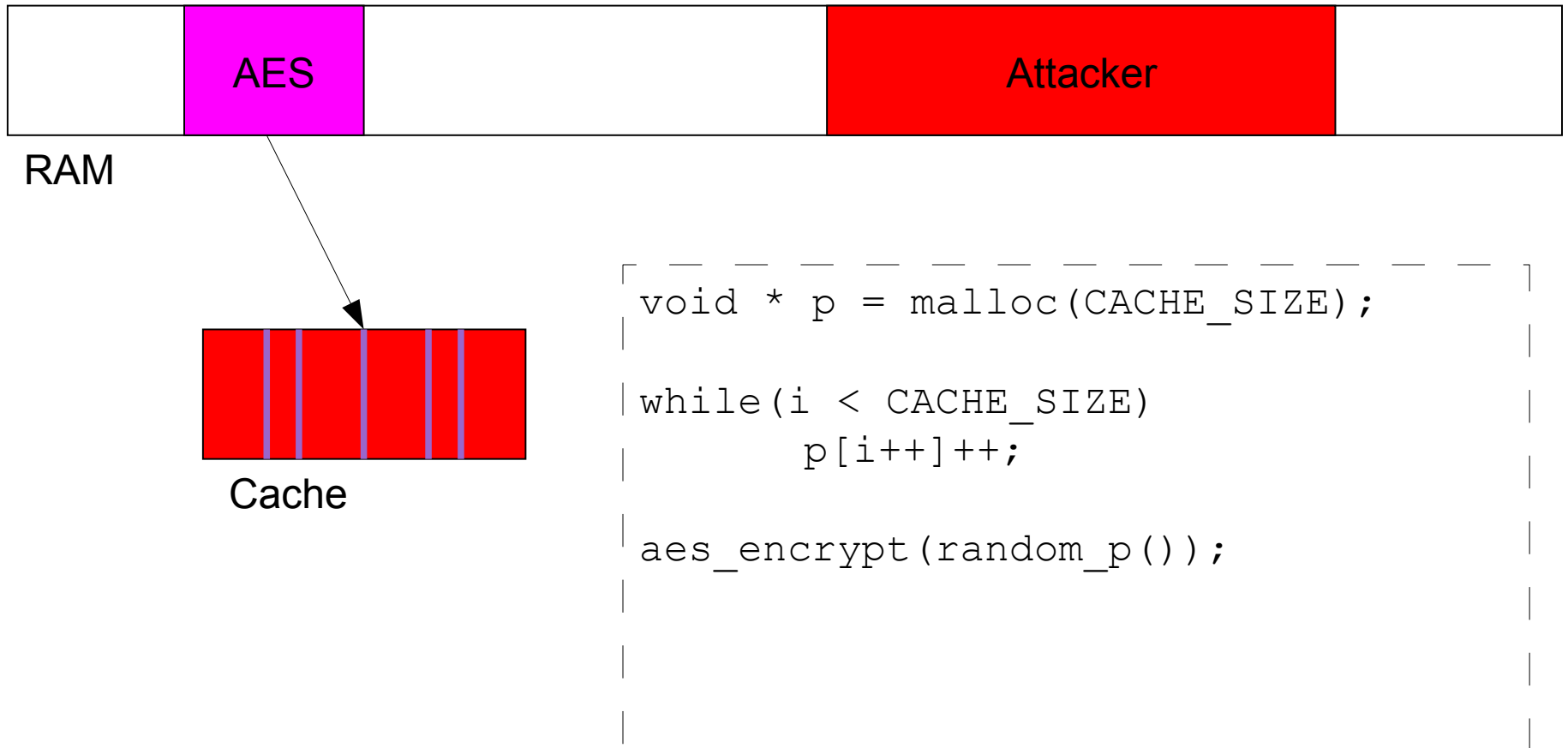
RAM



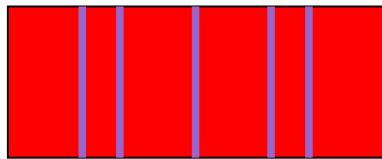
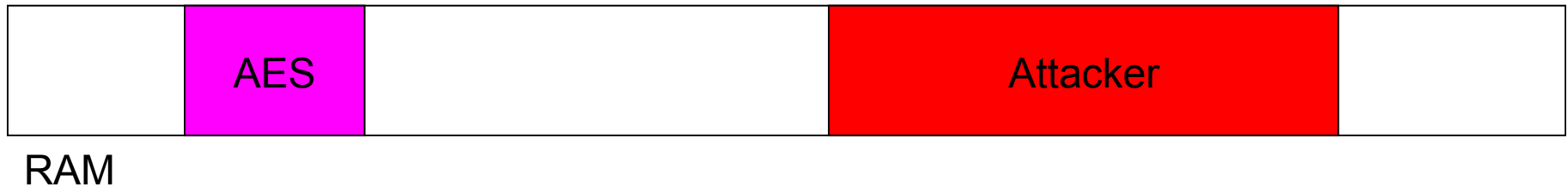
Cache

```
void * p = malloc(CACHE_SIZE);  
while(i < CACHE_SIZE)  
    p[i++]++;  
aes_encrypt(random_p());
```

3) AES loads some cache lines



4) Attacker can test which lines were touched



Cache

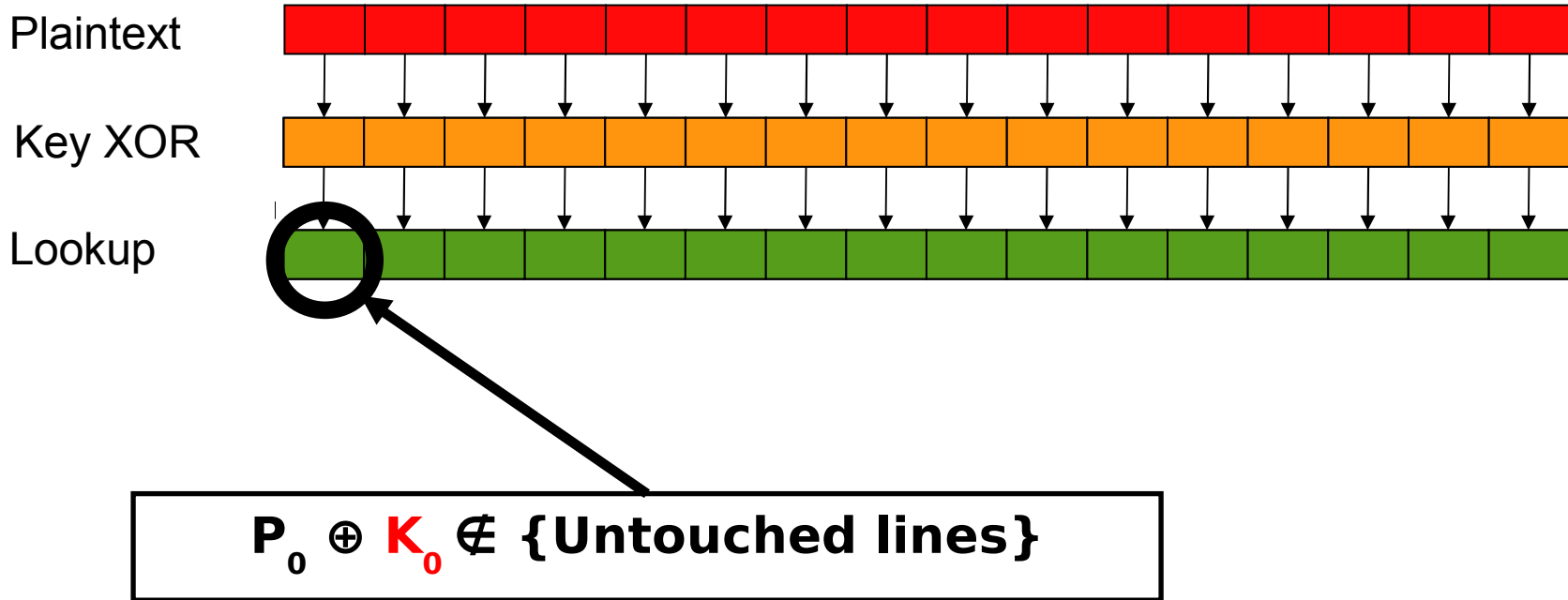
```
void * p = malloc(CACHE_SIZE);

while(i < CACHE_SIZE)
    p[i++]++;

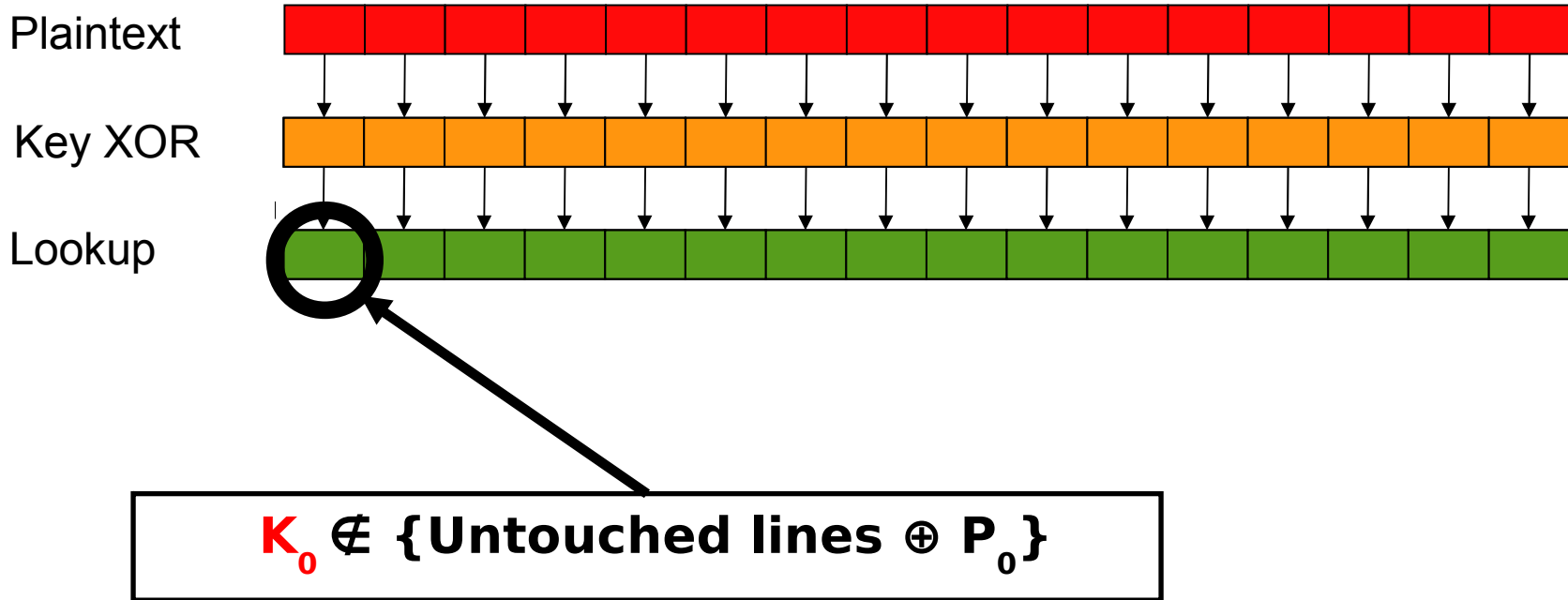
aes_encrypt(random_p());

while(i < CACHE_SIZE)
    t[i++] = timed_read(p, i);
```

