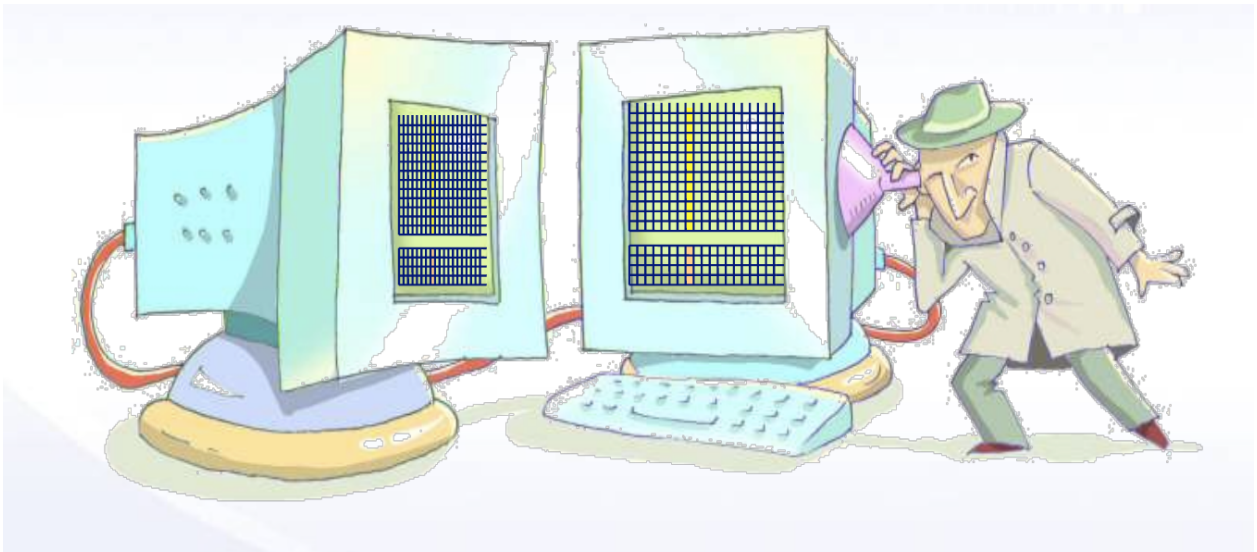
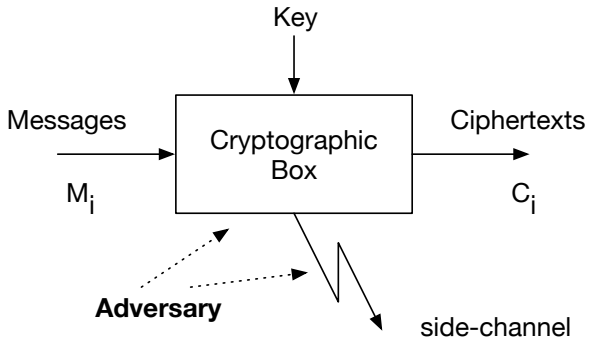
Side-Channel Attacks and Countermeasures

Çetin Kaya Koç



Contents

Side-Channel Cryptanalysis



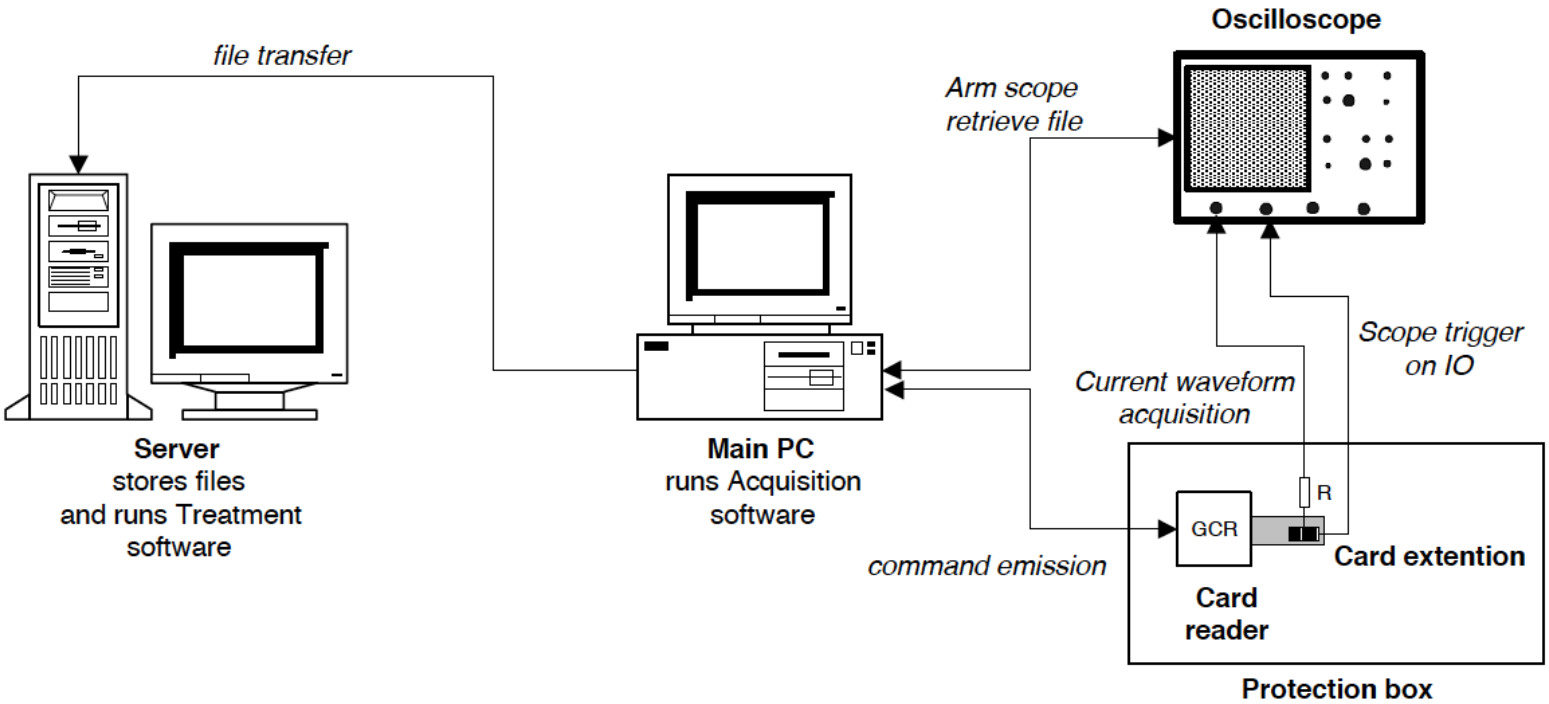
- Cryptographic algorithms must run on a real device
- Devices have physical properties
- Devices will emanate information regarding cryptographic algorithm, key, and message
- Adversary having access to these side channels will extract information
  - Timing
  - Power
  - Electromagnetic
  - Acoustic

Side-Channel Cryptanalysis

- A new area of applied cryptography
- The study of breaking cryptosystems using side-channel information

- **Timing attacks** exploit time differences occurring for various input values
- **Power attacks** exploit the instantaneous power consumption during critical phases of the cryptographic code
- **Electromagnetic attacks** exploit the instantaneous electromagnetic emanations during critical phases of the cryptographic code

