NTNU

Kunnskap for en bedre verden

# Side-Channel Attacks

Tjerand Silde,
Department of Mathematics,
NTNU in Trondheim

NTNU

# About me

- Doing a integrated Ph.D. in cryptography at IMF
- Focusing on Post-Quantum Cryptography
- Internship at Intrinsic-ID in Eindhoven, Netherlands summer of 2018: focusing on secure implementation of cryptography in hardware

NTNU

# Side-Channel Attacks

Cryptographic algorithms are traditionally designed and analyzed using the black-box model, where the adversary knows the specification of the algorithm and can observe pairs of inputs and outputs from the algorithm.

This is not sufficient to protect a system when it's implemented on embedded devices. In this case the device is often in the control of the adversary, and one can apply a wide range of attacks through different side-channels to break the cryptography.

NTNU

# Countermeasures

Countermeasures against side-channel attacks can be classified into two categories.

In the first category, one tries to eliminate or to minimize the leakage of information. This is achieved by reducing the signal-to-noise ratio of the side-channel signals.

In the second category, one tries to ensure that the information that leaks through side-channels cannot be exploited to recover secrets.

NTNU

# Attacks against primitives

- RSA
- DSA
- ECDSA
- ECDH
- AES
- ...

NTNU

# Today: The RSA Cryptosystem

NTNU

# RSA Key Generation

1. Let $p, q$ be distinct prime numbers, randomly chosen from the set of all prime numbers of a certain size.
2. Compute $n = pq$.
3. Select $e$ randomly with $\gcd(e, \phi(n)) = 1$.
4. Compute $d = e^{-1} \bmod \phi(n)$.
5. The public key is the pair $n$ and $e$.
6. The private key consists of the values $p$, $q$ and $d$.

# RSA Operations

**Encryption** The public key for encryption is $K_E = (n, e)$

1. Input is any value $M$ where $0 < M < n$.
2. Compute $C = E(M, K_E) = M^e \bmod n$.

**Decryption** The private key for decryption is $K_D = d$ (values $p$ and $q$ are not used here).

1. Compute $D(C, K_D) = C^d \bmod n = M$.

# Square-And-Multiply Algorithm

$$m^e = m^{e_0}(m^2)^{e_1}(m^4)^{e_2}\ldots(m^{2^k})^{e_k}$$

**Data:** $m, n, e = e_k e_{k-1} \ldots e_1 e_0$
**Result:** $m^e \bmod n$
$z \leftarrow 1$;
**for** $i = 0$ to $k$ **do**
  **if** $e_i = 1$ **then**
    $z \leftarrow z * m \bmod n$;
  **end**
  **if** $i < k$ **then**
    $m \leftarrow m^2 \bmod n$;
  **end**
**end**
**return** $z$

NTNU

# Square-And-Multiply

```
# compute m^e mod n
def square_and_multiply(m, exponent, n):

    e = bin(exponent) # string with prefix 0b
    z = m

    for i in range(3, len(exp)):

        z = z * z % n

        if( e[i] == '1' ):

            z  = z * m % n

    return z
```

NTNU

# Timing attacks

## Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems

Paul C. Kocher

Cryptography Research, Inc.
607 Market Street, 5th Floor, San Francisco, CA 94105, USA.
E-mail: paul@cryptography.com.

In proceedings of the 16th Annual International Cryptology Conference, 1996.
https://www.paulkocher.com/doc/TimingAttacks.pdf

NTNU

# Timing attacks

**Remote Timing Attacks are Practical**

David Brumley
*Stanford University*
dbrumley@cs.stanford.edu

Dan Boneh
*Stanford University*
dabo@cs.stanford.edu

In proceedings of the 12th Usenix Security Symposium, 2003.
https://crypto.stanford.edu/~dabo/pubs/papers/ssl-timing.pdf

# Timing attacks

We show that using about a million queries we can remotely extract a 1024-bit RSA private key from an OpenSSL 0.9.7 server. The attack takes about two hours.