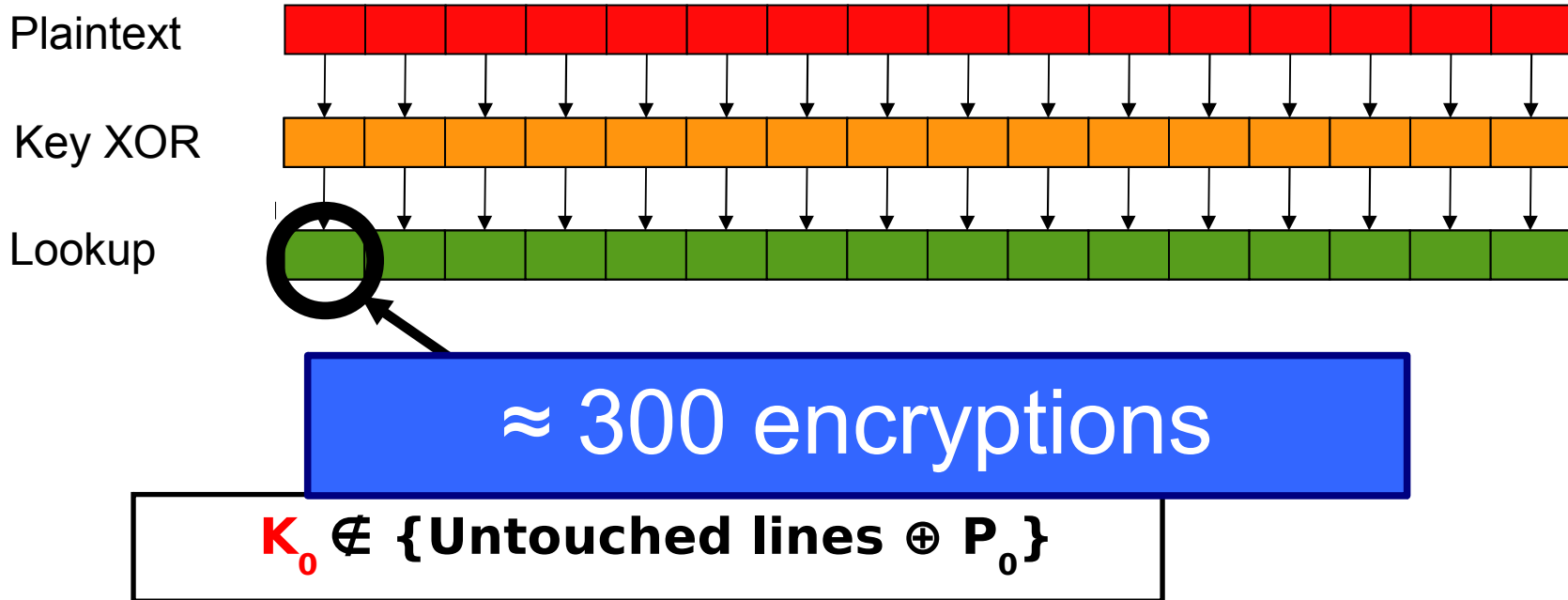
5) All untouched lines yield constraints



5) All untouched lines yield constraints



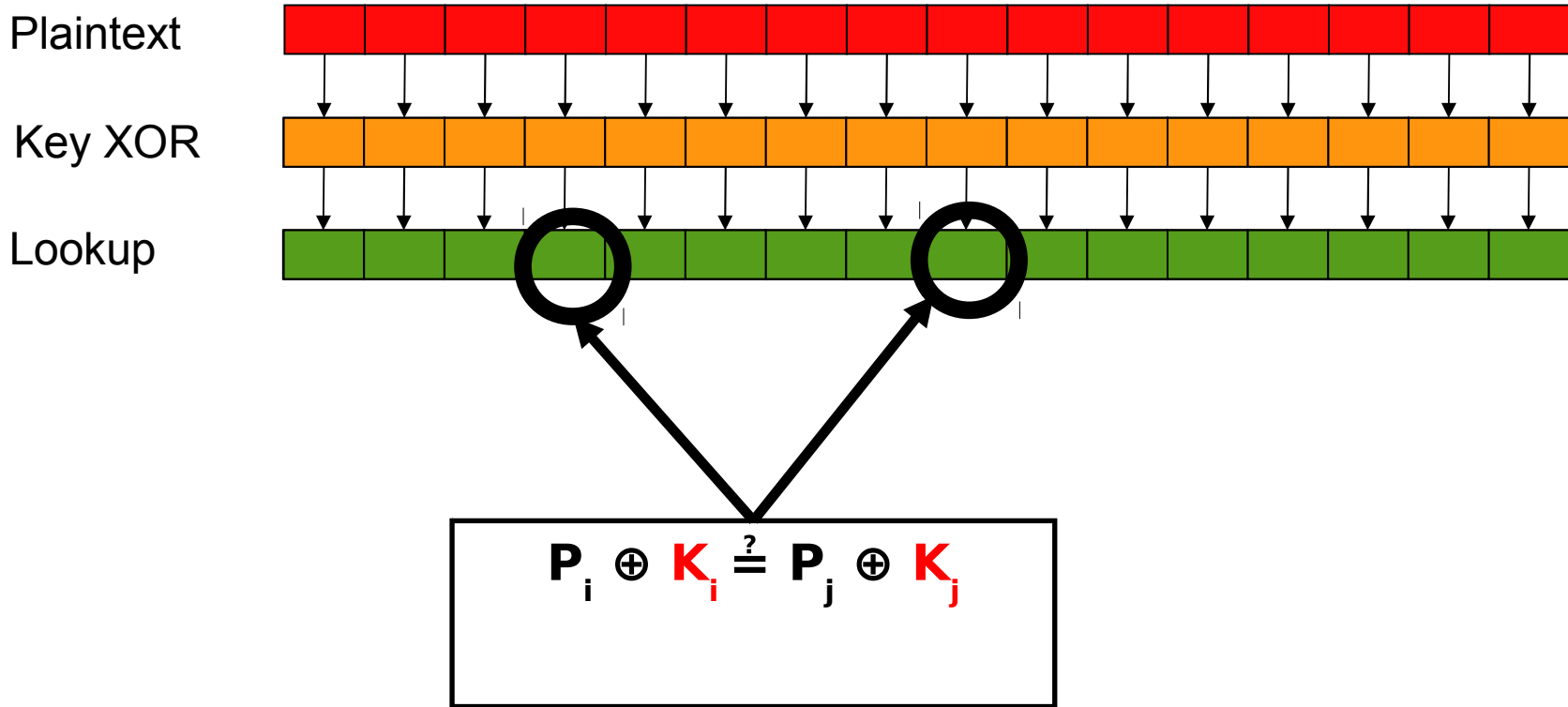
5) All untouched lines yield constraints



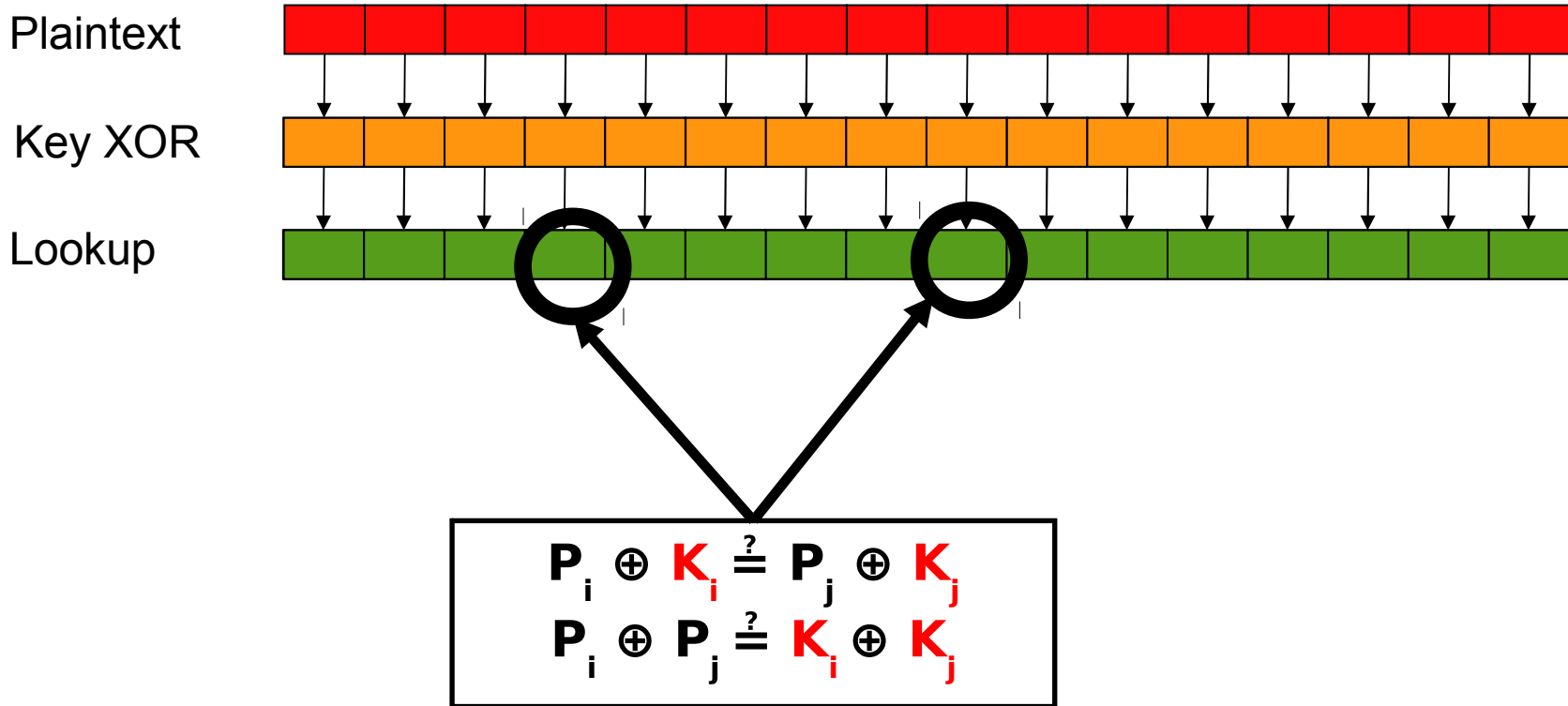
Cache timing attack

(Bonneau and Mironov, 2006)