Equipment Setup for Power and Timing Analysis

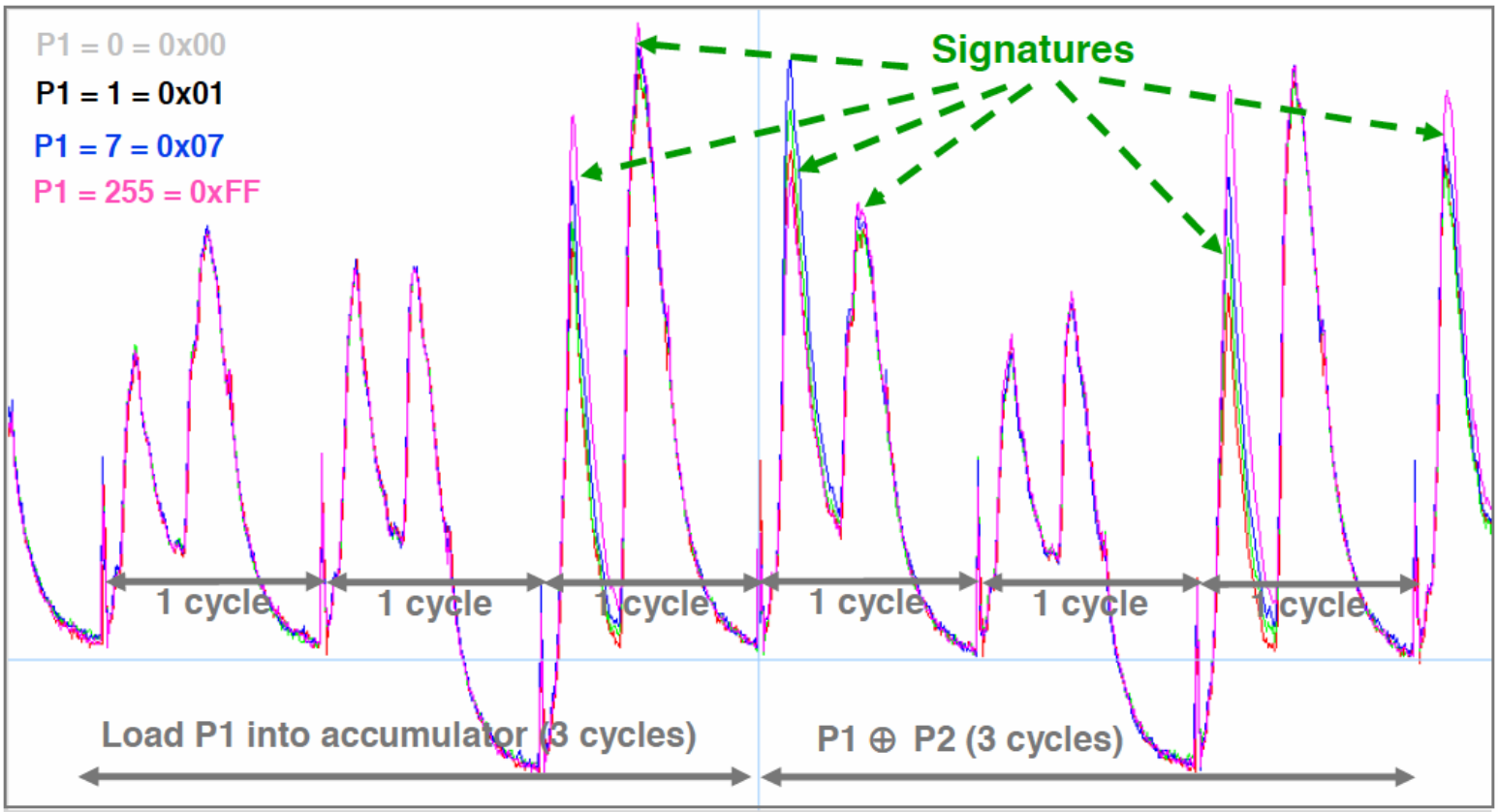


Information Leakage

- The power consumption of a chip depends on the manipulated data and the executed instruction
- Information leakage model (assumption): The consumed power is related to the Hamming weight of data (or address, op code)
- $H(0) = 0$
- $H(1) = H(2) = H(4) = H(8) = \cdots = 1$
- $H(3) = H(5) = H(6) = H(9) = \cdots = 2$
- $\cdots$
- $H(P_i \oplus P_{i-1})$

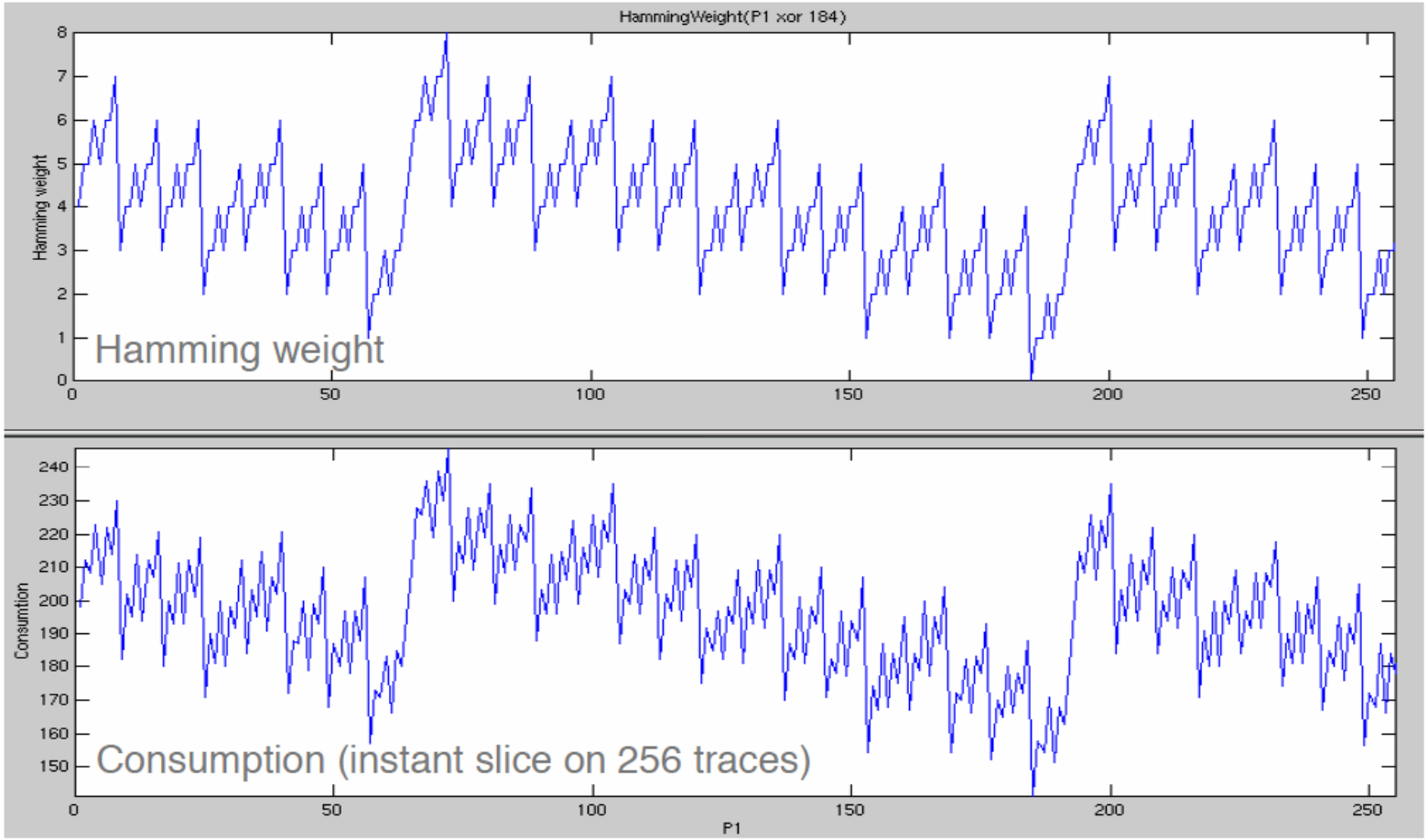
Information Leakage

- Load P_1 and XOR with $P_2 = 0$ such that $P_1 = 0, 1, 7, 255$



Information Leakage

- $H(P_1 \oplus 184)$ for $P_1 = 0, 1, 2, \ldots, 255$



Simple Power Analysis (SPA)

