Storing passwords in Linux

- Stored in /etc/shadow

```
seed:$6$5MfvmFOaDU$CVt7…:14400:0:99999:7::
```

- Colon-separated values, including:
  - username:hash
- Hash is $-separated values:
  - Id (6 = based on SHA-512, 1 = based on MD5, etc.)
  - Salt
  - The hash (after being taken multiple times)

Misusing crypto

Avoid shooting yourself in the foot:

- Do not **roll your own** cryptographic mechanisms
  - Takes peer review
  - Apply Kerkhoff’s principle
- Do not ***misuse*** existing crypto
- Do not even ***implement*** the underlying crypto

An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

A paper from 2013 that looked at how Android apps use crypto, as a function of 6 “rules” that reflect the bare minimum a secure programmer should know:

An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

A paper from 2013 that looked at how Android apps use crypto, as a function of 6 "rules" that reflect the bare minimum a secure programmer should know:

1. Do not use **ECB** mode for encryption. Period.

An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

A paper from 2013 that looked at how Android apps use crypto, as a function of 6 "rules" that reflect the bare minimum a secure programmer should know:

1. Do not use **ECB** mode for encryption. Period.
2. Do not use a **non-random IV** for CBC encryption.

An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

A paper from 2013 that looked at how Android apps use crypto, as a function of 6 "rules" that reflect the bare minimum a secure programmer should know:

1. Do not use **ECB** mode for encryption. Period.
2. Do not use a **non-random IV** for CBC encryption.
3. Do not use **constant encryption keys**.

An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

A paper from 2013 that looked at how Android apps use crypto, as a function of 6 "rules" that reflect the bare minimum a secure programmer should know:

1. Do not use **ECB** mode for encryption. Period.
2. Do not use a **non-random IV** for CBC encryption.
3. Do not use **constant encryption keys**.
4. (see paper)

An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

A paper from 2013 that looked at how Android apps use crypto, as a function of 6 "rules" that reflect the bare minimum a secure programmer should know:

1. Do not use **ECB** mode for encryption. Period.
2. Do not use a **non-random IV** for CBC encryption.
3. Do not use **constant encryption keys**.
4. (see paper)
5. (see paper)

An Empirical Study of Cryptographic Misuse in Android Applications

Manuel Egele, David Brumley
Carnegie Mellon University
{megele,dbrumley}@cmu.edu

Yanick Fratantonio, Christopher Kruegel
University of California, Santa Barbara
{yanick,chris}@cs.ucsb.edu

A paper from 2013 that looked at how Android apps use crypto, as a function of 6 "rules" that reflect the bare minimum a secure programmer should know:

1. Do not use **ECB** mode for encryption. Period.
2. Do not use a **non-random IV** for CBC encryption.
3. Do not use **constant encryption keys**.
4. (see paper)
5. (see paper)
6. Do not use **static seeds** to seed `SecureRandom(.)`

Crypto misuse in Android apps

15,134 apps from Google play used crypto;
Analyzed **11,748** of them

Crypto misuse in Android apps

15,134 apps from Google play used crypto;
Analyzed **11,748** of them

	# apps	violated rule
48%	5,656	Uses ECB (BouncyCastle default) (R1)
31%	3,644	Uses constant symmetric key (R3)
17%	2,000	Uses ECB (Explicit use) (R1)
16%	1,932	Uses constant IV (R2)
	1,636	Used iteration count $<$ 1,000 for PBE(R5)
14%	1,629	Seeds SecureRandom with static (R6)
	1,574	Uses static salt for PBE (R4)
12%	1,421	No violation

Crypto misuse in Android apps

15,134 apps from Google play used crypto;
Analyzed **11,748** of them

	# apps	violated rule
48%	5,656	Uses ECB (BouncyCastle default) (R1)
31%	3,644	Uses constant symmetric key (R3)
17%	2,000	Uses ECB (Explicit use) (R1)
16%	1,932	Uses constant IV (R2)
	1,636	Used iteration count < 1,000 for PBE(R5)
14%	1,629	Seeds SecureRandom with static (R6)
	1,574	Uses static salt for PBE (R4)
12%	1,421	No violation

BouncyCastle defaults

- BouncyCastle is a library that conforms to Java's `Cipher` interface:

```
Cipher c =
   Cipher.getInstance("AES/CBC/PKCS5Padding");

// Ultimately end up wrapping a ByteArrayOutputStream
// in a CipherOutputStream
```

- Java documentation specifies:

> If no mode or padding is specified, provider-specific default values for the mode and padding scheme are used. For example, the `SunJCE` provider uses `ECB` as the default mode, and `PKCS5Padding` as the default padding scheme for `DES`, `DES-EDE` and `Blowfish` ciphers.

#Occurences	Symmetric encryption scheme
5878	AES/CBC/PKCS5Padding
4803	AES *
1151	DES/ECB/NoPadding
741	DES *
501	DESede *
473	DESede/ECB/PKCS5Padding
468	AES/CBC/NoPadding
443	AES/ECB/PKCS5Padding
235	AES/CBC/PKCS7Padding
221	DES/ECB/PKCS5Padding
220	AES/ECB/NoPadding
205	DES/CBC/PKCS5Padding
155	AES/ECB/PKCS7Padding
104	AES/CFB8/NoPadding

Table 4: Distribution of frequently used symmetric encryption schemes. Schemes marked with * are used in ECB mode by default.