In proceedings of the 12th Usenix Security Symposium, 2003.
https://crypto.stanford.edu/~dabo/pubs/papers/ssl-timing.pdf

NTNU

# Timing attacks

Timing attacks try to extract cryptographic keys by measuring how long it takes for a system to perform cryptographic computations.

If the algorithms use different amounts of time depending on the input and the keys, it is possible to extract secret information from the time the system takes to respond to different requests.

NTNU

# Simple Power Analysis

## Differential Power Analysis

Paul Kocher, Joshua Jaffe, and Benjamin Jun

Cryptography Research, Inc.
607 Market Street, 5th Floor
San Francisco, CA 94105, USA.
http://www.cryptography.com
E-mail: {paul,josh,ben}@cryptography.com.

In proceedings of the 19th Annual International Cryptology Conference, 1999.
https://www.paulkocher.com/doc/DifferentialPowerAnalysis.pdf

# Simple Power Analysis

Simple Power Analysis (SPA) is a technique that involves directly interpreting power consumption measurements collected during cryptographic operations. SPA can yield information about a device's operation as well as key material.

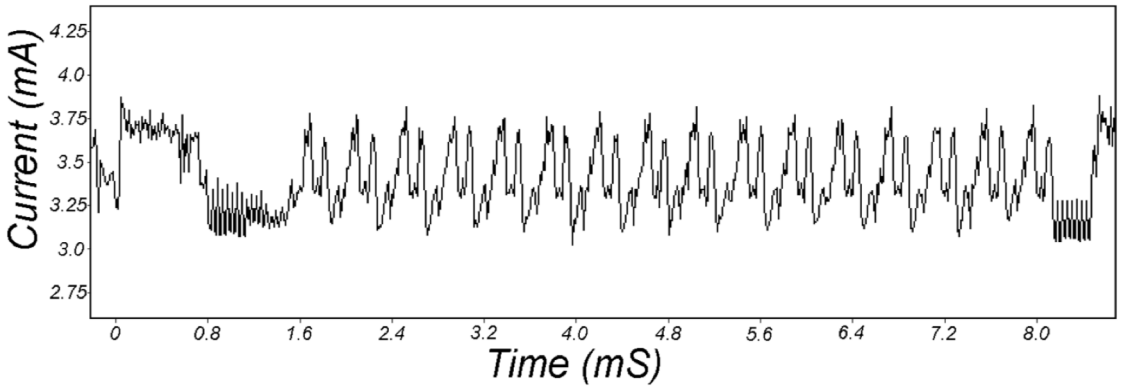



**Figure 1:** SPA trace showing an entire DES operation.

In proceedings of the 19th Annual International Cryptology Conference, 1999.
https://www.paulkocher.com/doc/DifferentialPowerAnalysis.pdf

# Simple Power Analysis

Figure 3 shows the execution path through an SPA feature where a jump instruction is performed, and the lower trace shows a case where the jump is not taken. The point of divergence is at clock cycle 6 and is clearly visible.