- The objective is to find the secret or private key
- Algorithm is known
- Implementation is unknown however some background is available
- Reverse engineering is required
- A single power curve may be sufficient
- A known plaintext, ciphertext pair may be required

SPA Attack on RSA Signature Operation

- The signature computation

$$s = \mu(m)^d \pmod{n}$$

- n is large modulus, say 1024 bits or more
- m is the message
- $\mu(m)$ is the padded and hashed message
- s is the signature
- d is the private key such that $e \cdot d = 1 \pmod{\phi(n)}$
- The attacker aims to obtain d

SPA Attack on RSA Signature Operation

- Implementation details:
  - n, m, s, and d are 128-byte buffers
  - the binary method of exponentiation
  - the exponent bits are scanned from MSB to LSB
  - k is the bit size of d

Input: m, $d = (d_{k-1}, \ldots, d_0)_2$, n
Output: $s = m^d \pmod{n}$

1. $s \leftarrow 1$

2. For $i = k - 1$ downto 0

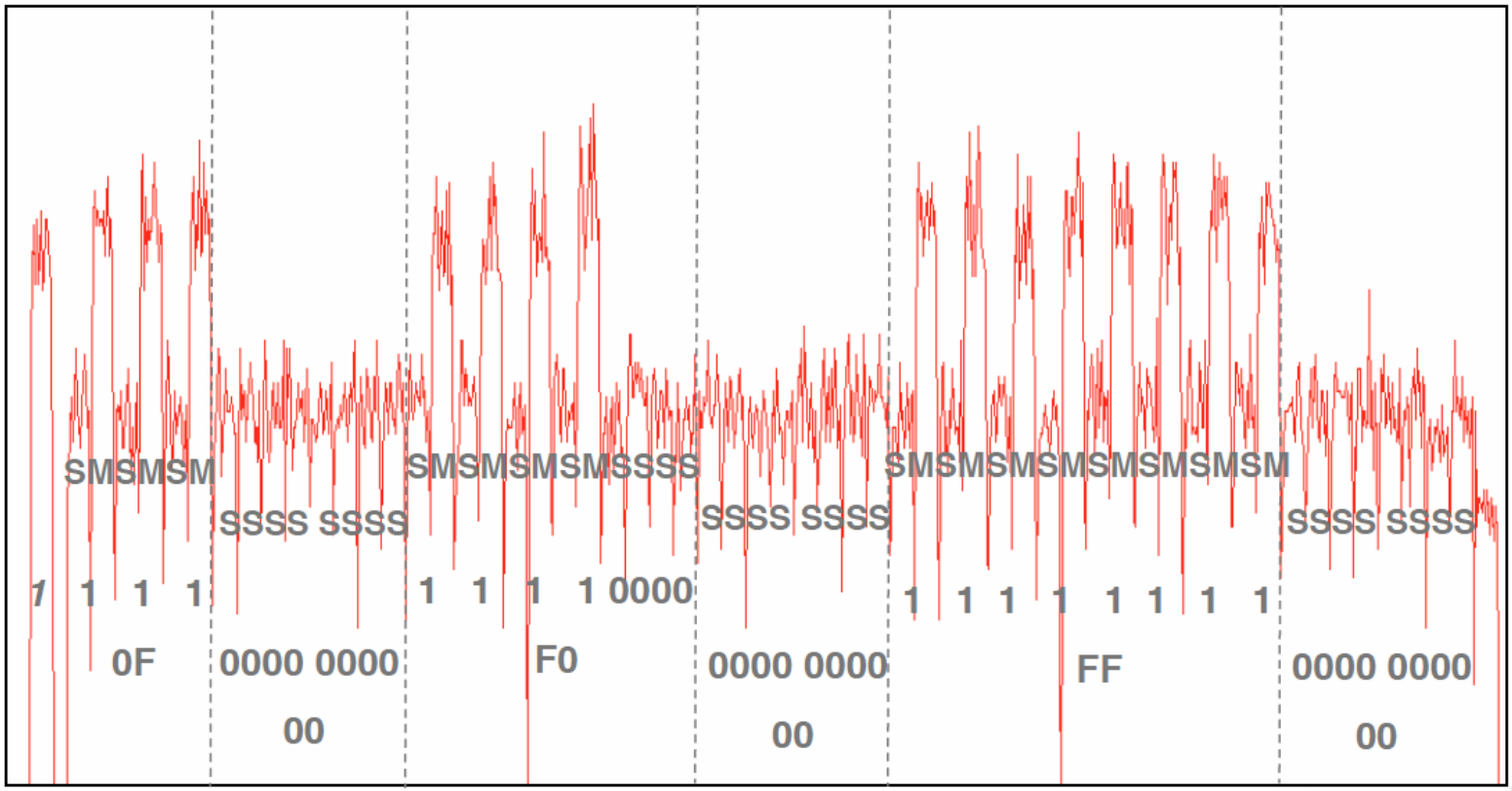
$s \leftarrow s \cdot s \pmod{n}$

If $d_i = 1$ then $s \leftarrow s \cdot m \pmod{n}$

3. Return s

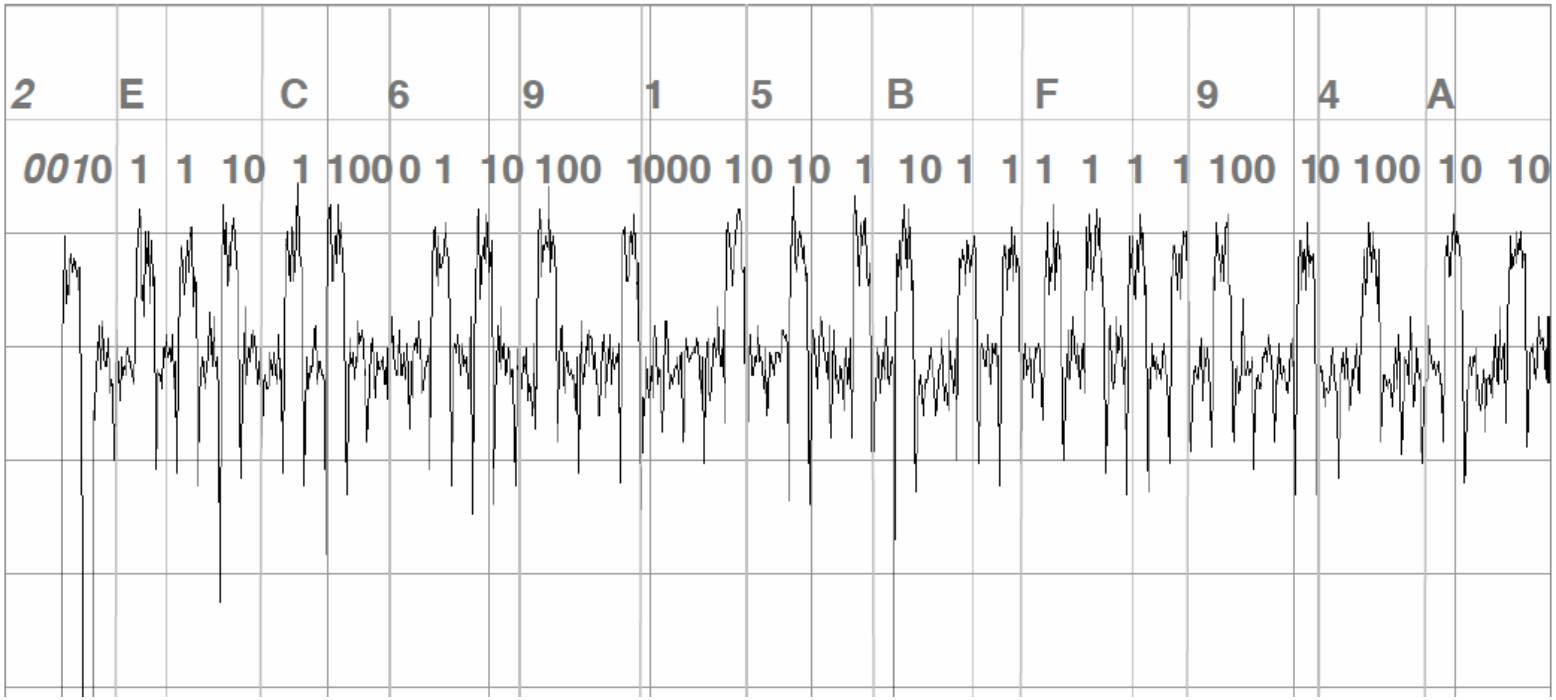
SPA Attack on RSA Signature Operation

- Test key value: 0F 00 F0 00 FF 00



SPA Attack on RSA Signature Operation

- Test key value: 2E C6 91 5B F9 4A



SPA Attack on RSA Signature Operation

- SPA uses implementation details
- SPA requires:
  - algorithm knowledge,
  - reverse engineering,
  - representation tuning, and
  - playing with implementation assumptions
- SPA depends on
  - Algorithm implementation
  - Application constraints
  - The technology (electrical properties) of the chip
  - Possible countermeasures

Countermeasures Against SPA Attack

- What is a countermeasure? Anything that foils the attack
- Basic countermeasure: remove code branches that depend on secret or private key bits
- Advanced countermeasure:
  - Algorithm specification refinement
  - Data whitening (blinding)
  - Make changes in the instruction set
  - Electrical behavior changes (current scramblers, coprocessor usage)

Timing Attacks

- Processing time depends on the value of the secret key bit