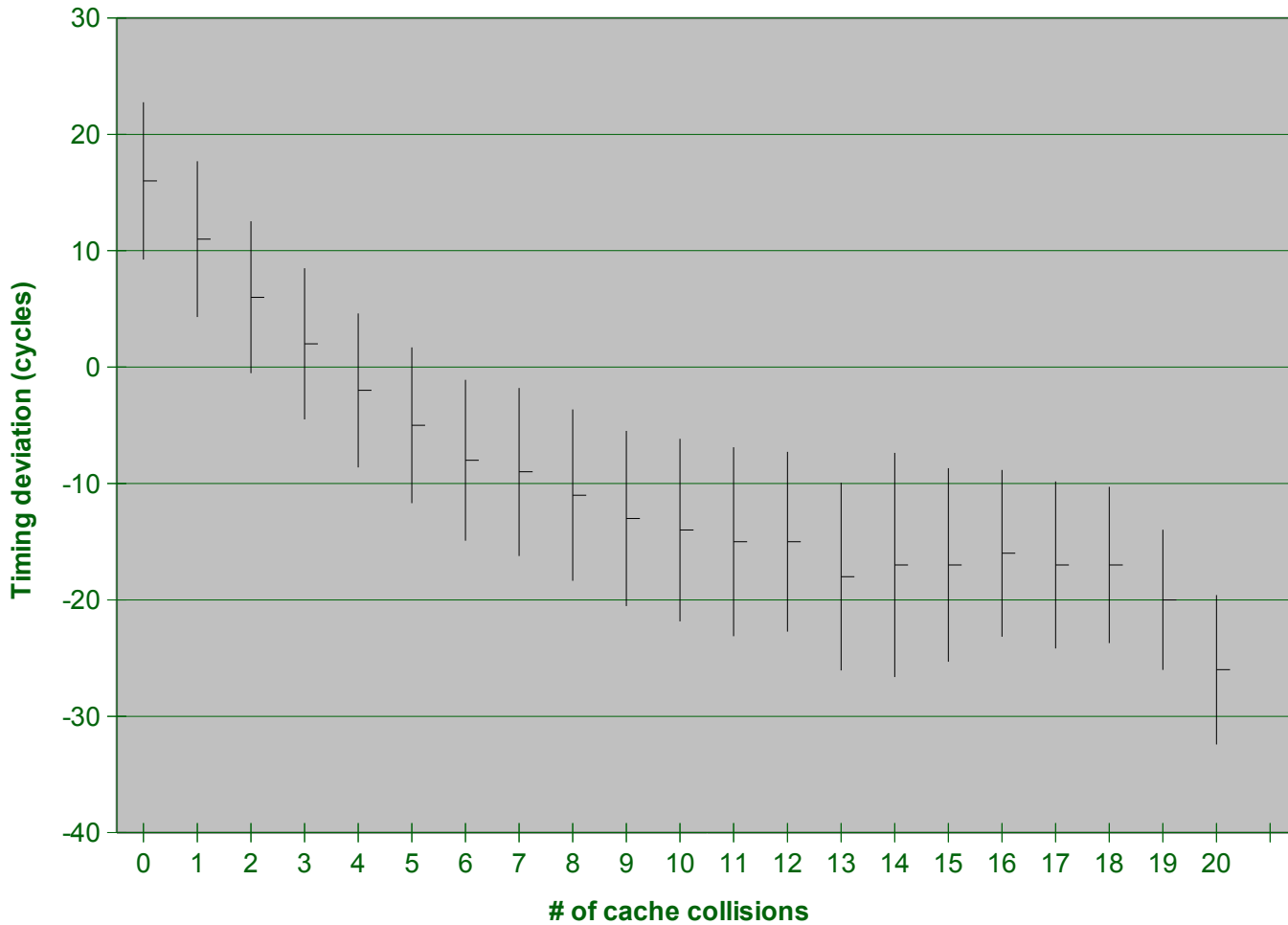
Observation: self-collisions lower encryption time



Observation: self-collisions lower encryption time



Internal collisions cause most timing variation

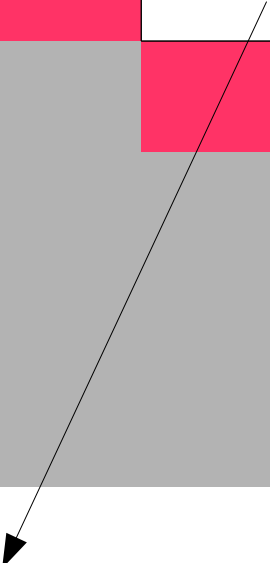


Key byte differences ranked by average time

	K0	K1	K2	...	K15
K0					
K1					
K2					
...					
K15					

Key byte differences ranked by average time

	K0	K1	K2	...	K15
K0					
K1					
K2					
...					
K15					



0)	f2	1024.32
1)	37	1036.71
2)	7a	1036.84
3)	26	1036.91
	...	
255)	a2	1038.42

Key byte differences ranked by average time

	K0	K1	K2	...	K15
K0					
K1					
K2					
...					
K15					

0)	f2	1024.32
1)	37	1036.71
2)	7a	1036.84
3)	26	1036.91
	...	
255)	a2	1038.42

0)	5d	1025.61
1)	10	1036.64
2)	46	1036.79
3)	dc	1036.98
	...	
255)	03	1038.16

Key byte differences ranked by average time

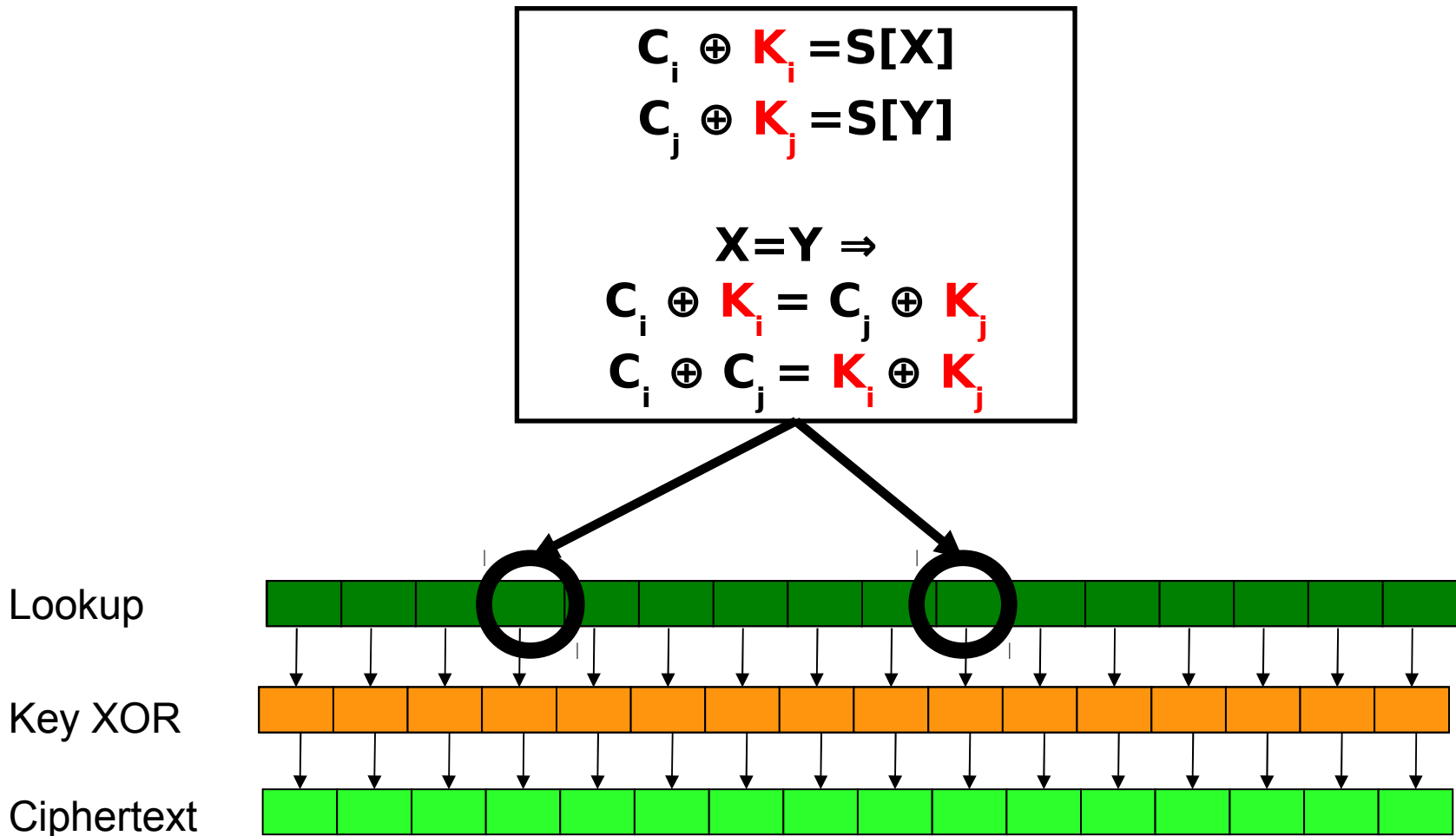
	K0	K1	K2	...	K15
K0					
K1					
K2					
...					
K15					

≈ 100,000 encryptions

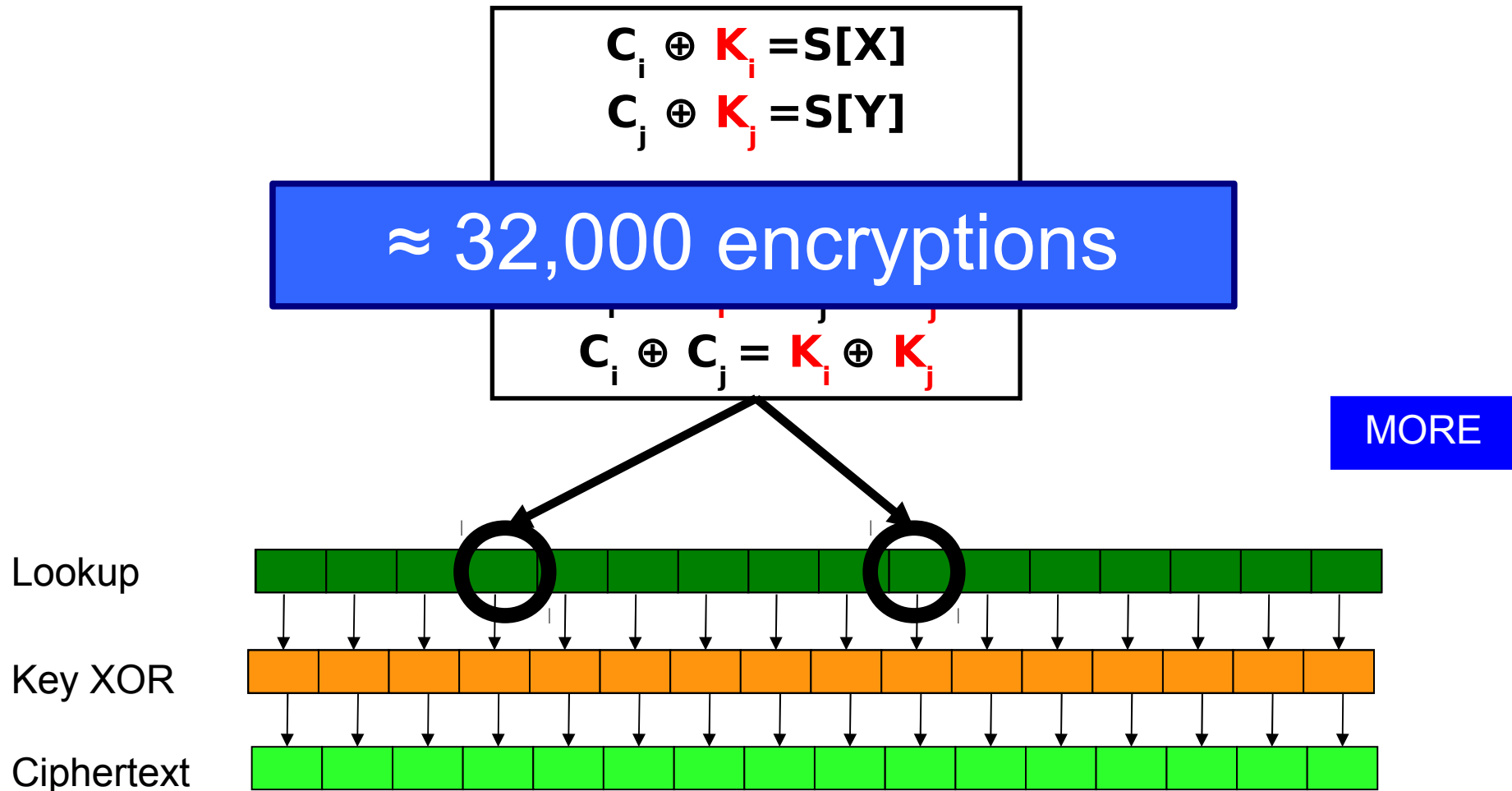
0) f2 1024.32
1) 37 1036.71
2) 7a 1036.84
3) 26 1036.91
...
255) a2 1038.42