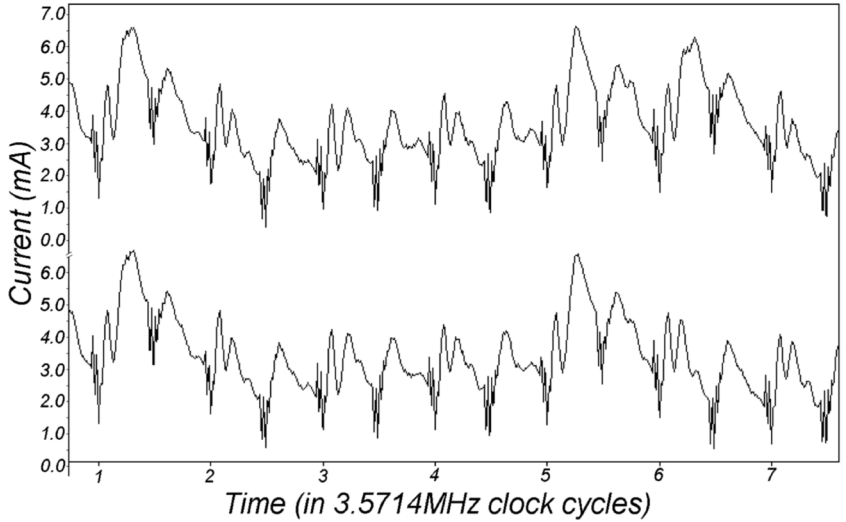



**Figure 3:** SPA trace showing individual clock cycles.

In proceedings of the 19th Annual International Cryptology Conference, 1999.
https://www.paulkocher.com/doc/DifferentialPowerAnalysis.pdf

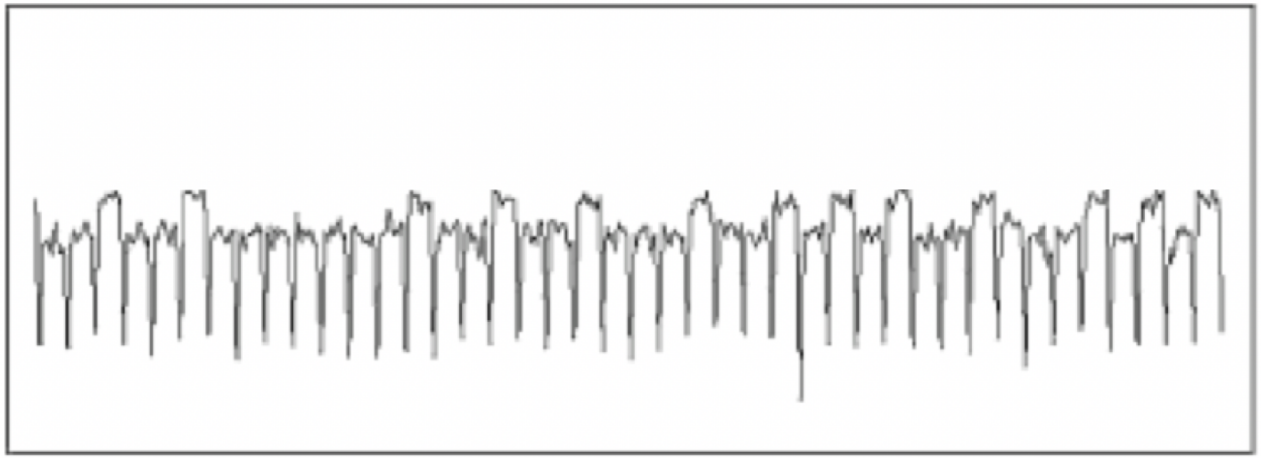
NTNU

# Simple Power Analysis

Simple power analysis (SPA) attacks try to extract cryptographic keys by measuring the power consumption of a device while the system is doing cryptographic computations.