- It leaks information about it
- There are ways to measure it
- Timing attack conditions
  - The processing should be monitored
  - Processing durations need to be recorded
  - Some basic computational and statistical tools are needed
  - Knowledge of the implementation will be required

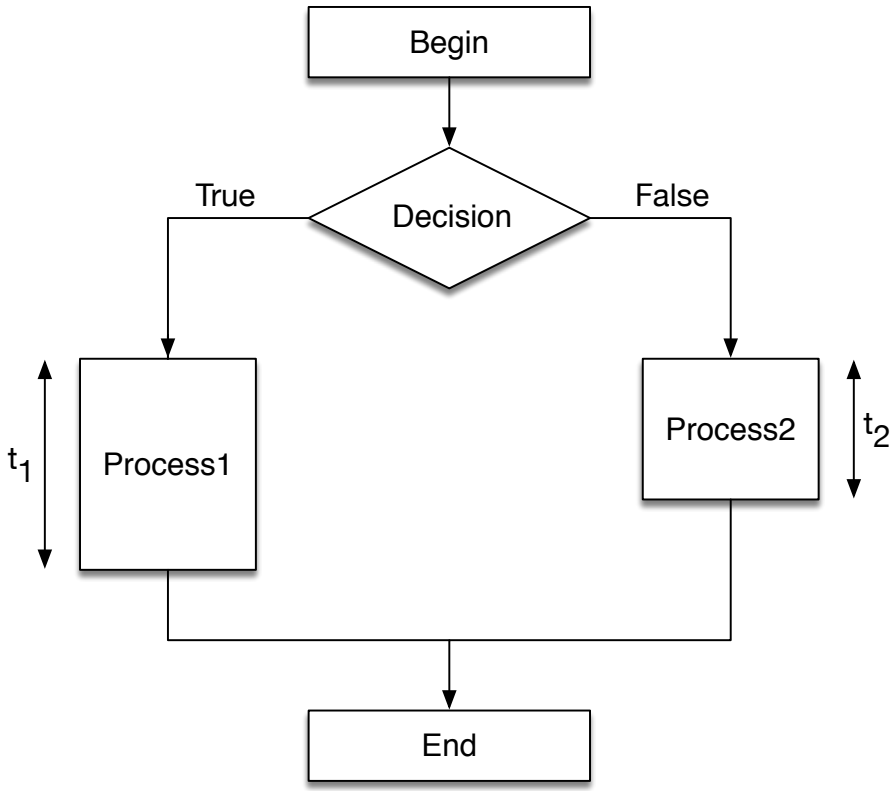
Timing Attacks

The code starts unconditionally

The test is based on secret bit

Depending on the Boolean condition the process may be long (t_1) or short (t_2)

The code continues unconditionally



Timing Attacks

- The term "Timing Attack" was introduced by Paul Kocher in 1996
- First practical attacks in Crypto 1997 Conference
- Applicable to RSA and in fact all cryptosystems
  - Basic mathematical operations
  - Modular exponentiation
  - Cryptographic algorithms
- Knowledge and variability of messages are needed
- Time measurements must be accurate to within few clock cycles

Attacking RSA Algorithm

- The standard RSA exponentiation $s = m^d \pmod{n}$
- The Montgomery method for modular multiplication
- Timing variations in Montgomery due to Subtraction Step
- The binary method of exponentiation yields bits of d

The Binary Method of Exponentiation

- Input: m, d, n
- m: message which is k bits
- (d, n): the RSA private key, k bits each
- Output: $s = m^d \pmod{n}$
- m: signature which is k bits

Input: $m,\ d = (d_{k-1}, \ldots, d_0)_2,\ n$
Output: $s = m^d \pmod{n}$

1. $s \leftarrow 1$

2. For $i = k - 1$ downto 0

$s \leftarrow s \cdot s \pmod{n}$

If $d_i = 1$ then $s \leftarrow s \cdot m \pmod{n}$

3. Return s

The Montgomery Modular Multiplication

- The Montgomery modular multiplication ($\odot$) is a special, high-speed modular multiplication algorithm
- It is significantly faster than Multiply-and-Reduce algorithm
- It produces a result in the range $[0, 2n)$
- A subtraction may be required to fully reduce mod n

- Multiply step for bit d_i
- If $d[i] = 1$ then $s = s \odot m \pmod{n}$
- Step 1: The multiply-add steps of Montgomery multiplication
- Step 2: The optional subtraction by n (if the result $> n$)