0) 5d 1025.61
1) 10 1036.64
2) 46 1036.79
3) dc 1036.98
...
255) 03 1038.16

Final round is much better to attack



Final round is much better to attack



Hardware countermeasures on the way

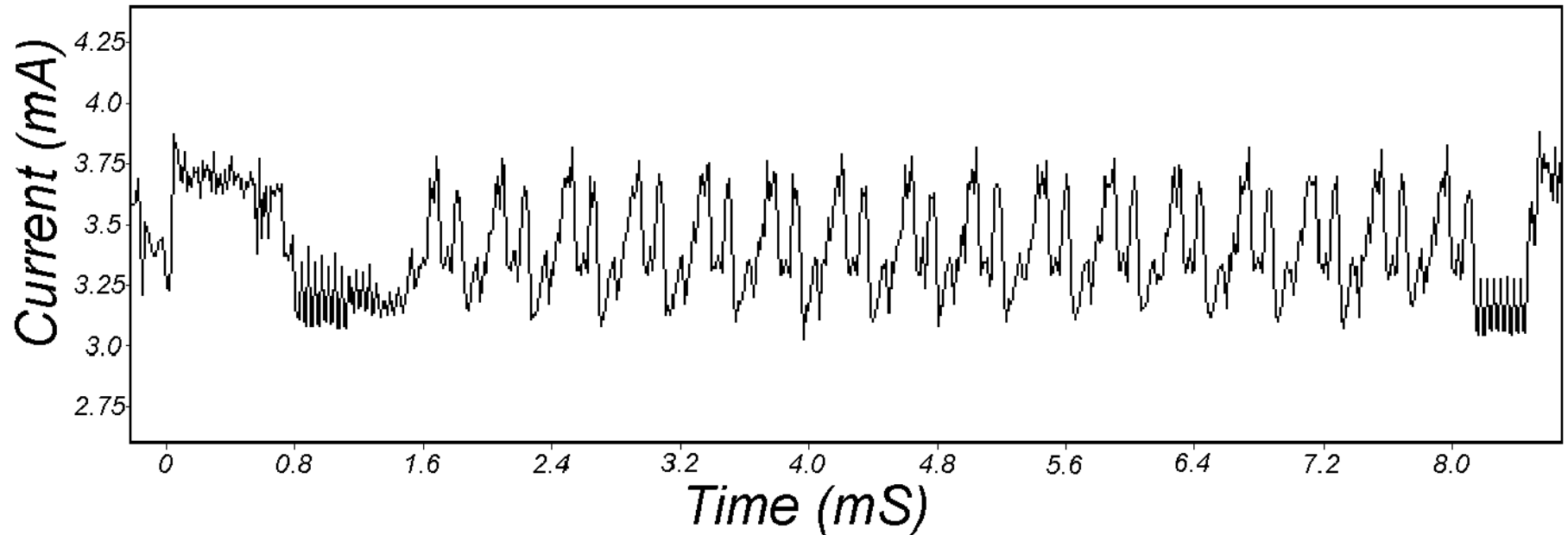
```
/*  
AES-128 encryption sequence.  
The data block is in xmm15.  
Registers xmm0-xmm10 hold the round keys (from 0 to 10 in  
this order).  
In the end, xmm15 holds the encryption result.  
*/
```

```
    pxor xmm15, xmm0           // Input whitening  
    aesenc xmm15, xmm1         // Round 1  
    aesenc xmm15, xmm2         // Round 2  
    aesenc xmm15, xmm3         // Round 3  
    aesenc xmm15, xmm4         // Round 4  
    aesenc xmm15, xmm5         // Round 5  
    aesenc xmm15, xmm6         // Round 6  
    aesenc xmm15, xmm7         // Round 7  
    aesenc xmm15, xmm8         // Round 8  
    aesenc xmm15, xmm9         // Round 9  
    aesenclast xmm15, xmm10     // Round 10
```

Differential power analysis

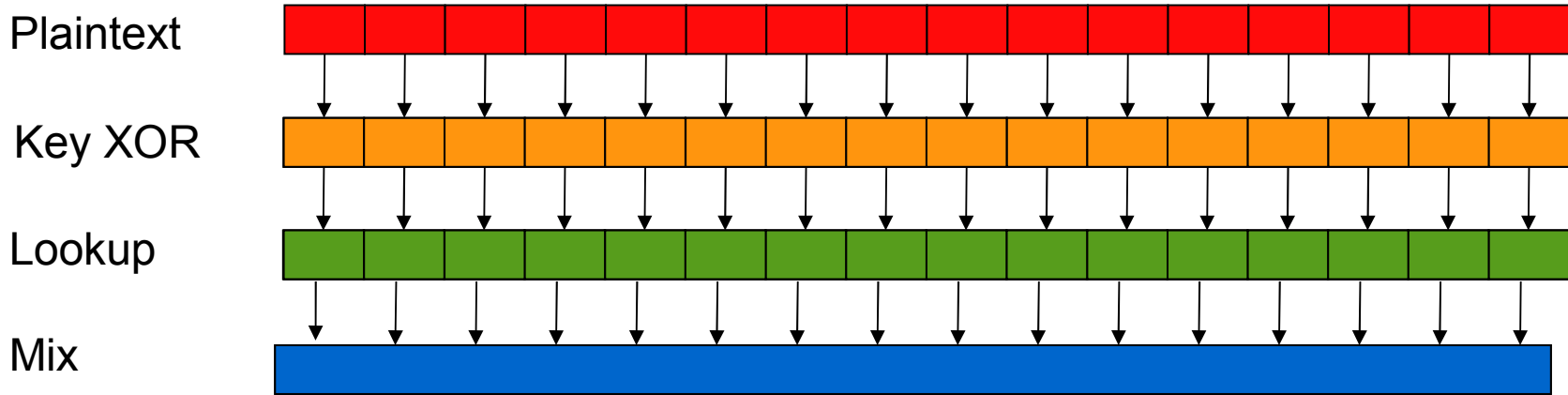
(Kocher et. al, 1999)

Simple power analysis ineffective

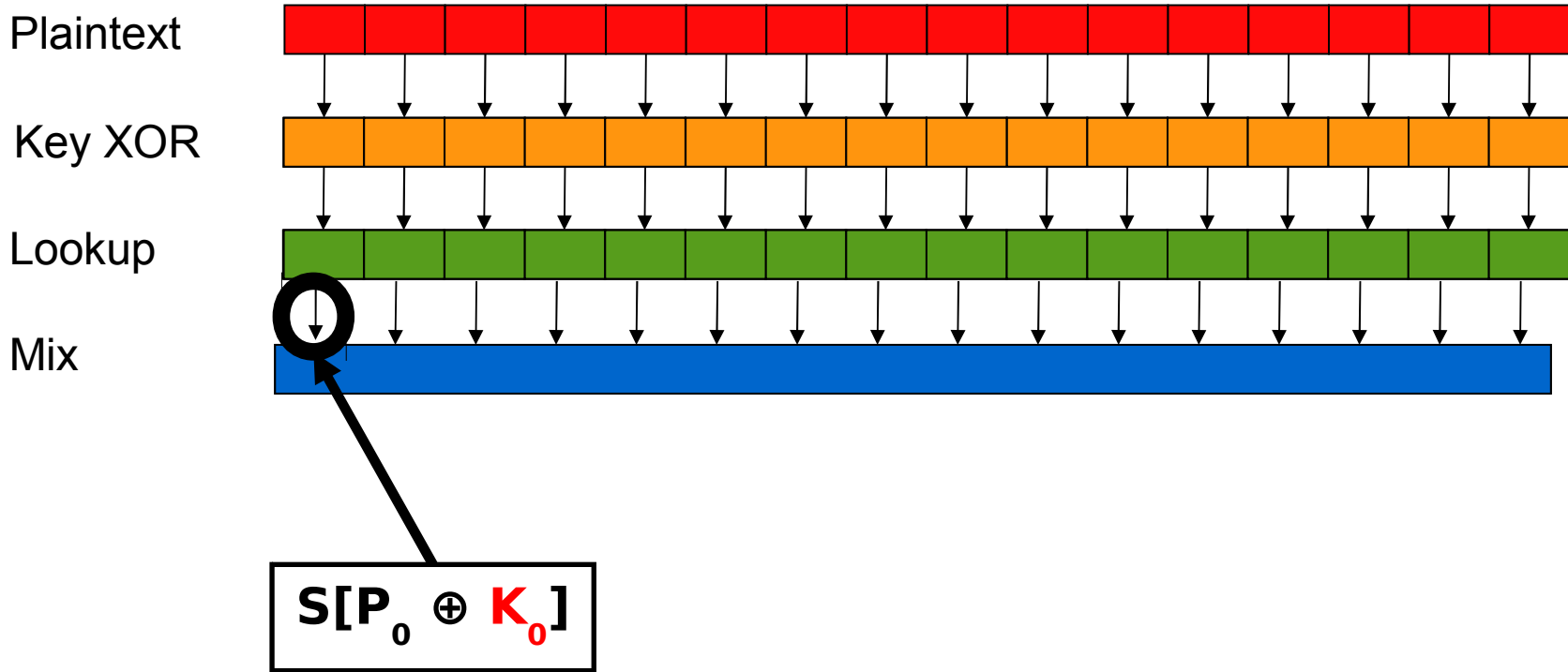


Trace courtesy of Cryptography Research, Inc.