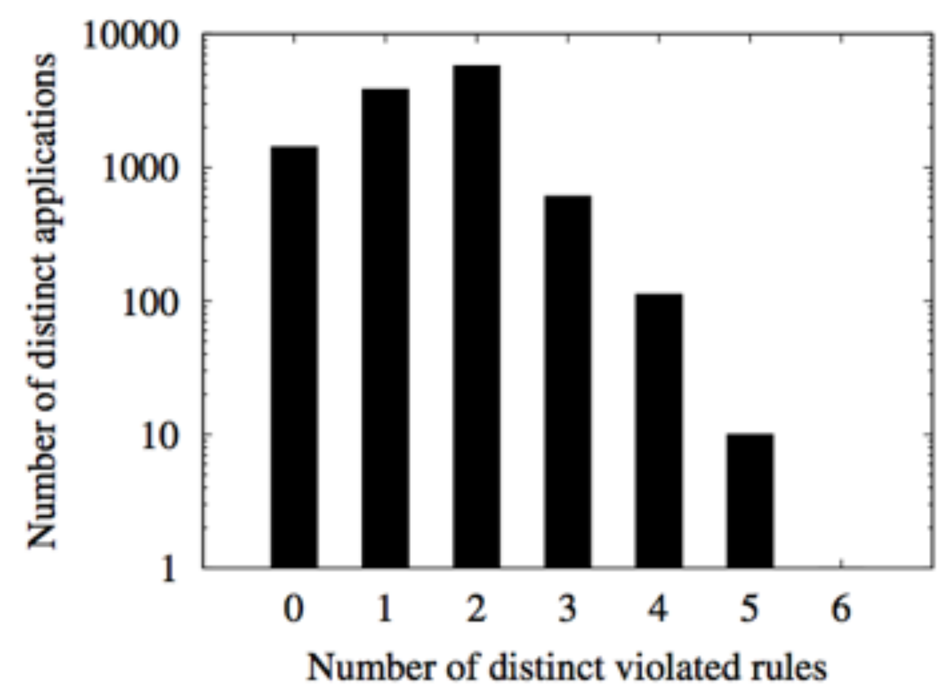
#Occurences	Symmetric encryption scheme
5878	AES/CBC/PKCS5Padding
4803	AES *
1151	DES/ECB/NoPadding
741	DES *
501	DESede *
473	DESede/ECB/PKCS5Padding
468	AES/CBC/NoPadding
443	AES/ECB/PKCS5Padding
235	AES/CBC/PKCS7Padding
221	DES/ECB/PKCS5Padding
220	AES/ECB/NoPadding
205	DES/CBC/PKCS5Padding
155	AES/ECB/PKCS7Padding
104	AES/CFB8/NoPadding

Table 4: Distribution of frequently used symmetric encryption schemes. Schemes marked with * are used in ECB mode by default.

Crypto misuse in Android apps

15,134 apps from Google play used crypto;
Analyzed **11,748** of them

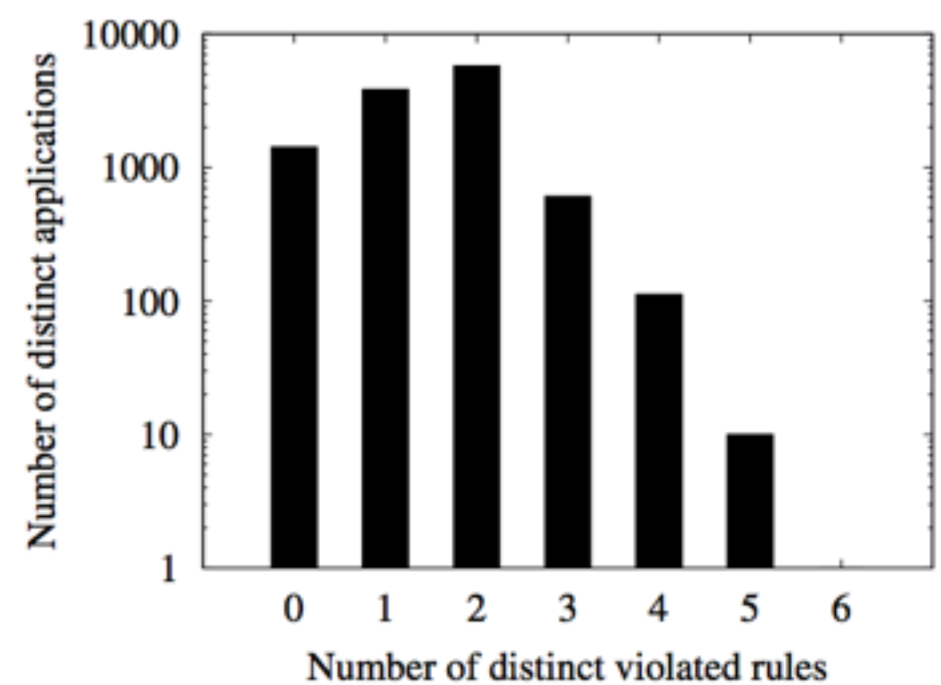
	# apps	violated rule
48%	5,656	Uses ECB (BouncyCastle default) (R1)
31%	3,644	Uses constant symmetric key (R3)
17%	2,000	Uses ECB (Explicit use) (R1)
16%	1,932	Uses constant IV (R2)
	1,636	Used iteration count < 1,000 for PBE(R5)
14%	1,629	Seeds SecureRandom with static (R6)
	1,574	Uses static salt for PBE (R4)
12%	1,421	No violation