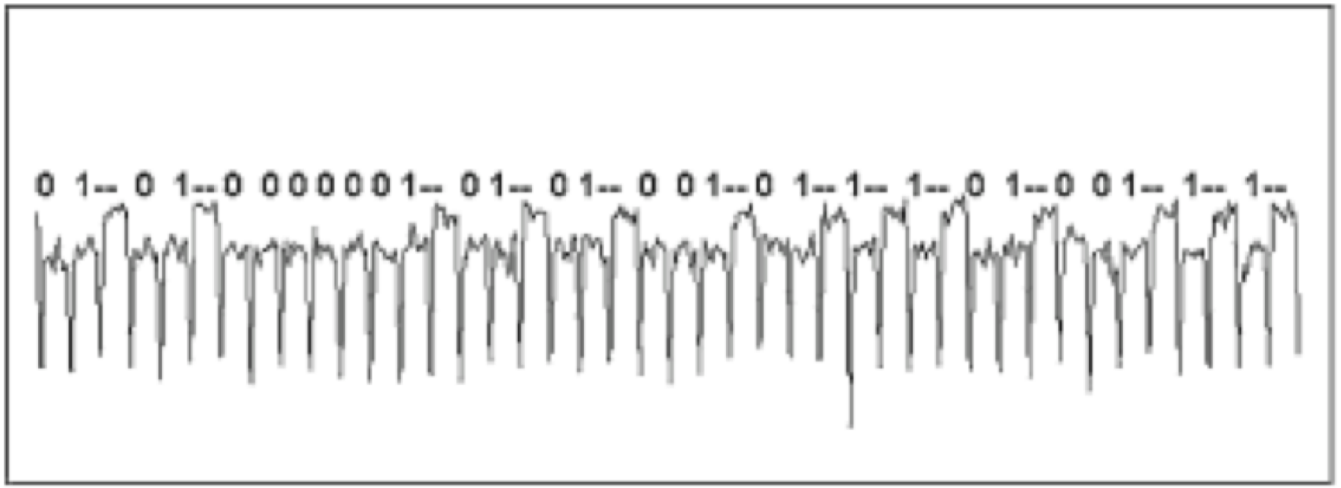
NTNU

# Example: Simple power analysis



NTNU

# Example: Simple power analysis



NTNU

# Countermeasure: Constant Time Code

Make it so, that the variations in execution time do not depend on secret information.

NTNU

```
# compute m^e mod n in constant time
def square_and_multiply(m, exponent, n):

    e = bin(exponent) # string with prefix 0b
    z = m
    x = 0

    for i in range(3, len(e)):

        z = z * z % n

        if( e[i] == '1' ):

            z  = z * m % n

        if( e[i] == '0' ):

            x  = z * m % n

    return z
```

NTNU

# Fault attacks

## Checking Before Output May Not Be Enough Against Fault-Based Cryptanalysis

[Published in *IEEE Transactions on Computers* **49**(9):967-970, 2000.]

Sung-Ming Yen[1] and Marc Joye[2]

[1] Laboratory of Cryptography and Information Security (LCIS)
National Central University, Chung-Li, Taiwan 320, R.O.C.
yensm@csie.ncu.edu.tw
[2] Gemplus Card International
Parc d'Activités de Gémenos, B.P. 100, 13881 Gémenos, France
marc.joye@gemplus.com

IEEE Transactions on Computers, 2000.
http://joye.site88.net/papers/YJ00chkb.pdf

NTNU

# Fault attacks

**Abstract.** In order to avoid fault-based attacks on cryptographic security modules (e.g., smart-cards), some authors suggest that the computation results should be checked for faults before being transmitted. In this paper, we describe a potential fault-based attack where key bits leak only through the information whether the device produces after a temporary fault a correct answer or not. This information is available to the adversary even if a check is performed before output.

IEEE Transactions on Computers, 2000.
http://joye.site88.net/papers/YJ00chkb.pdf

# Fault attacks

It is important to note that when an error is introduced to register $A$ during the operation $A \leftarrow A^2 \bmod N$, it will force the squaring operation to be incorrect. This is evident because the correct value of $A$ is required during each iteration of the interleaved modular multiplication procedure. However, if a *safe error* is introduced into $A$ during the operation $A \leftarrow A \cdot M \bmod N$, then this error will not damage the final result. The above attack is sketched hereafter.

IEEE Transactions on Computers, 2000.
http://joye.site88.net/papers/YJ00chkb.pdf

NTNU

# Countermeasure: Data-Dependency

Make it so, that the final result is dependent on each calculation in the computation.