Attacking RSA Algorithm

- Assume, higher $(i-1)$ bits of the exponent d are already known
- That is, we know $d[k-1], d[k-2], \ldots, d[k-(i-1)]$
- Knowing the message m, we can compute the intermediate value of the signature s after the square operation for index $(k-i)$
- We can also determine whether the Montgomery multiplication operation $s \odot m \pmod{n}$ will cause a subtraction
- However, we do not know the value of the bit $d[k-i]$
  - If $d[k-i] = 0$, there will not be a Montgomery multiplication (and thus no subtraction either)
  - if $d[k-i] = 1$, there will be a Montgomery multiplication and we have determined whether there will be a subtraction or not

Description of the Attack

- Sign L messages using (d, n) and obtain signatures and timings
- Let $\mathcal{S}$ the set of messages: $\mathcal{S} = \{m_1, m_2, \ldots, m_L\}$
- Let $\mathcal{T}$ the set of timings: $\mathcal{S} = \{t_1, t_2, \ldots, t_L\}$ such that t_i is the timing for processing the message m_i
- Assume $d[k - i] = 1$
- Partition $\mathcal{S}$ into two disjoint subsets: $\mathcal{S}_0$ and $\mathcal{S}_1$ such that
- $\mathcal{S}_0 = \{m \ : \ s \odot \pmod n$ implies no subtraction$\}$
- $\mathcal{S}_1 = \{m \ : \ s \odot \pmod n$ implies subtraction$\}$
- Compute the mean time $\overline{\mathcal{T}}_0$ of the messages in $\mathcal{S}_0$
- Compute the mean time $\overline{\mathcal{T}}_1$ of the messages in $\mathcal{S}_1$

Description of the Attack

- Case $d[k-i]=0$
  Global times for sets $\mathcal{S}_0$ and $\mathcal{S}_1$ are not statistically distinguishable since the split is based on a multiplication which does not occur

- Case $d[k-i]=1$
  Global times for sets $\mathcal{S}_0$ and $\mathcal{S}_1$ show a statistical difference to the optional multiplication since it does occur

- Time measurements validate or invalidate the assumption:

  If $\overline{\mathcal{T}}_0 - \overline{\mathcal{T}}_1 \gg 0$, the assumption is valid, that is $d[k-i]=1$

  If $\overline{\mathcal{T}}_0 - \overline{\mathcal{T}}_1 \approx 0$, the assumption is wrong, that is $d[k-i]=0$

Conclusions

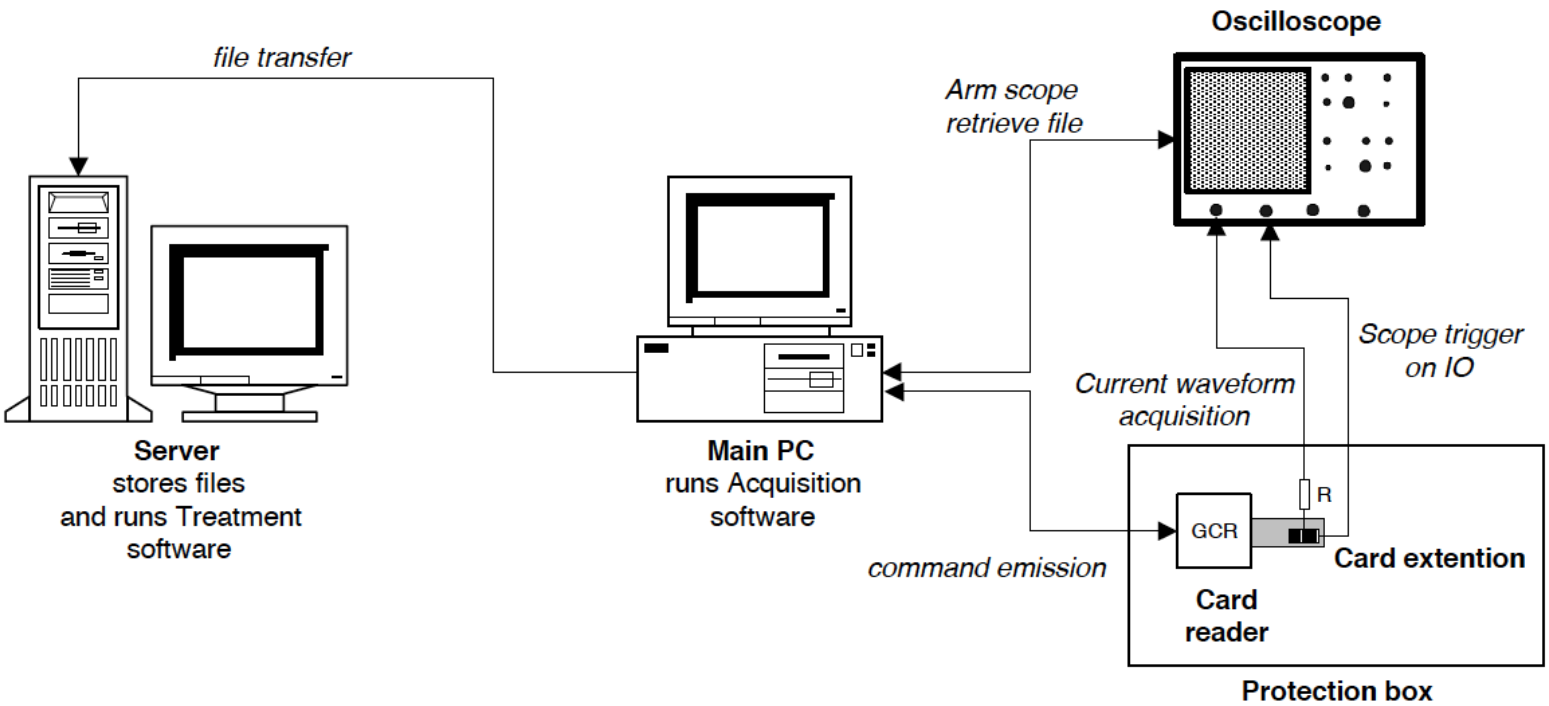
- For 128 bits, the attack recovers 2 bits/sec for $L = 10,000$
- For 512 bits, the attack recovers 1 bits/sec for $L = 100,000$
- Together with other side-channel attacks they become more efficient
- A real threat for many devices
- It works against computers, servers, not just smart cards

- There are solutions
- A basic countermeasure would be to create constant-time processing
- Blinding (whitening or randomization) approaches also work