Crypto misuse in Android apps

15,134 apps from Google play used crypto;
Analyzed **11,748** of them

	# apps	violated rule
48%	5,656	Uses ECB (BouncyCastle default) (R1)
31%	3,644	Uses constant symmetric key (R3)
17%	2,000	Uses ECB (Explicit use) (R1)
16%	1,932	Uses constant IV (R2)
	1,636	Used iteration count < 1,000 for PBE(R5)
14%	1,629	Seeds SecureRandom with static (R6)
	1,574	Uses static salt for PBE (R4)
12%	1,421	No violation



A failure of the programmers to **know the tools** they use

A failure of library writers to **provide safe defaults**

Misusing crypto

Avoid shooting yourself in the foot:

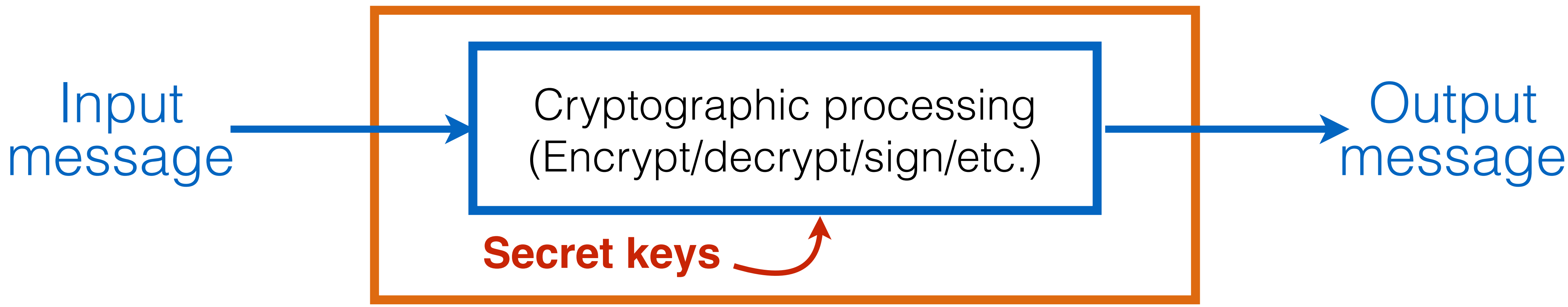
- Do not **roll your own** cryptographic mechanisms
  - Takes peer review
  - Apply Kerkhoff's principle
- Do not ***misuse*** existing crypto
- Do not even ***implement*** the underlying crypto

Why not implement AES/RSA/etc. yourself?

- Not talking about creating a brand new crypto scheme, just implementing one that's already widely accepted and used.
- Kerkhoff's principle: these are all open standards; should be implementable.
- Potentially buggy/incorrect code, but so might be others' implementations (viz. OpenSSL bugs, poor defaults in Bouncy castles, etc.)
- So why not implement it yourself?