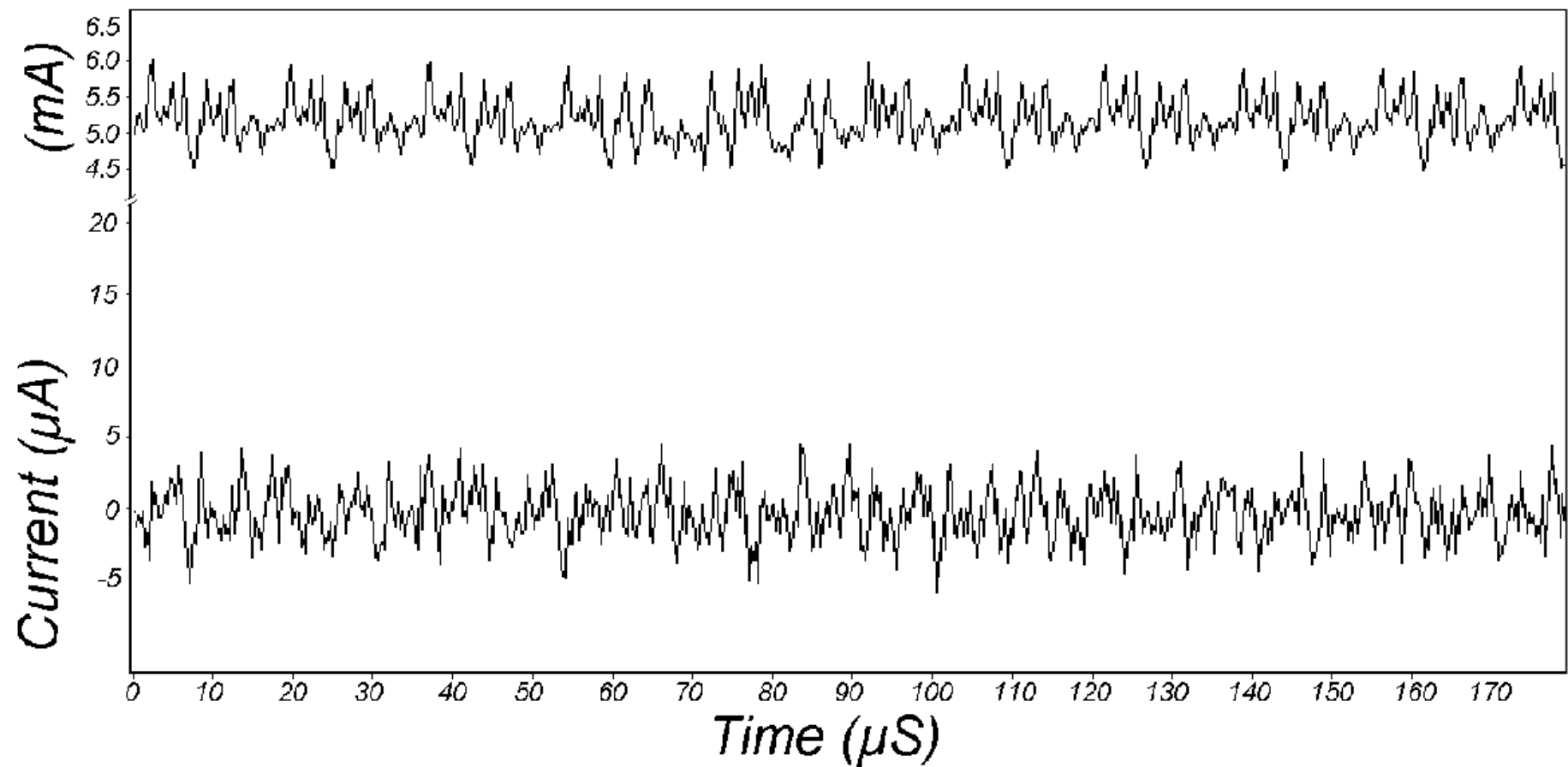
Hardware implementations don't use cache



Hardware implementations don't use cache

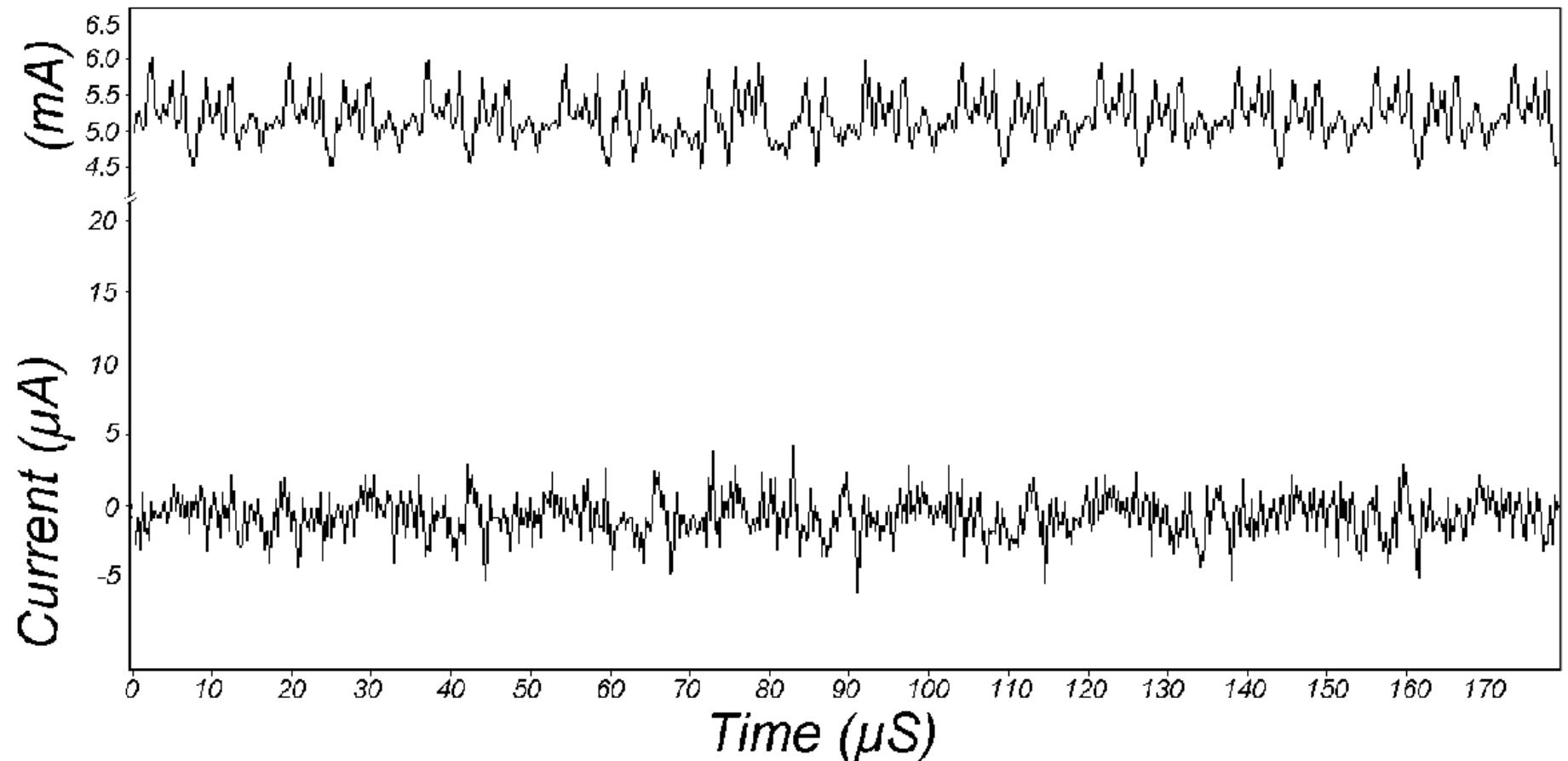


Partition traces by some predicted intermediate bit



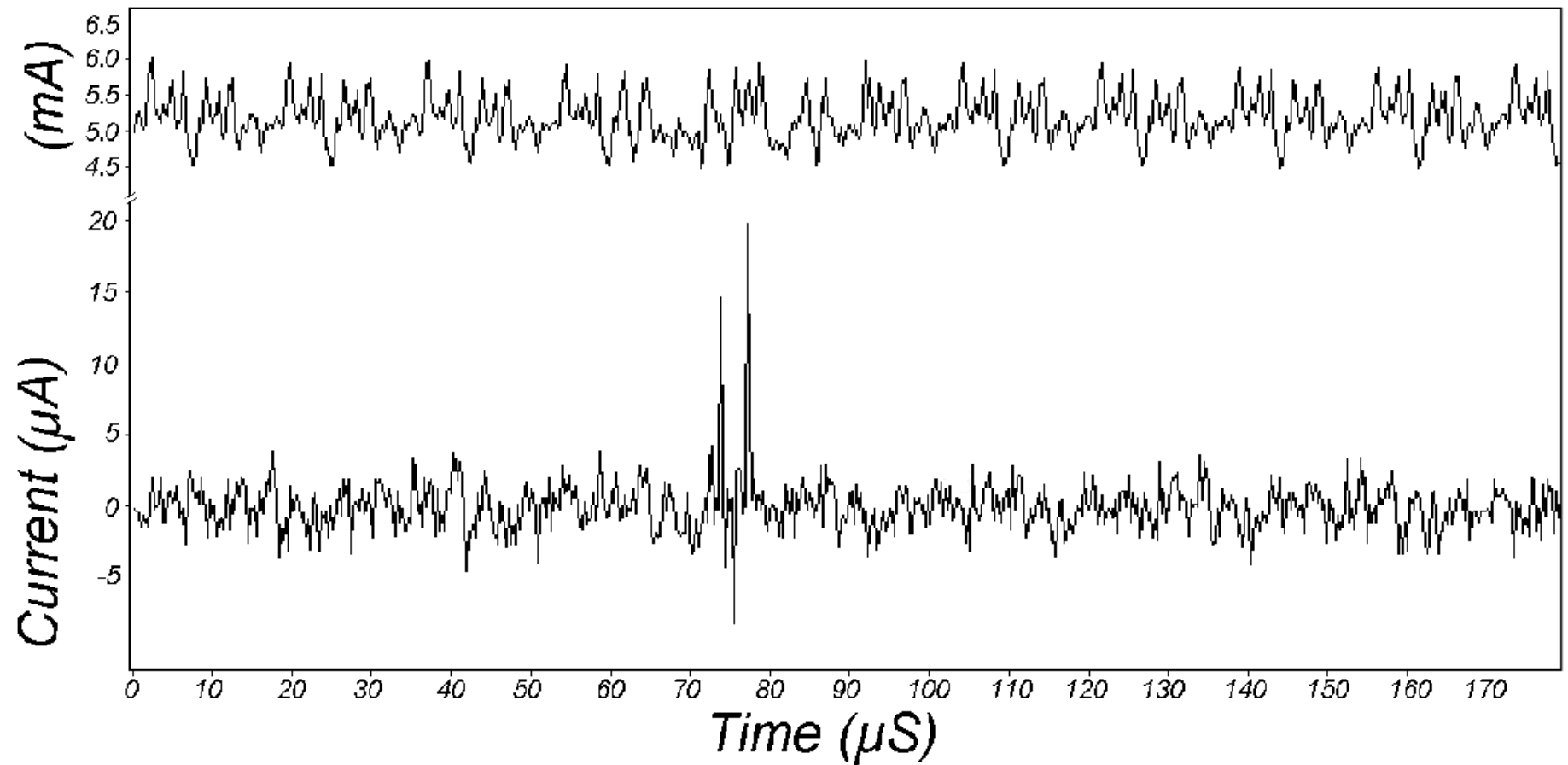
Guessing $\mathbf{K}_0 = 00$, traces where high bit of $S[P_0 \oplus \mathbf{K}_0]$ is set