Differential Power Analysis

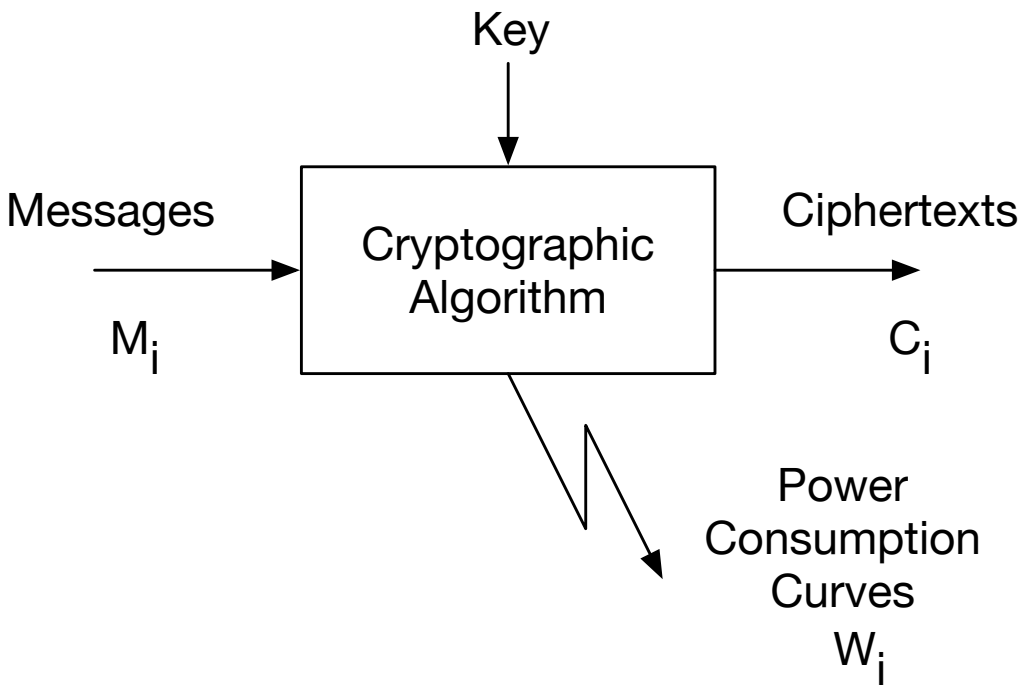
- Also invented by Paul Kocher (1998)
- A powerful and generic power attack
- DPA uses statistics and signal processing
- DPA requires known random messages
- DPA targets a known algorithm
- Applicable to a smart card
- Big noise in crypto community
- Big fear in the smart card industry

Acquisition Procedure



Acquisition Procedure

- Apply the algorithm L times, $10^3 < L < 10^5$



Selection and Prediction

- Assume the message is processed by a known deterministic function f (transfer, permutation)
- Knowing the message, one can recompute its image through f offline

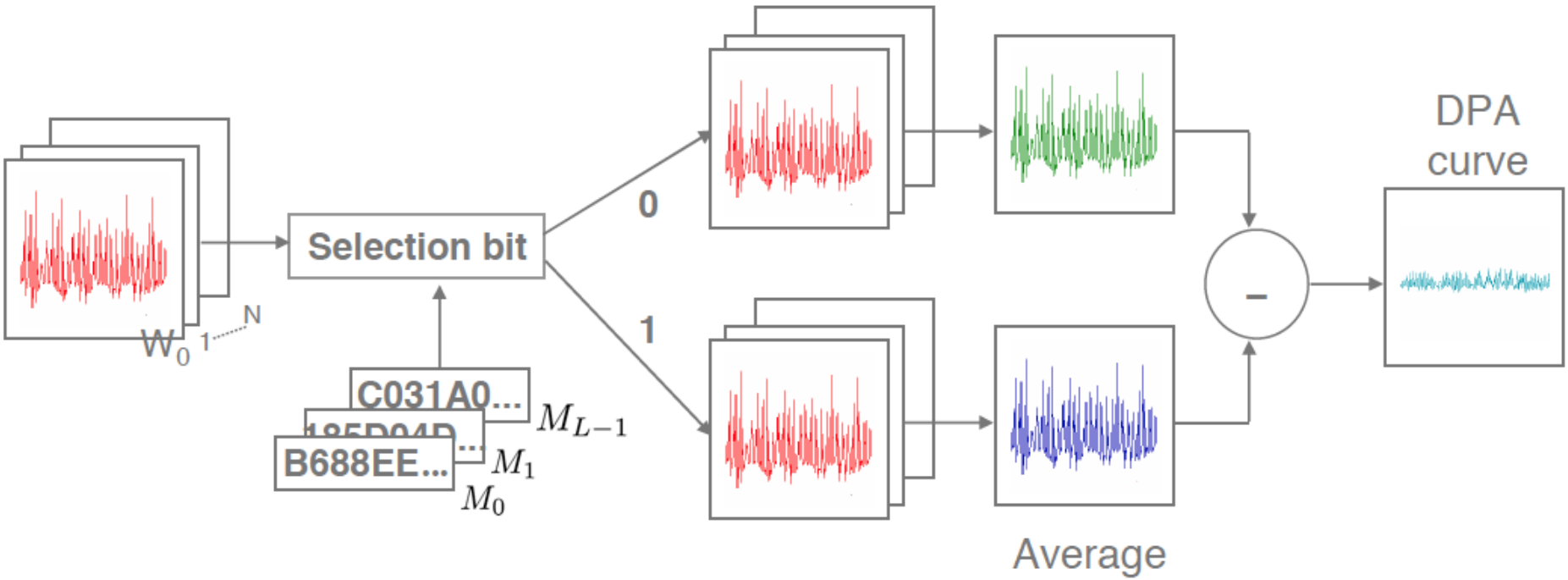
$$M_i \longrightarrow \boxed{\text{f}} \longrightarrow M'_i = f(M_i)$$

- Now select a single bit from M' buffer
- One can predict the true story of its variations for $i = 0, 1, \ldots, L-1$