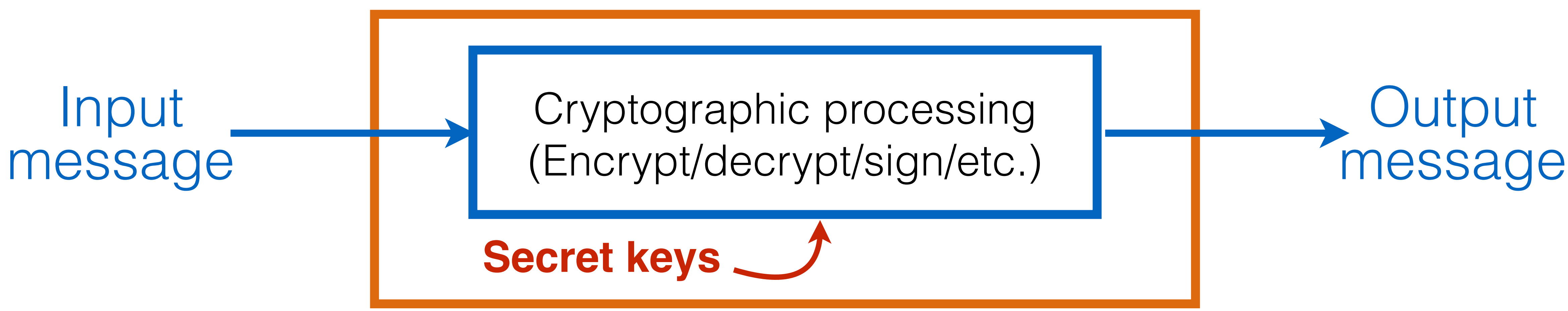
Side-channel attacks

- Cryptography concerns the *theoretical* difficulty in breaking a cipher



Side-channel attacks

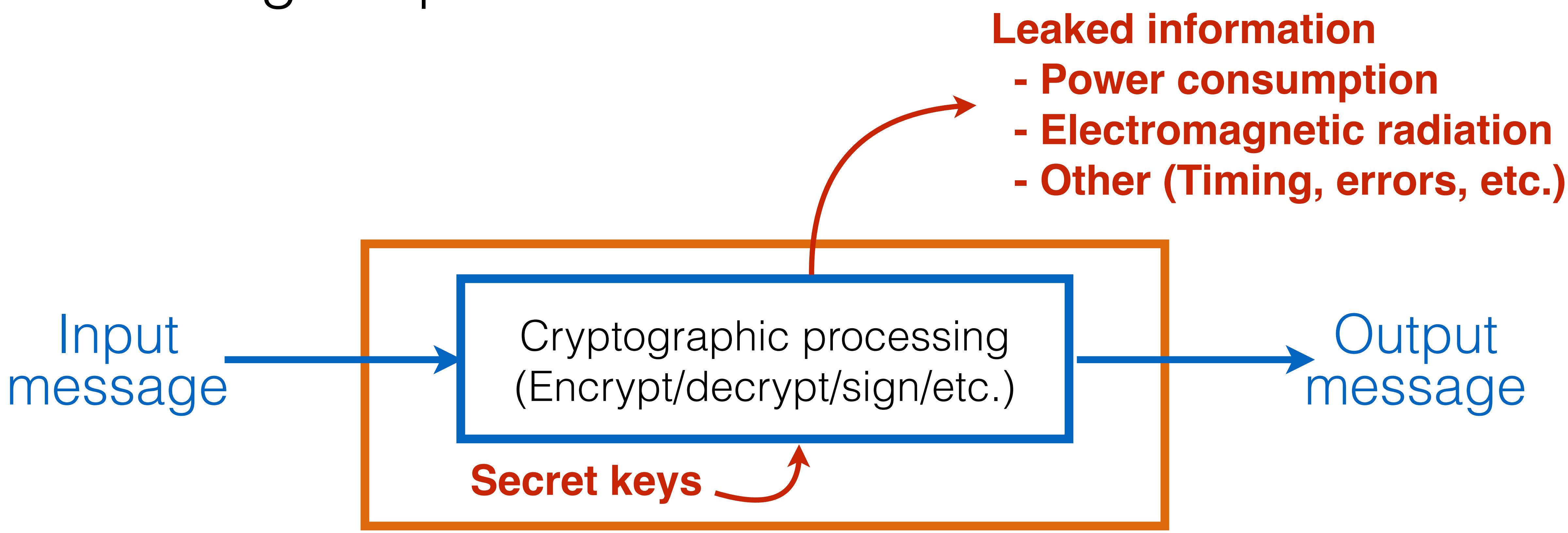
- Cryptography concerns the *theoretical* difficulty in breaking a cipher



- But what about the information that a particular *implementation* could leak?
  - Attacks based on these are "**side-channel attacks**"

Side-channel attacks

- Cryptography concerns the *theoretical* difficulty in breaking a cipher



- But what about the information that a particular *implementation* could leak?
    - Attacks based on these are "**side-channel attacks**"

Simple Power Analysis (SPA)

- Interpret *power traces* taken during a cryptographic operation

- Simple power analysis can reveal the sequence of instructions executed

SPA on DES

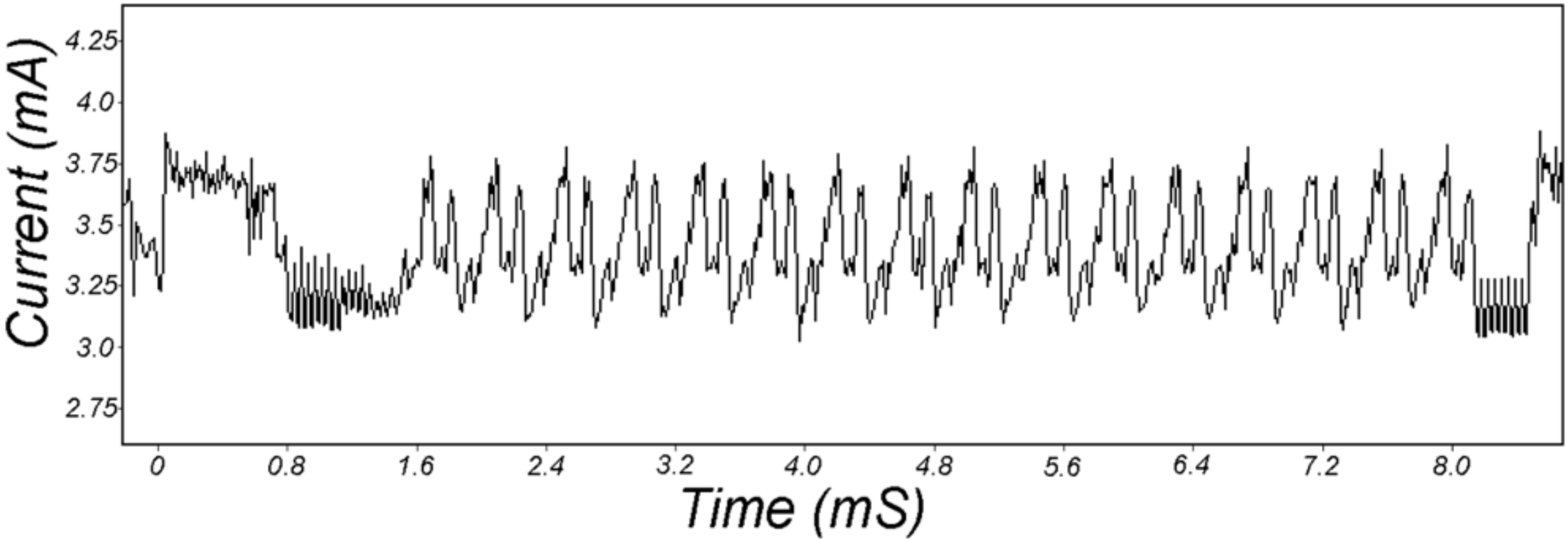


Figure 1: SPA trace showing an entire DES operation.