Partition traces by some predicted intermediate bit



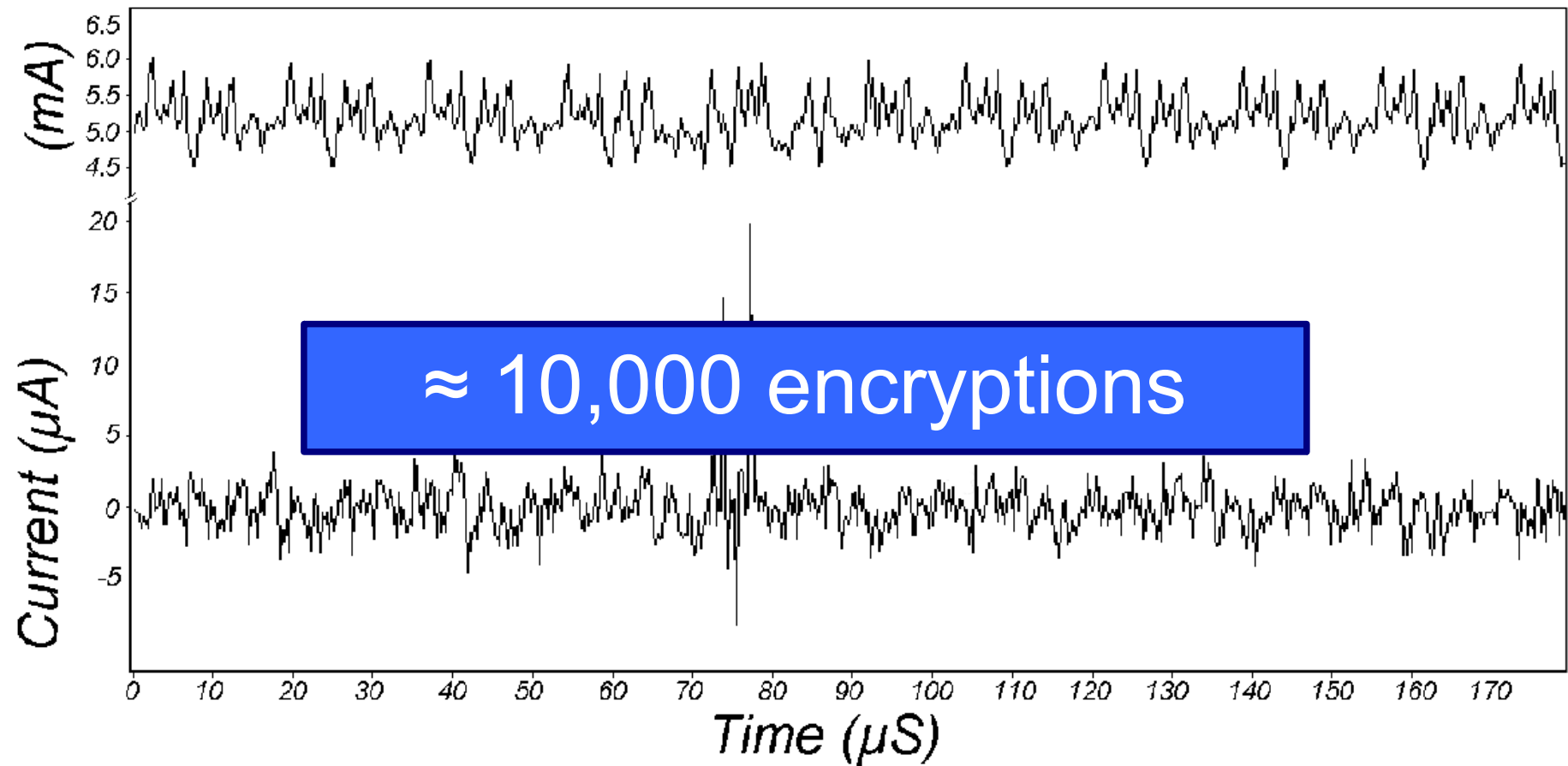
Guessing $\mathbf{K}_0 = 01$, traces where high bit of $S[P_0 \oplus \mathbf{K}_0]$ is set

Partition traces by some predicted intermediate bit



Guessing $\mathbf{K}_0 = 02$, traces where high bit of $S[P_0 \oplus \mathbf{K}_0]$ is set

Partition traces by some predicted intermediate bit



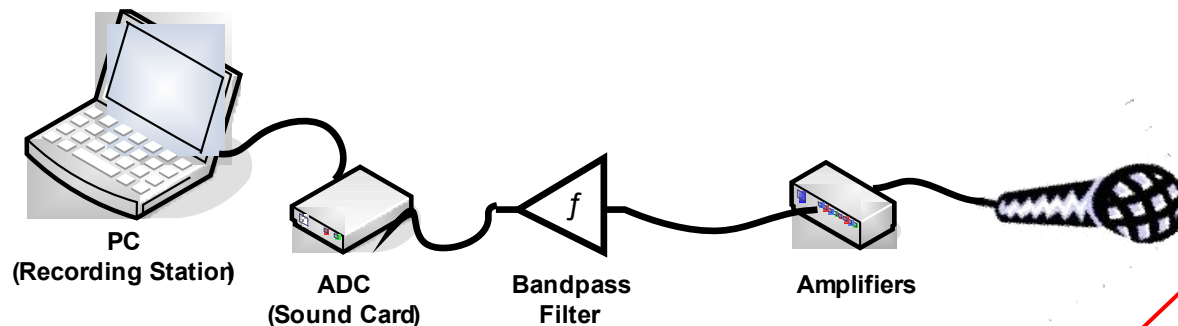
Guessing $K_0 = 02$, traces where high bit of $S[P_0 \oplus K_0]$ is set

Perfect countermeasures are very difficult

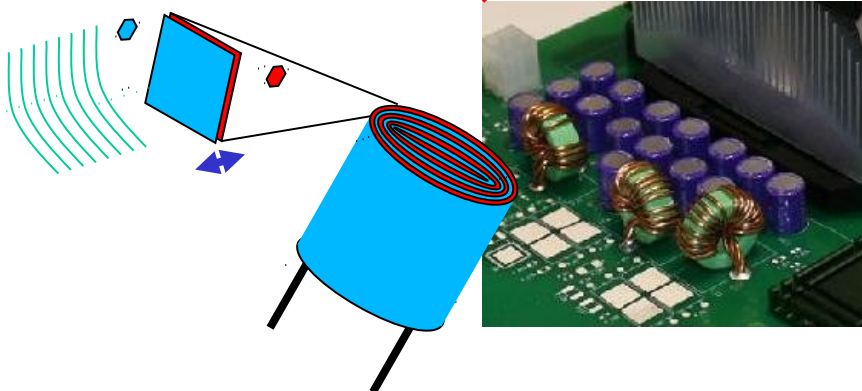
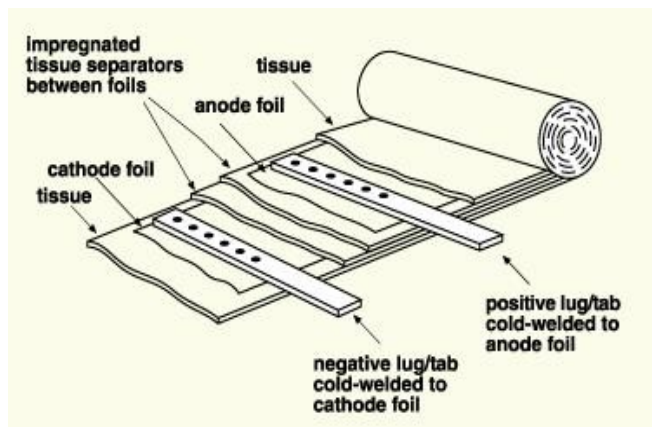


Even further down the
rabbit hole...

CPU Noise

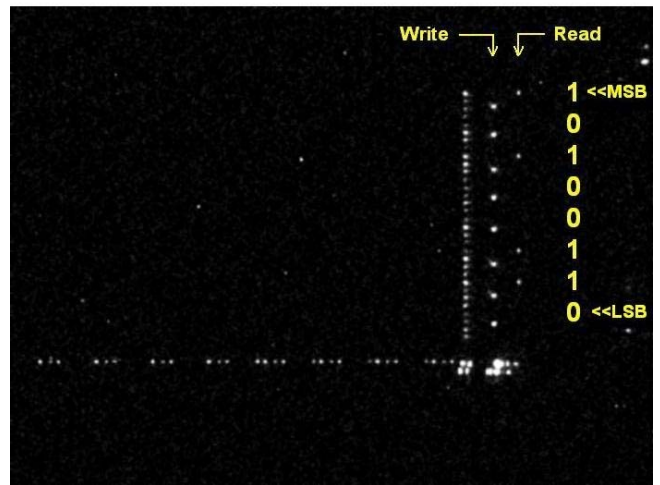
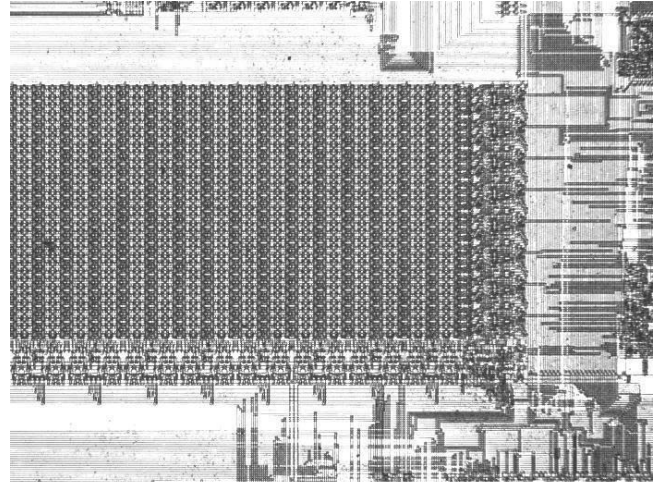
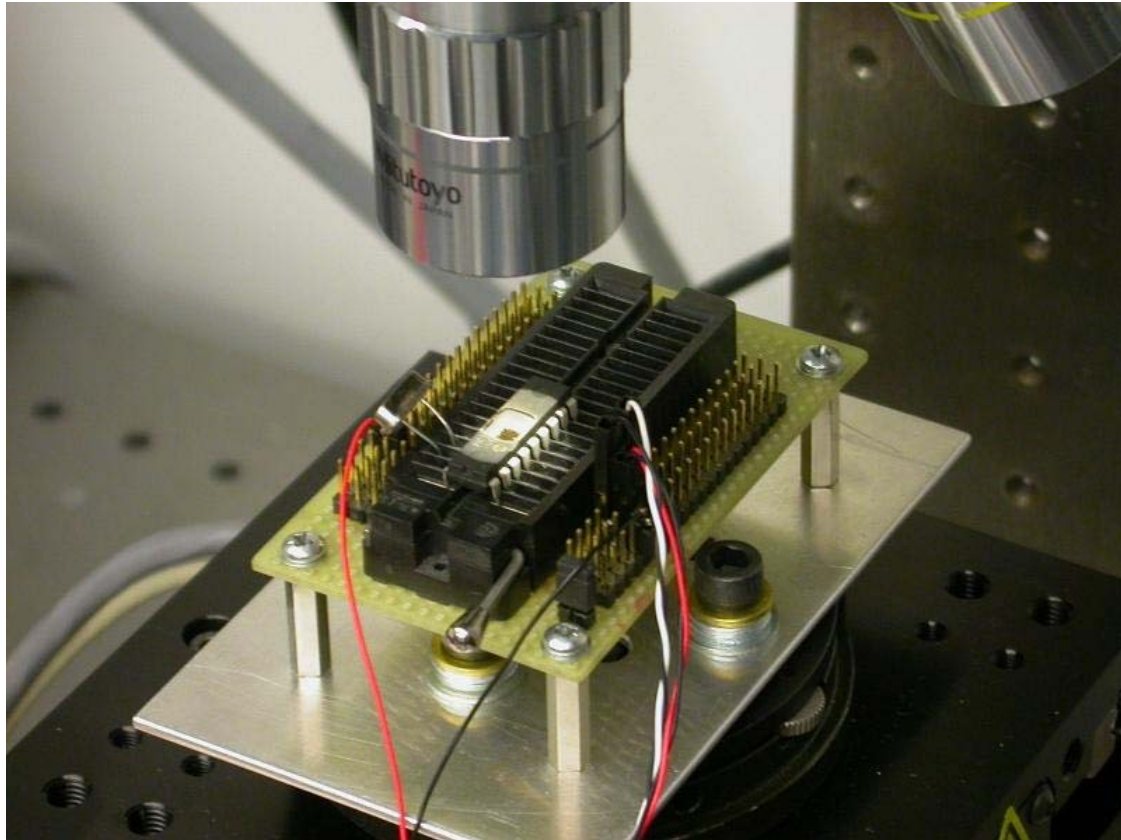


Collects sounds which emanates from CPU



Capacitor emits sound from its foils due to fast pulse charging n discharging

Photon emissions



Lessons

Don't design or implement your own crypto

Foot-Shooting Prevention Agreement

I, _____, promise that once
Your Name

I see how simple AES really is, I will
not implement it in production code
even though it would be really fun.