i	Message	bit
0	2A5A058FC295ED	0
1	17BD152B330F0A	1
2	BD9D5EE99FE1F8	0

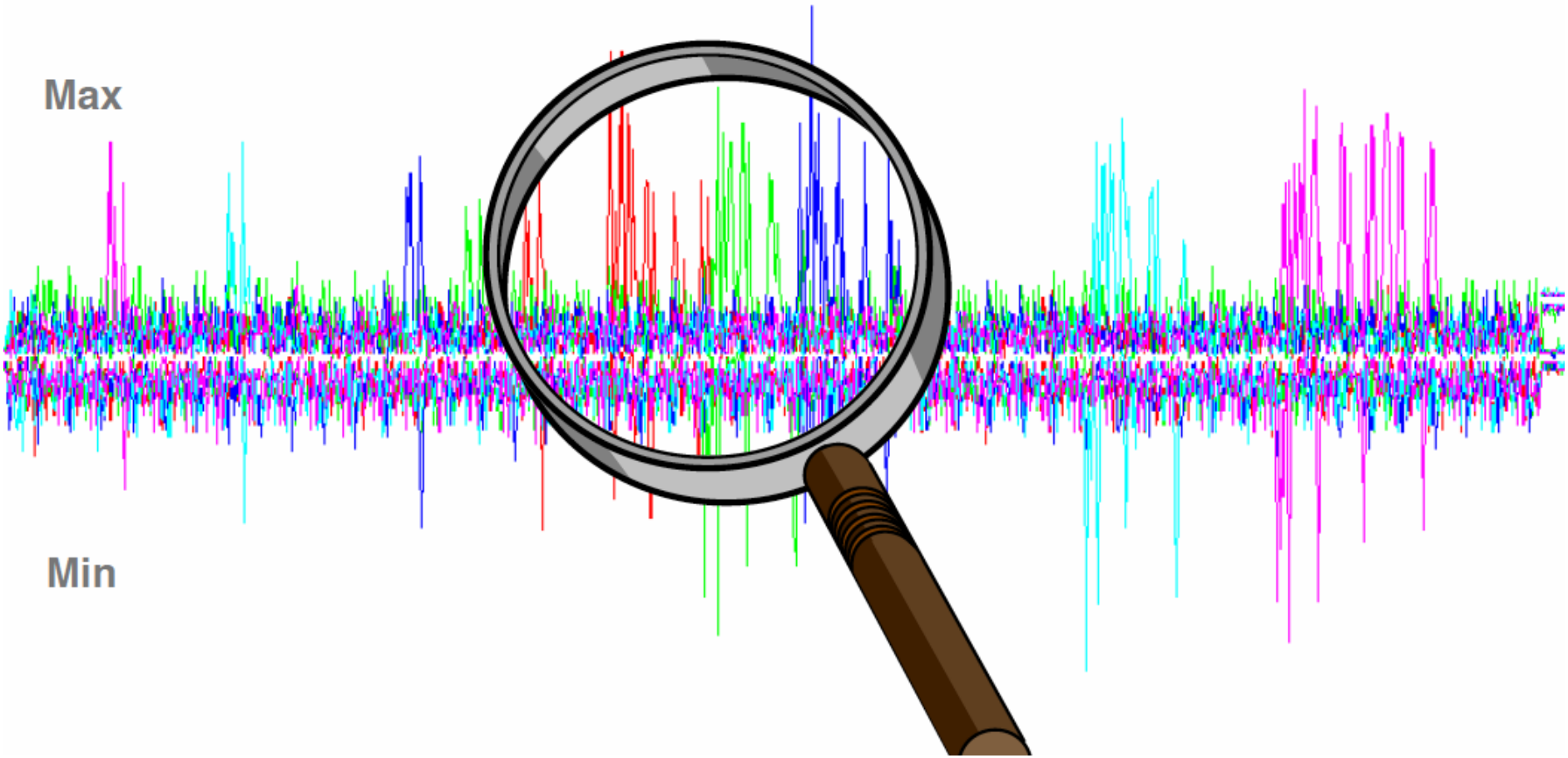
DPA Operator and Curve

- DPA curve construction



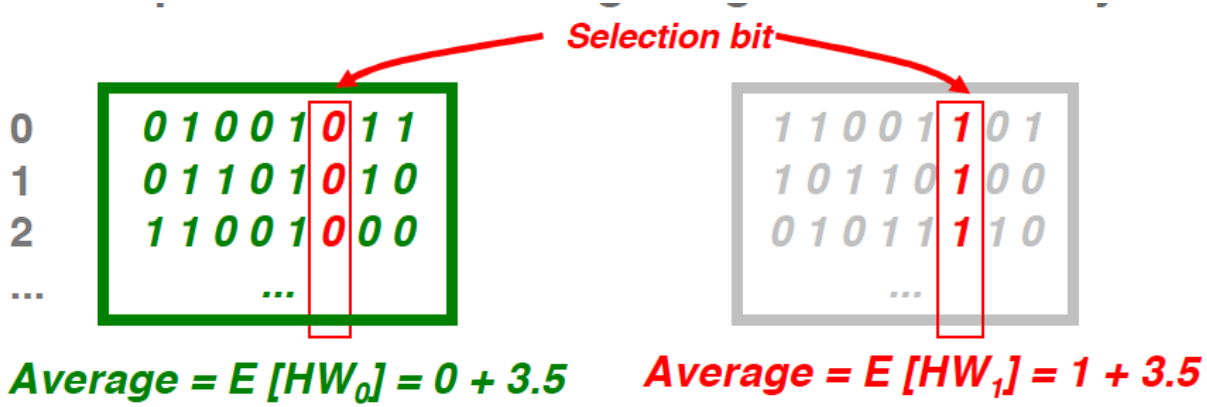
DPA Operator and Curve

- DPA curves for different selection bits



DPA Operator and Curve

- Spikes explanation: Hamming weight of the byte of the selection bits



$$\Delta = E(HW_1) - E(HW_0) = 1$$

- The peak height is proportional to $\sqrt{L}$
- If prediction was wrong, the selection bit would random

$$E(HW_1) = E(HW_0) = 4 \quad \Rightarrow \quad \Delta = 0$$

DPA on RSA

- The entire key (the private exponent d) is not handled together, rather bit by bit in progression
- The prediction can be done by time slices
- Prediction of the next bit requires the previous bit to be broken

RSA Primitive

- Key generation:
  - Input: Key length k
  - Output: $n = p \cdot q$ such that $|n| = k$
  - $\gcd(e, \phi(n)) = 1$ and $d = e^{-1} \pmod{\phi(n)}$
  - Public key: (e, n) and Private key d
- Plain RSA encryption:
  - Input: Message m and public key (e, n)
  - Output: Ciphertext $c = m^e \pmod{n}$
- Plain RSA decryption:
  - Input: Ciphertext c and private key d
  - Output: Message $m = c^d \pmod{n}$

RSA Encryption in Practice

- Plain RSA is insecure
  - Encryption should be probabilistic
  - Plain RSA is multiplicatively homomorphic
- RSA-OAEP
  - Optimal Asymmetric Encryption Padding

$$c = \mu_{\text{OAEP}}(m, r)^e \pmod{n}$$

where r is random

  - OAEP is proposed by Bellare and Rogaway in 1994
  - It is included in PKCS1 and NIST Standards

RSA Signature in Practice

- Plain RSA signature is universally forgeable
- Messages should be appropriately padded and hashed
- RSA signature primitive
  - Setup $n = p \cdot q$ with p and q are prime
  - The public and private exponents satisfy $e \cdot d = 1 \pmod{\phi(n)}$
  - Public parameters (e, n)
  - Private parameters (d, n)
- Signature on message m