Overall operation clearly visible:
Can identify the **16 rounds of DES**

SPA on DES

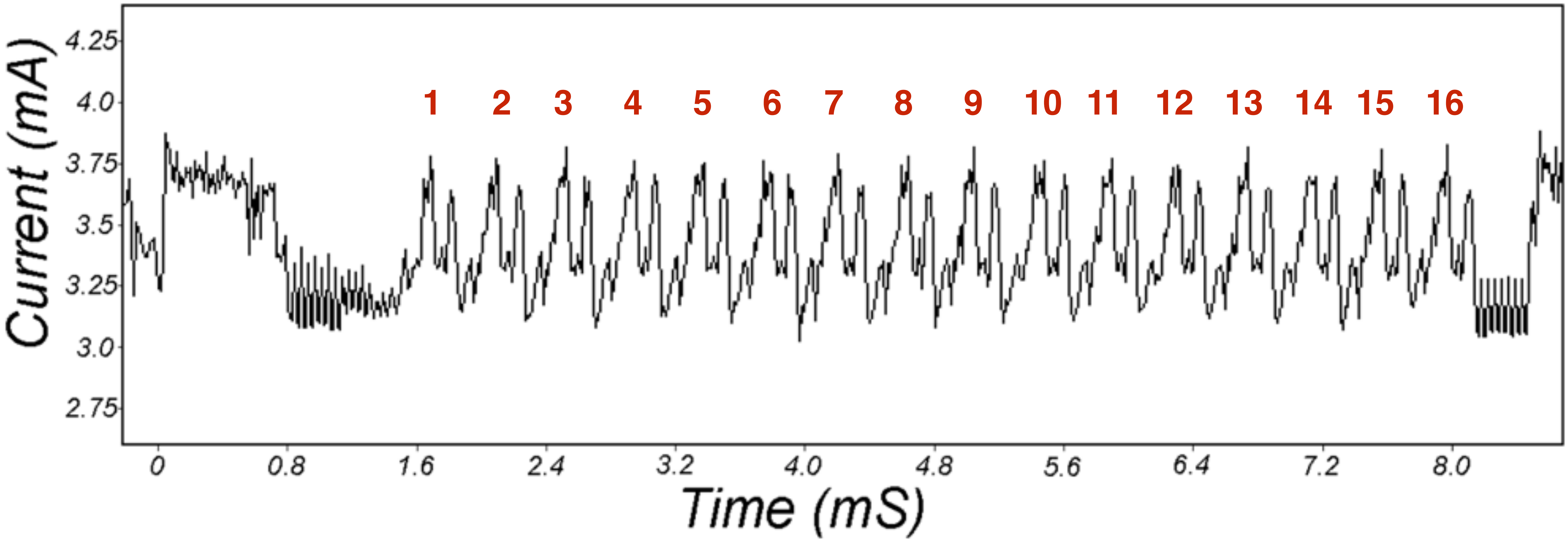


Figure 1: SPA trace showing an entire DES operation.