This agreement shall be in effect
until the undersigned creates a
meaningful interpretive dance that
compares and contrasts cache-based,
timing, and other side channel attacks
and their countermeasures.

X _____
Signature Date

Do break whatever you can

Algorithms:

- Elliptic curve DSS/DH
- Pairing-based algorithms
- AES-GCM authentication
- SHA-3 candidates

Side-channels:

- Motherboard sensors
- CPU debug registers

Killer target:

- Cross-VM key compromise

Thank you

Complication: cache lines

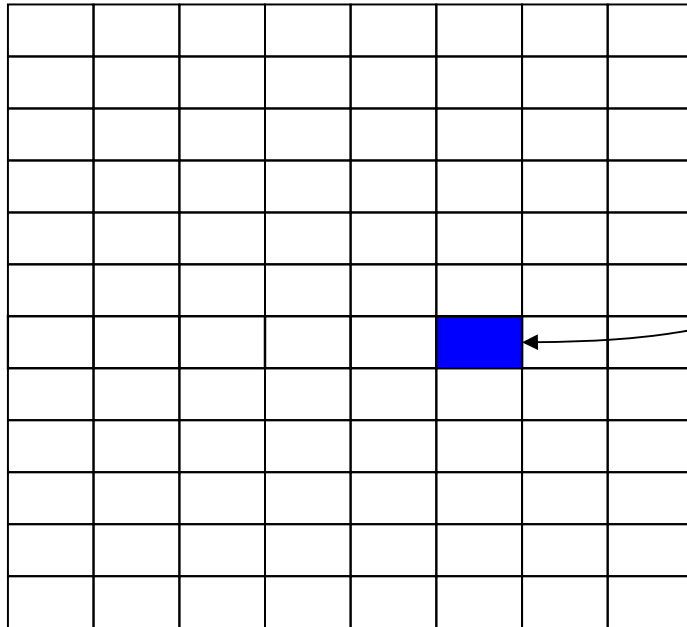
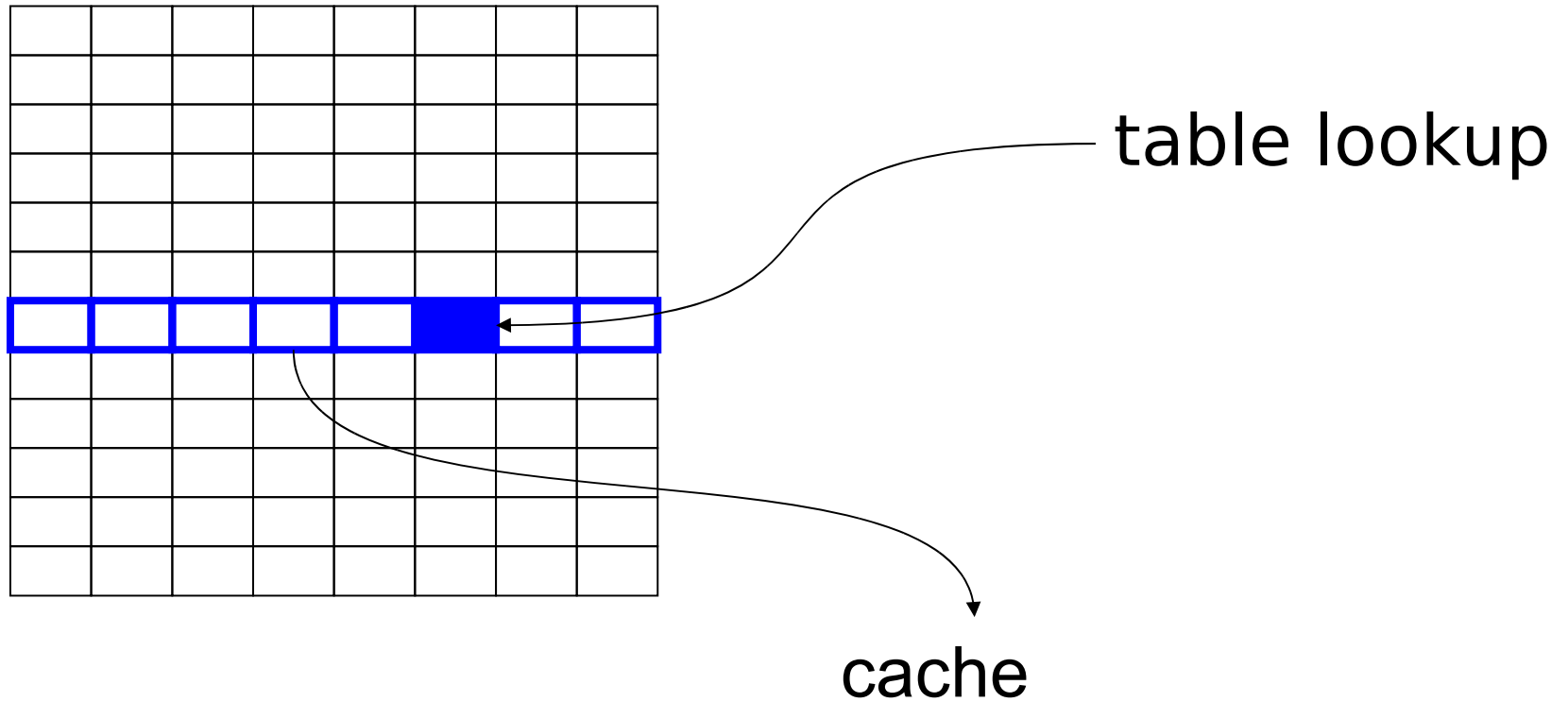


table lookup

Complication: cache lines



Complication: Table families

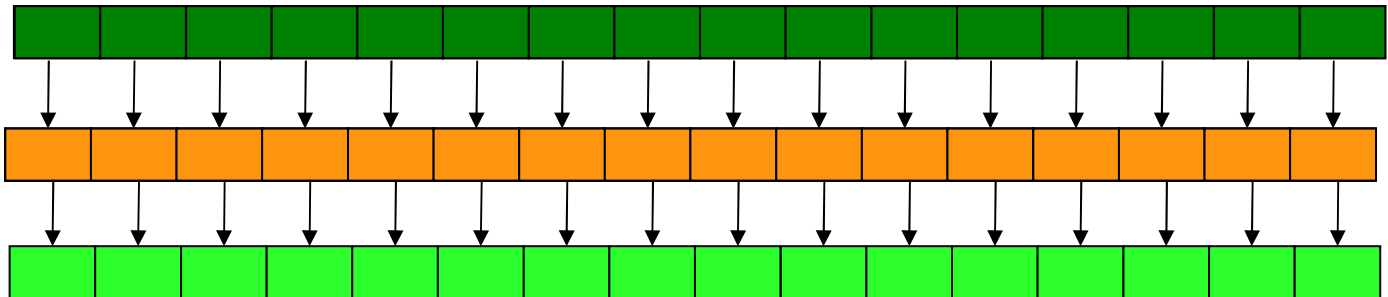
```
t0 = Te0[(s0 >> 24)          ] ^  
     Te1[(s1 >> 16) & 0xff] ^  
     Te2[(s2 >>  8) & 0xff] ^  
     Te3[(s3          ) & 0xff] ^  
     rk[0];  
t1 = Te0[(s1 >> 24)          ] ^  
     Te1[(s2 >> 16) & 0xff] ^  
     Te2[(s3 >>  8) & 0xff] ^  
     Te3[(s0          ) & 0xff] ^  
     rk[1];  
t2 = Te0[(s2 >> 24)          ] ^  
     Te1[(s3 >> 16) & 0xff] ^  
     Te2[(s0 >>  8) & 0xff] ^  
     Te3[(s1          ) & 0xff] ^  
     rk[2];  
t3 = Te0[(s3 >> 24)          ] ^  
     Te1[(s0 >> 16) & 0xff] ^  
     Te2[(s1 >>  8) & 0xff] ^  
     Te3[(s2          ) & 0xff] ^  
     rk[3];
```