NTNU

```python
# compute m^e mod n in constant time
# using Montgomery ladder
def square_and_multiply(m, exponent, n):

    e = bin(exponent)           # string with prefix 0b
    z1 = m
    z2 = m*m

    for i in range(3, len(e)):

        if( e[i] == '1' ):

            z1  = z1 * z2 % n   # z1 = z1 * z2
            z2 = z2 * z2 % n    # z2 = z2 * z2

        if( e[i] == '0' ):

            z2  = z1 * z2 % n   # z2 = z1 * z2
            z1 = z1 * z1 % n    # z1 = z1 * z1

    return z1
```

NTNU

# Differential Power Analysis

## Differential Power Analysis

Paul Kocher, Joshua Jaffe, and Benjamin Jun

Cryptography Research, Inc.
607 Market Street, 5th Floor
San Francisco, CA 94105, USA.
http://www.cryptography.com
E-mail: {paul,josh,ben}@cryptography.com.

In proceedings of the 19th Annual International Cryptology Conference, 1999.
https://www.paulkocher.com/doc/DifferentialPowerAnalysis.pdf

# Differential Power Analysis

To implement the DPA attack, an attacker first observes $m$ encryption operations and captures power traces $\mathbf{T}_{1..m}[1..k]$ containing $k$ samples each. In addition, the attacker records the ciphertexts $C_{1..m}$. No knowledge of the plaintext is required.

DPA analysis uses power consumption measurements to determine whether a key block guess $K_s$ is correct. The attacker computes a $k$-sample differential trace $\Delta_D[1..k]$ by finding the difference between the average of the traces for which $D(C, b, K_s)$ is one and the average of the traces for which $D(C, b, K_s)$ is zero.

In proceedings of the 19th Annual International Cryptology Conference, 1999.
https://www.paulkocher.com/doc/DifferentialPowerAnalysis.pdf

NTNU

# Differential Power Analysis

Differential power analysis (DPA) attacks try to extract cryptographic keys by performing a statistical analysis of many executions of the same algorithm.

The attack is performed with many different inputs measuring the differences in power consumption of a device while the system is performing cryptographic computations.

NTNU

# Countermeasure: Randomizing Message

Make it so, that the calculations are performed on a randomly chosen message each time, not an input by the attackers choice.

```
# compute m^e mod n in constant time using Montgomery
# ladder with randomized message, given r1 and r2
# such that r1 = ( r2^-1 mod n )^e mod n
def square_and_multiply(m, exponent, r1, r2, p, q):

    n = p*q
    m = m*r1                    # randomize message
    e = bin(exponent)           # string with prefix 0b
    z1 = m
    z2 = m*m

    for i in range(3, len(e)):

        if( e[i] == '1' ):

            z1  = z1 * z2 % n   # z1 = z1 * z2
            z2 = z2 * z2 % n    # z2 = z2 * z2

        if( e[i] == '0' ):

            z2  = z1 * z2 % n   # z2 = z1 * z2
            z1 = z1 * z1 % n    # z1 = z1 * z1

    return z1*r2                # de-randomize message
```

NTNU

# Countermeasure: Scalar Blinding

Make it so, that the calculations are performed with a randomly chosen exponent each time, to not leak to the attacker how large the real exponent is.

```
# compute m^e mod n in constant time using Montgomery
# ladder with randomized message, given r1 and r2
# such that r1 = ( r2^-1 mod n )^e mod n, and
# randomized exponend, given some r
def square_and_multiply(m, exponent, r, r1, r2, p, q):

    n = p*q
    m = m*r1                        # randomize message
    exponent += r * (p-1)*(q-1) # exp = exp + r * phi(n)
    e = bin(exponent)               # string with prefix 0b
    z1 = m
    z2 = m*m

    for i in range(3, len(e)):

        if( e[i] == '1' ):

            z1  = z1 * z2 % n   # z1 = z1 * z2
            z2 = z2 * z2 % n    # z2 = z2 * z2

        if( e[i] == '0' ):

            z2  = z1 * z2 % n   # z2 = z1 * z2
            z1 = z1 * z1 % n    # z1 = z1 * z1

    return z1*r2                    # de-randomize message
```

# But there are also...

- Cache timing attacks
- Electromagnetic attacks
- Acoustic cryptanalysis
- Optical side-channel attack
- Horizontal attacks
- …

NTNU

# Questions?