Overall operation clearly visible:
Can identify the **16 rounds of DES**

SPA on DES

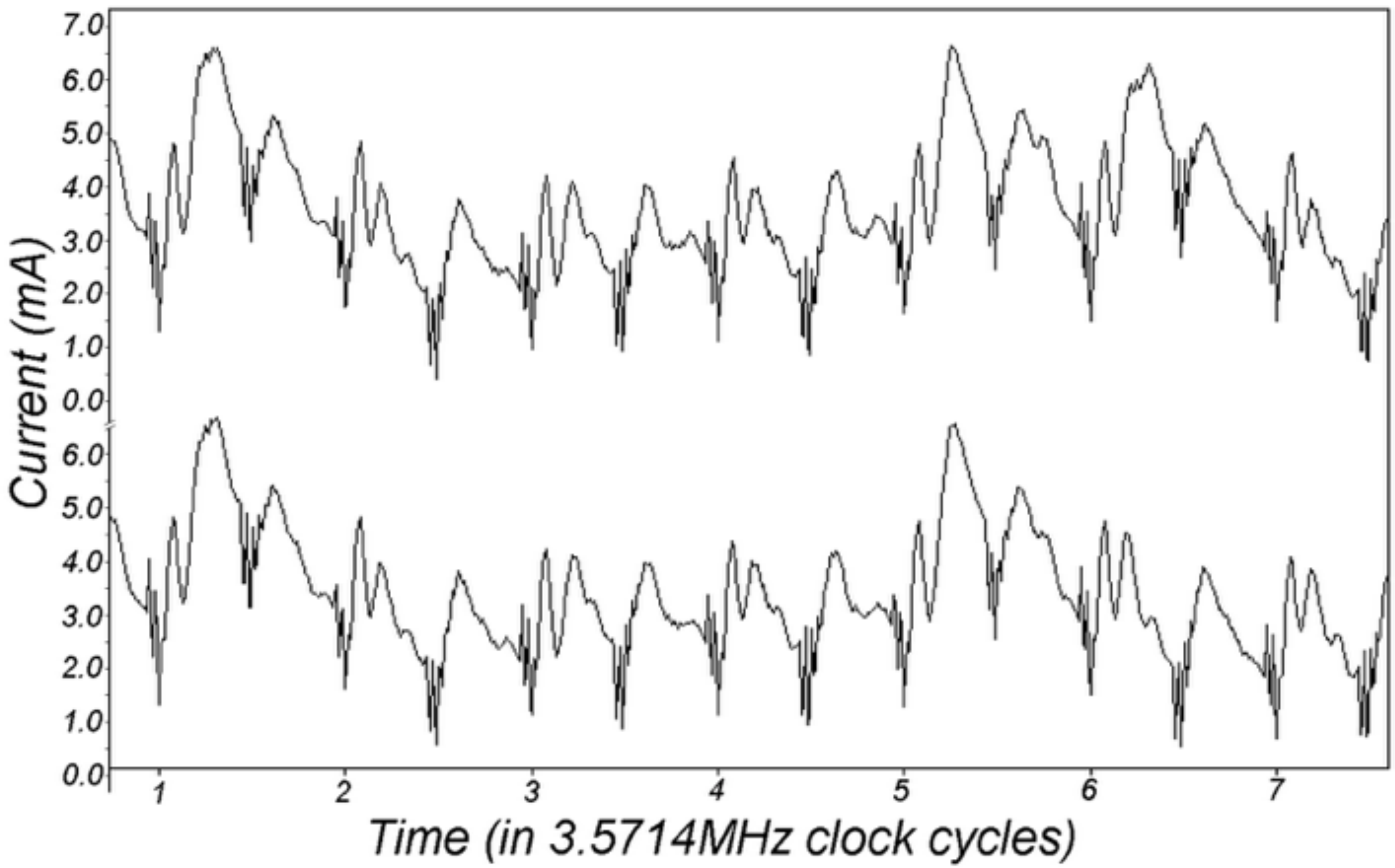


Figure 3: SPA trace showing individual clock cycles.

Specific **instructions** are also discernible

SPA on DES

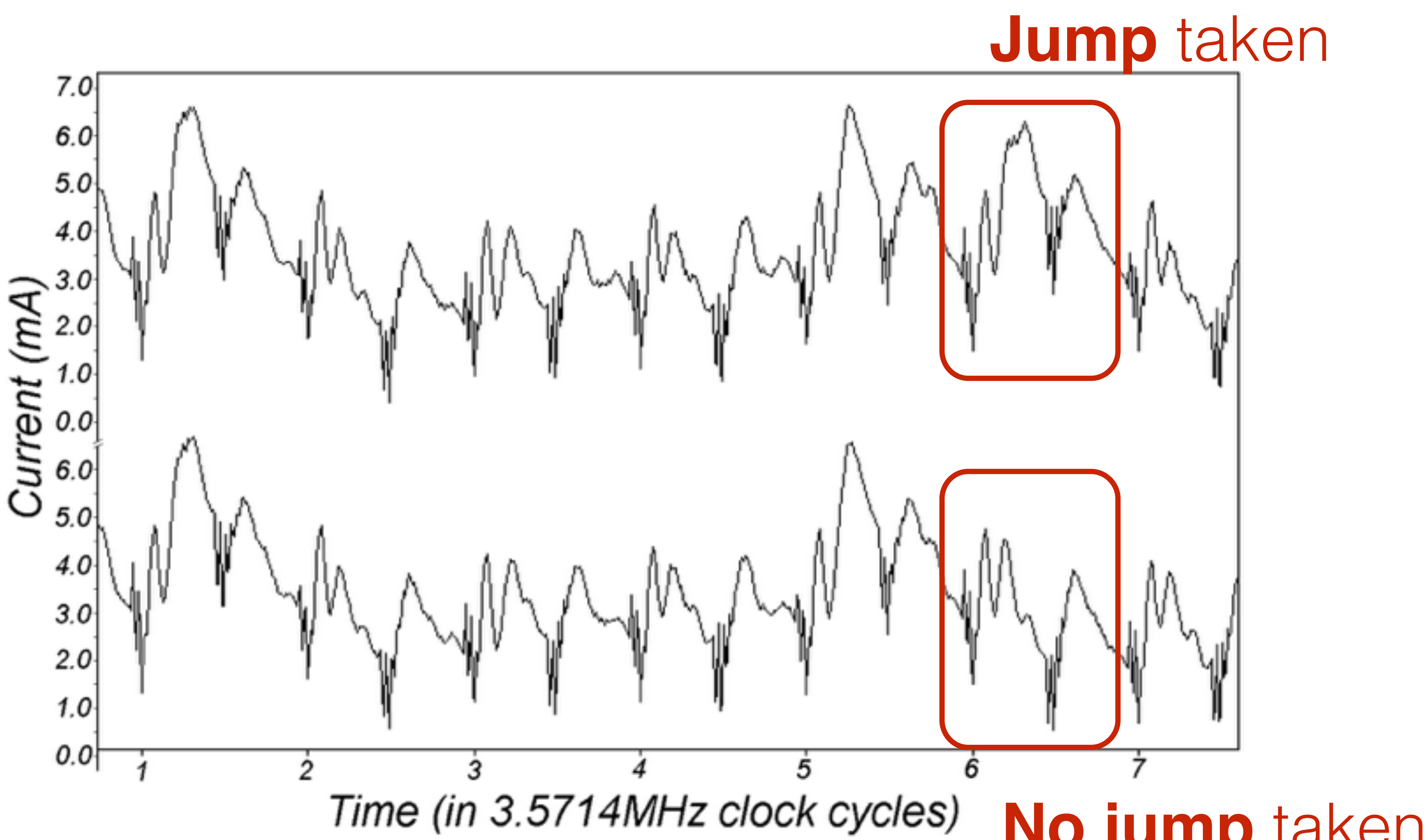


Figure 3: SPA trace showing individual clock cycles.