Final round is much better to attack

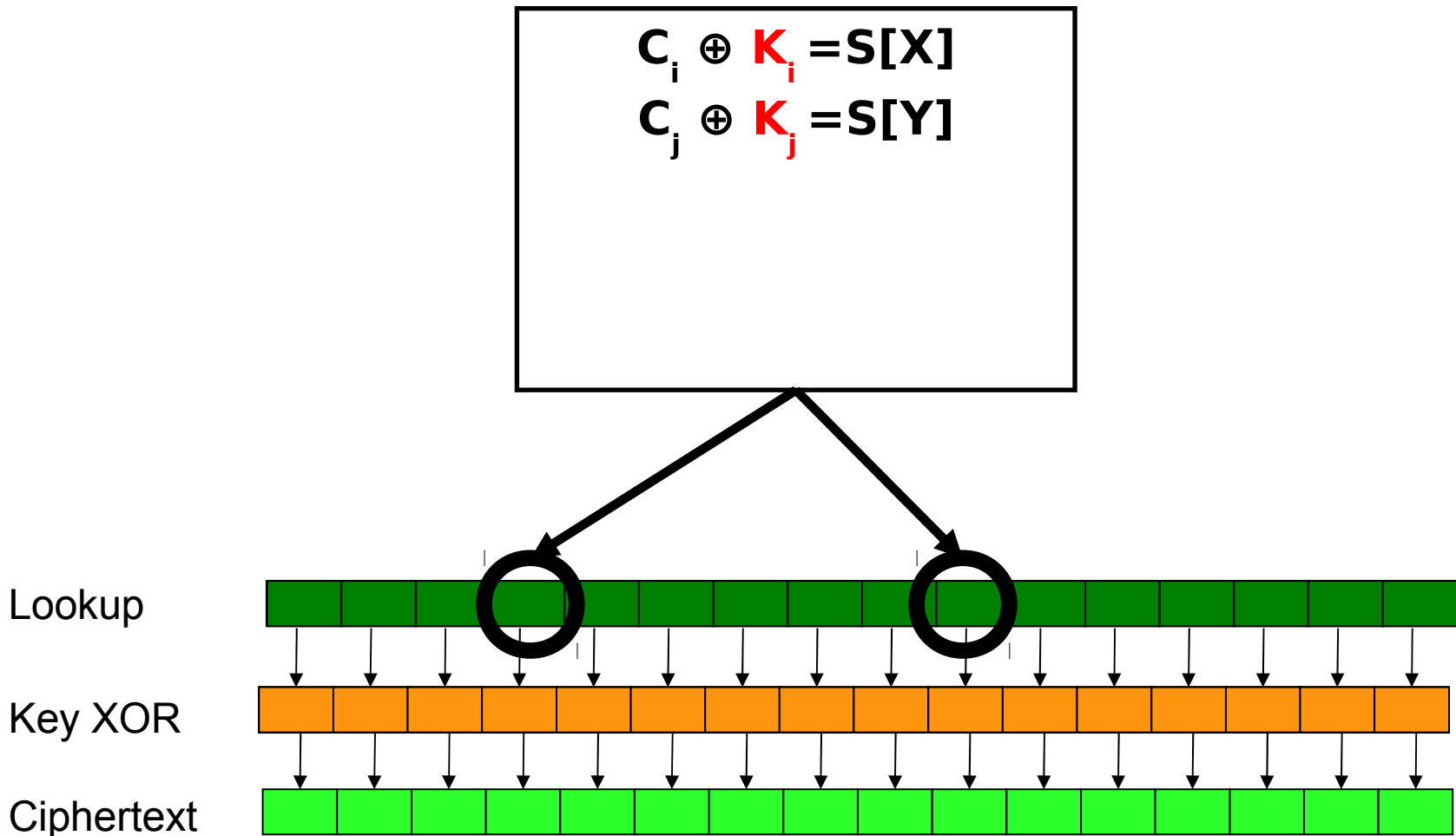
Lookup

Key XOR

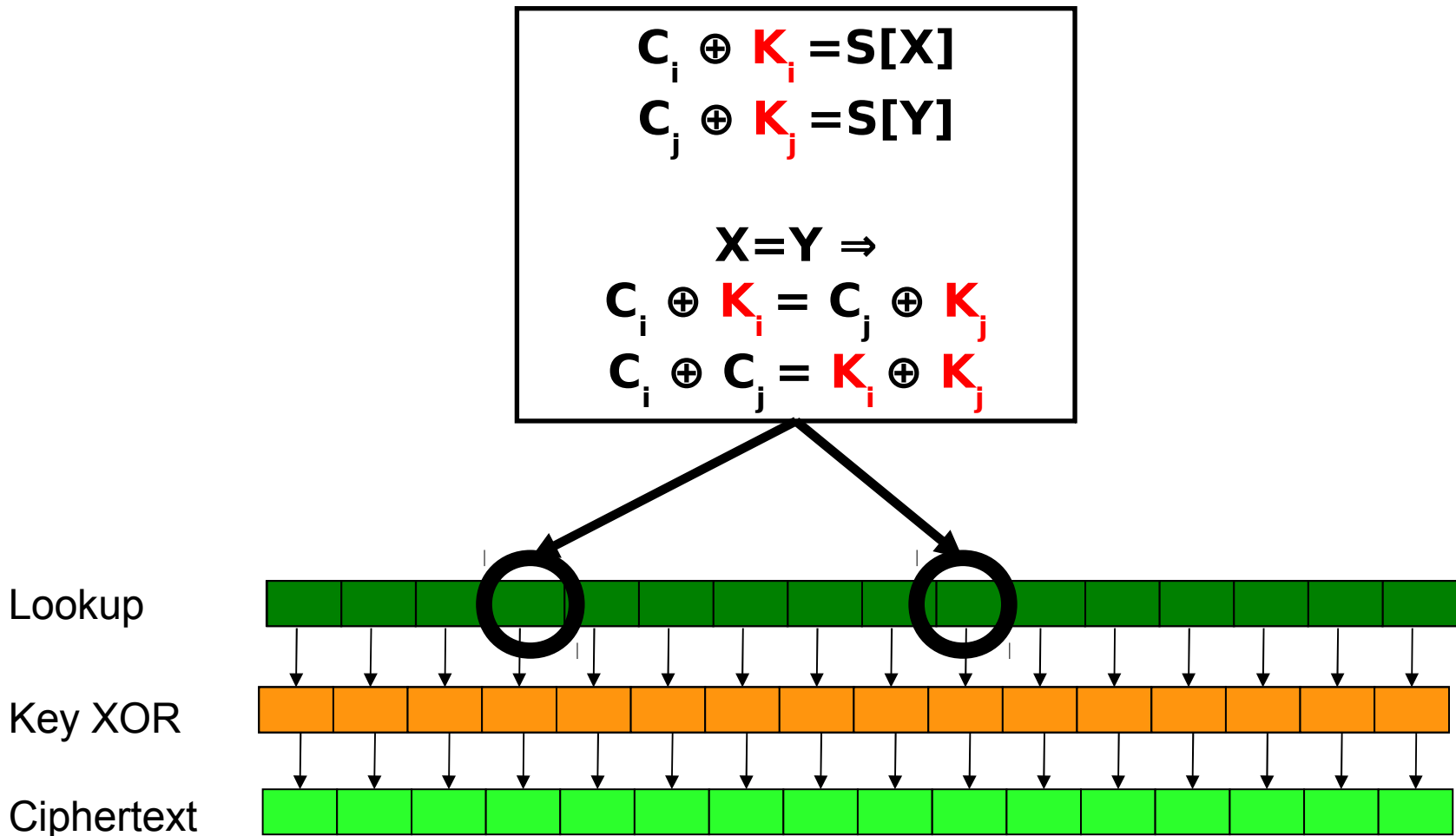
Ciphertext



Final round is much better to attack

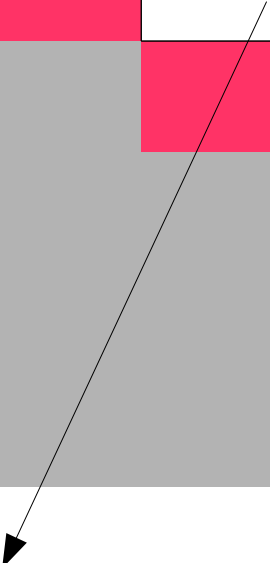


Final round is much better to attack



Better approach-rank all possible pairs

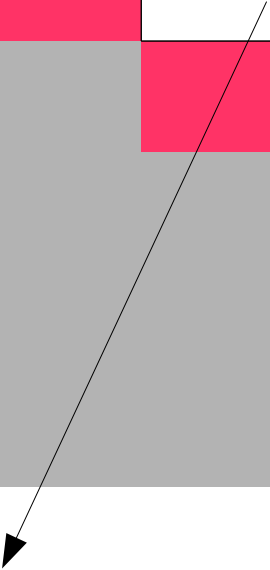
	K0	K1	K2	...	K15
K0					
K1					
K2					
...					
K15					



0)	f2	1024.32
1)	37	1036.71
2)	7a	1036.84
3)	26	1036.91
	...	
255)	a2	1038.42

Better approach-rank all possible pairs

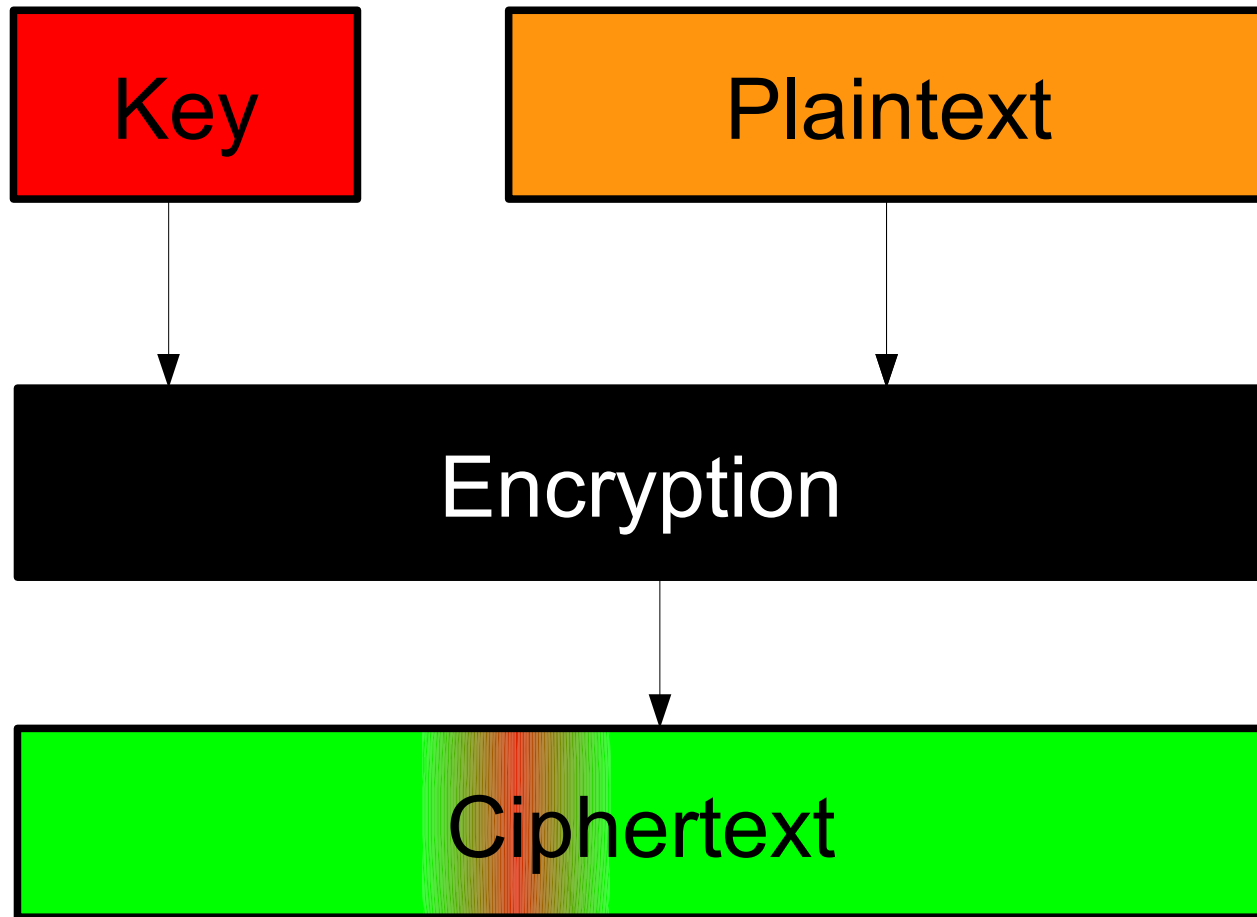
	K0	K1	K2	...	K15
K0					
K1					
K2					
...					
K15					



	0x00	0x01	...	0xFF
0x00	1036.81	1036.22	...	1036.71
0x01	1036.47	1024.32	...	1036.55
...	...	...	...	...
0xFF	1037.03	1036.89	...	1036.69

BACK

Fault injection: inducing a telling error



Normal RSA-CRT

Private Key: p, q (random primes)
 $d \equiv e^{-1} \pmod{\varphi(N)}$ (exponent)

Public Key: $N = p \cdot q$ (modulus)
 e (exponent)

Signing:
 $s_p = m^d \pmod{p}$
 $s_q = m^d \pmod{q}$
 $s = \text{CRT}(s_q, s_p) = m^d \pmod{N}$

Fault-injected RSA-CRT

Private Key: p, q (random primes)
 $d \equiv e^{-1} \pmod{\phi(N)}$ (exponent)

Public Key: $N = p \cdot q$ (modulus)
 e (exponent)

Signing:
 $s_p = m^d \pmod{p}$
 $x \neq m^d \pmod{q}$
 $s = \text{CRT}(x, s_p) \neq m^d \pmod{N}$

Fault-injected RSA-CRT

Private Key: p, q (random primes)

$d \equiv e^{-1} \pmod{\varphi(N)}$ (exponent)

Public Key: $N = p \cdot q$ (modulus)

e (exponent)

Signing: $s^e = m \pmod{p}$

$s^e \neq m \pmod{q}$

$p = \text{GCD}(s^e - m, N)$

BACK