Specific **instructions** are also discernible

High-level idea

```
HypotheticalEncrypt(msg, key) {
   for(int i=0; i < key.len(); i++) {
      if(key[i] == 0)
         // branch 0
      else
         // branch 1
   }
}
```

High-level idea

```
HypotheticalEncrypt(msg, key) {
  for(int i=0; i < key.len(); i++) {
    if(key[i] == 0)
      // branch 0
    else
      // branch 1
  }
}
```

What if branch 0 had, e.g., a `jmp` that brand 1 didn't?

High-level idea

```
HypotheticalEncrypt(msg, key) {
  for(int i=0; i < key.len(); i++) {
    if(key[i] == 0)
      // branch 0
    else
      // branch 1
  }
}
```

What if branch 0 had, e.g., a `jmp` that brand 1 didn't?

Implementation issue: If the execution path depends on the inputs (key/data), then *SPA can reveal keys*

High-level idea

```
HypotheticalEncrypt(msg, key) {
  for(int i=0; i < key.len(); i++) {
    if(key[i] == 0)
      // branch 0
    else
      // branch 1
  }
}
```

What if branch 0 had, e.g., a `jmp` that brand 1 didn't?

What if branch 0

Implementation issue: If the execution path depends on the inputs (key/data), then *SPA can reveal keys*

High-level idea

```
HypotheticalEncrypt(msg, key) {
  for(int i=0; i < key.len(); i++) {
    if(key[i] == 0)
      // branch 0
    else
      // branch 1
  }
}
```

What if branch 0 had, e.g.,
a `jmp` that brand 1 didn't?

What if branch 0
- took longer? (timing attacks)

Implementation issue: If the execution path depends on the inputs (key/data), then *SPA can reveal keys*

High-level idea

```
HypotheticalEncrypt(msg, key) {
  for(int i=0; i < key.len(); i++) {
    if(key[i] == 0)
      // branch 0
    else
      // branch 1
  }
}
```

What if branch 0 had, e.g., a `jmp` that brand 1 didn't?

What if branch 0
- took longer? (timing attacks)
- gave off more heat?

Implementation issue: If the execution path depends on the inputs (key/data), then *SPA can reveal keys*

High-level idea

```
HypotheticalEncrypt(msg, key) {
  for(int i=0; i < key.len(); i++) {
    if(key[i] == 0)
      // branch 0
    else
      // branch 1
  }
}
```

What if branch 0 had, e.g., a **jmp** that brand 1 didn't?

What if branch 0
- took longer? (timing attacks)
- gave off more heat?
- made more noise?
- …

Implementation issue: If the execution path depends on the inputs (key/data), then *SPA can reveal keys*

Differential Power Analysis (DPA)

- SPA just visually inspects a single run
- DPA runs iteratively and reactively
  - Get multiple samples
  - Based on these, construct new plaintext messages as inputs, and repeat

Mitigating such attacks

- Hide information by making the execution paths depend on the inputs as little as possible
  - Have to *give up some optimizations* that depend on particular bit values in keys
    - Some Chinese Remainder Theorem (CRT) optimizations permitted remote timing attacks on SSL servers

- The crypto community should seek to design cryptosystems under the assumption that some information is going to leak

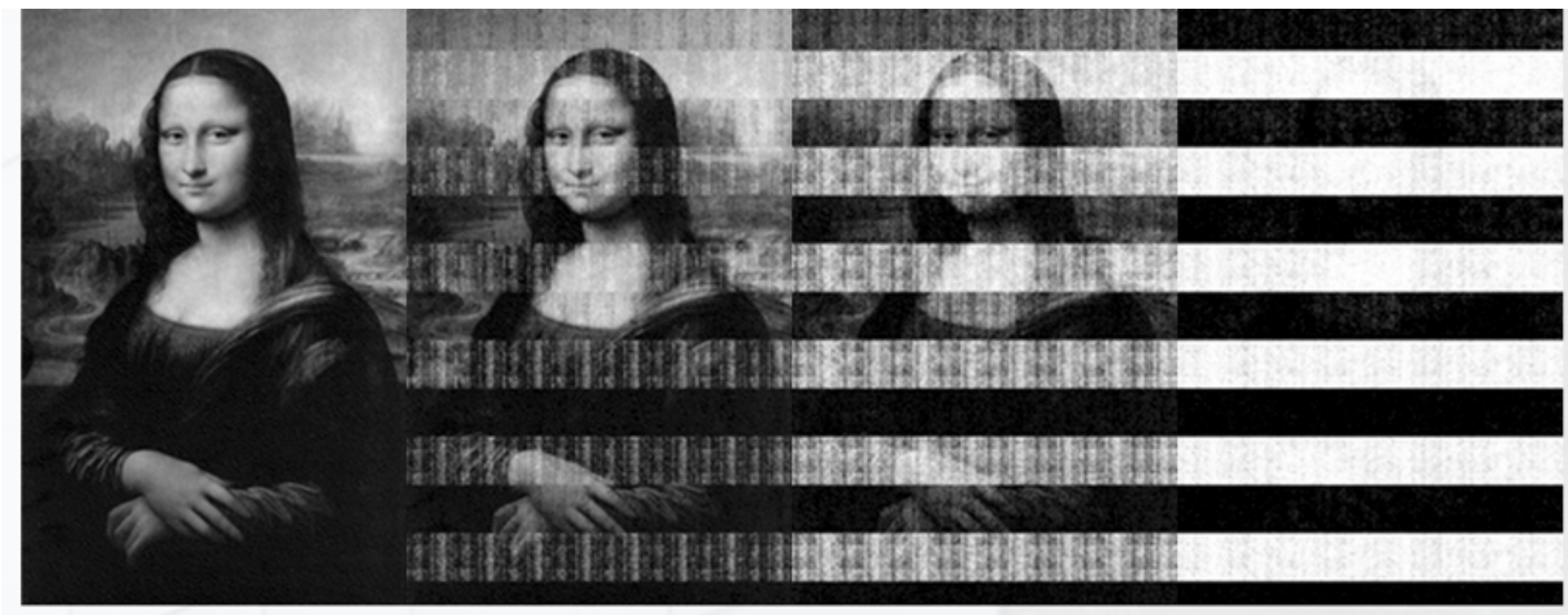
Other side-channel attacks

- Typical threat model: attacker doesn't have root access to a particular machine
  - So we safely store keys in memory
- But what if the attacker had physical access to the machine?

Attack

- Attacker's goal: reboot the machine into an OS that that he or she controls to look at memory contents

- Challenge: memory loses state without power



5 sec 30 sec 60 sec 5 min

Cold boot attack

Memory loses its state slower at really cold temperatures

	Seconds w/o power	Error % at operating temp.	Error % at $-50\,^{\circ}$C
A	60	41	(no errors)
	300	50	0.000095
B	360	50	(no errors)
	600	50	0.000036
C	120	41	0.00105
	360	42	0.00144
D	40	50	0.025
	80	50	0.18

Table 2: Effect of cooling on error rates

Cold boot attack

Memory loses its state slower at really cold temperatures

	Seconds w/o power	Error % at operating temp.	Error % at $-50\,^{\circ}C$
A	60	41	(no errors)
	300	50	0.000095
B	360	50	(no errors)
	600	50	0.000036
C	120	41	0.00105
	360	42	0.00144
D	40	50	0.025
	80	50	0.18

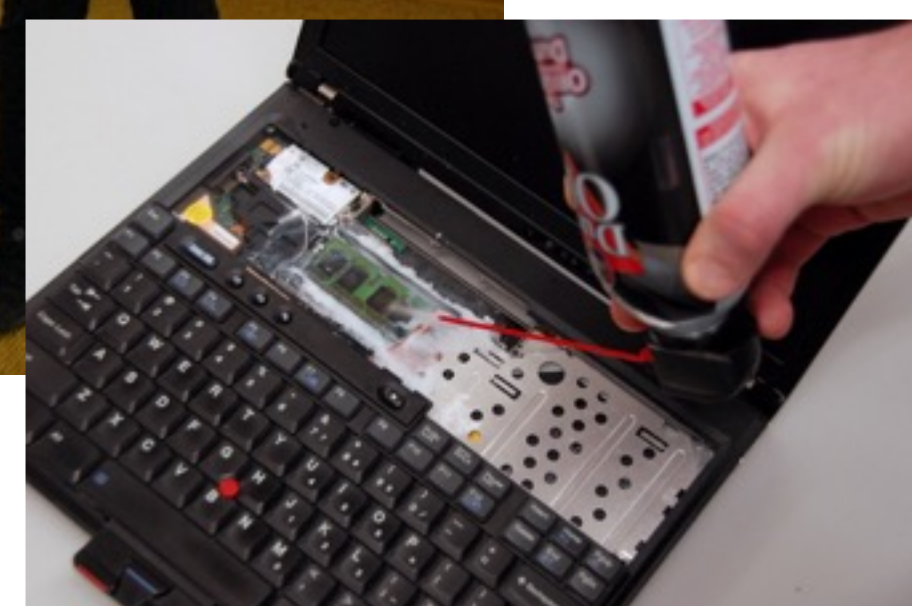
Table 2: Effect of cooling on error rates

Cold boot attack

Memory loses its state slower at really cold temperatures

	Seconds w/o power	Error % at operating temp.	Error % at −50°C
A	60	41	(no errors)
	300	50	0.000095
B	360	50	(no errors)
	600	50	0.000036
C	120	41	0.00105
	360	42	0.00144
D	40	50	0.025
	80	50	0.18

Table 2: Effect of cooling on error rates



Cold boot attack

- **Launching** the attack:
  - Cool down the memory & then power off/take it out
  - Boot into your own OS
  - Scan the memory image for keys (non-trivial but doable, especially if the keys have a format that's easy to detect)

- Some **defenses** against the attack:
  - Encrypt all of memory (increased CPU support for this)
  - Use trusted hardware (Xbox does this)
    - TPM (Trusted Platform Module) stores keys in hardware that is very difficult to inspect (some self-destruct)
  - Limit the amount of time keys live in memory
    - E.g., remove keys from memory when you enter Sleep mode

Misusing crypto

Avoid shooting yourself in the foot:

- Do not **roll your own** cryptographic mechanisms
  - Takes peer review
  - Apply Kerkhoff's principle
- Do not ***misuse*** existing crypto
- Do not even ***implement*** the underlying crypto

Ongoing work in crypto

- Constantly attacking/defending crypto schemes
    - Must stay on top of best practices
    - Ideally, write your code so you can change easily
- New mechanisms to permit new types of interactions
- A style of interaction that's been getting a lot of attention:
    - Alice has proprietary data
    - Bob has proprietary code (or computational resources)
    - Goal: Bob runs his code on Alice's data without learning her input *or* the output

Some things to look for

- Fully homomorphic encryption
  - Normal encryption: D(k, E(k, m)) = m
  - FHE: D(k, F(E(k, m))) = F(m)

- Secure multiparty computation
  - Alice and Bob both have data and want to know the output of F(Alice, Bob)
  - E.g., “which of us has more money” without having to reveal exactly how much either has