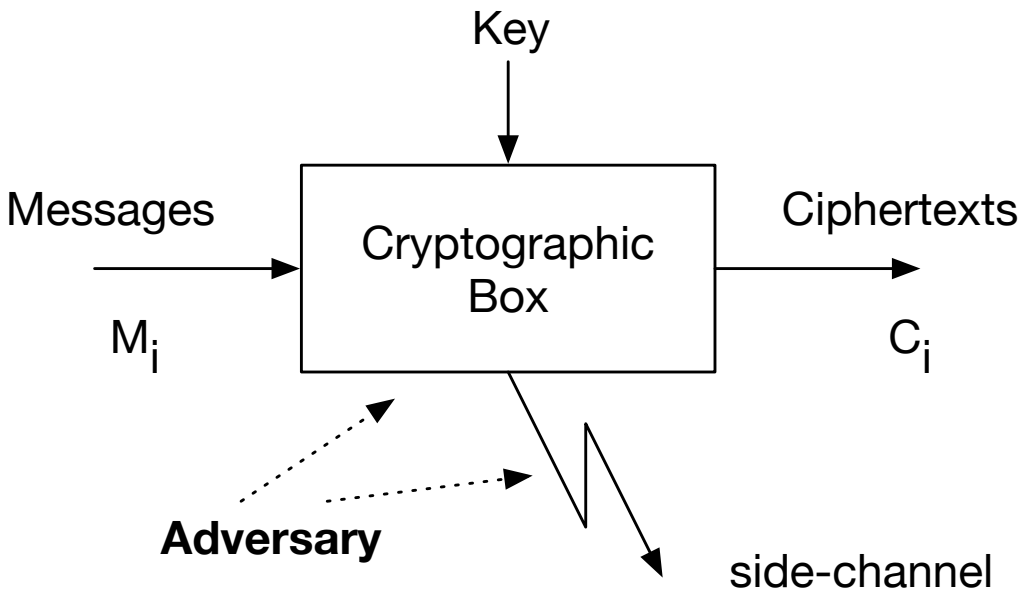
$$s = \mu(m)^d \pmod{n}$$

- Verification of the signature s

$$s^e \stackrel{?}{=} \mu(m) \pmod{n}$$

RSA Countermeasures

- The binary method of exponentiation leaks information on private key



Square-and-Multiply Algorithm

- The binary method is also known as Square-and-multiply algorithm

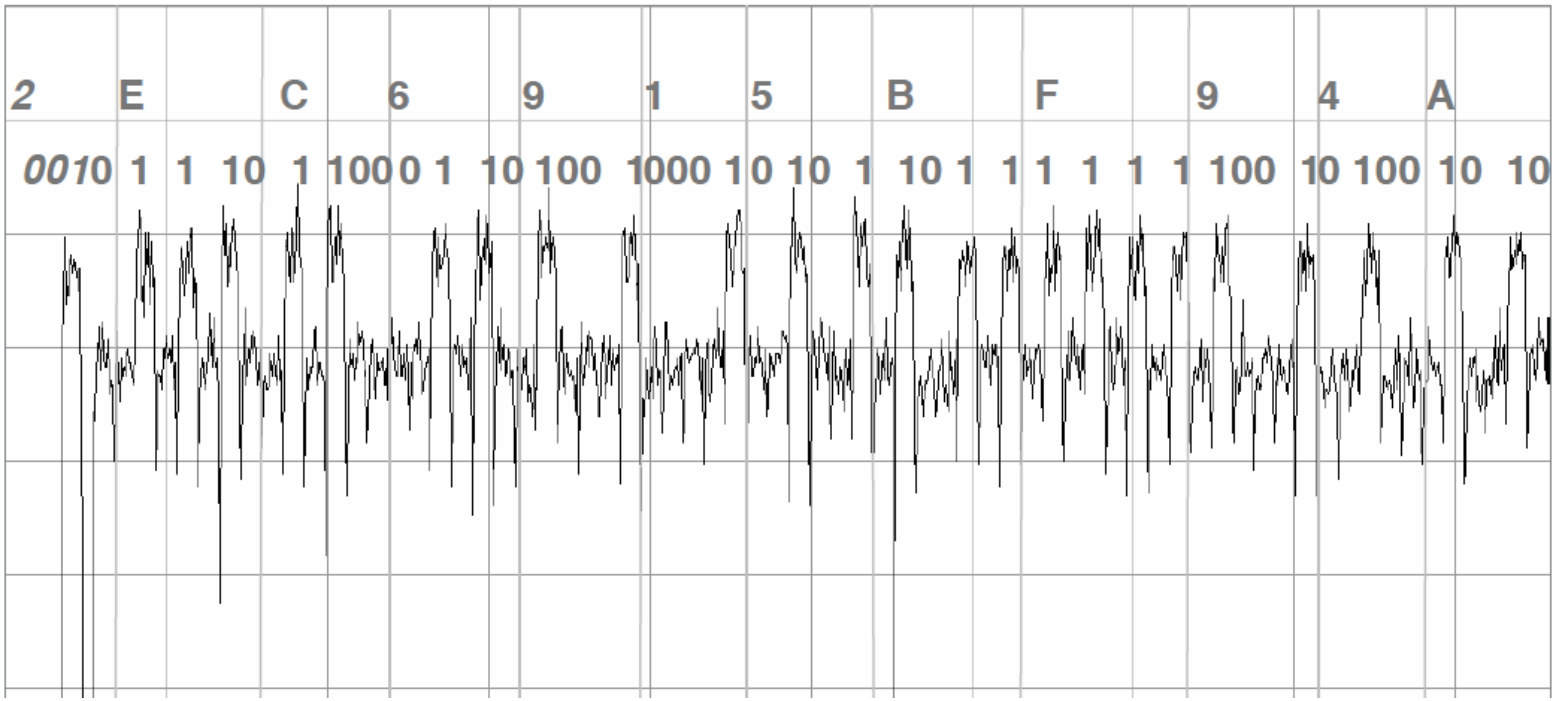
Input: m, $d = (d_{k-1}, \ldots, d_0)_2$, n
Output: $s = m^d \pmod{n}$

1. $R_0 \leftarrow 1$
2. For $i = k - 1$ downto 0
$R_0 \leftarrow R_0^2 \pmod{n}$
If $d_i = 1$ then $R_0 \leftarrow R_0 \cdot m \pmod{n}$
3. Return R_0

- It performs exponentiation left to right
- 2 Temporary variables R_0 and m
- Susceptible to SPA-type attacks

Square-and-Multiply Algorithm

- The key: 2E C6 91 5B F9 4A



Square-and-Multiply-Always Algorithm

- One way to avoid leakage is to square and multiply at every step

Input: $m,\ d = (d_{k-1}, \ldots, d_0)_2,\ n$
Output: $s = m^d \pmod{n}$

1. $R_0 \leftarrow 1\ ;\ R_1 \leftarrow 1$

2. For $i = k - 1$ downto 0

$R_0 \leftarrow R_0^2 \pmod{n}$

$b \leftarrow 1 - d_i\ ;\ R_b \leftarrow R_b \cdot m \pmod{n}$

3. Return R_0

- When $b = 1$ (i.e., $d_i = 0$), there is a dummy multiplication
- The power trace is a regular succession of squares and multiplies
- 3 Temporary variables: R_0, R_1 and m
- Not susceptible to SPA-type attacks
- Susceptible to Safe-Error attacks

Safe-Error Attacks

- Timely induce a fault into ALU during multiply operation at step i
- Check the output
  - If the result is incorrect (invalid signature or error notification), then the error was effective $\Rightarrow d_i = 1$
  - If the result is correct, then the multiplication was dummy (safe error) $\Rightarrow d_i = 0$
- Re-iterate the attack for another value of i

Montgomery Powering Ladder

- Montgomery exponentiation algorithm

Input: $m,\ d = (d_{k-1}, \ldots, d_0)_2,\ n$
Output: $s = m^d \pmod{n}$

1. $R_0 \leftarrow 1\ ;\ \ R_1 \leftarrow m$

2. For $i = k-1$ downto 0

$b \leftarrow 1 - d_i\ ;\ R_b \leftarrow R_0 \cdot R_1 \pmod{n}$

$R_{d_i} \leftarrow R_{d_i}^2 \pmod{n}$

3. Return R_0

- This algorithm behaves regularly without dummy operations
- 2 Temporary variables: R_0 and R_1
- Not susceptible to SPA-type attacks
- Not susceptible to Safe-Error attacks

Square-and-Multiply Algorithm Example

- $e = 9 = (1001)_2$
- Square-and-Multiply Algorithm
- Start with $R_0 = 1$

i	d_i	Step 2a	Step 2b
3	1	$R_0 = R_0^2 = 1$	$R_0 = R_0 m = m$
2	0	$R_0 = R_0^2 = m^2$	
1	0	$R_0 = R_0^2 = m^4$	
0	1	$R_0 = R_0^2 = m^8$	$R_0 = R_0 m = m^9$

- Result: $R_0 = m^9$

- Total of 4 squarings and 2 multiplications

Square-and-Multiply-Always Algorithm Example

- $e = 9 = (1001)_2$
- Square-and-Multiply-Always Algorithm
- Start with $R_0 = 1$ and $R_1 = 1$

i	d_i	b	Step 2a	Step 2b
3	1	0	$R_0 = R_0^2 = 1$	$R_0 = R_0 m = m$
2	0	1	$R_0 = R_0^2 = m^2$	$R_1 = R_1 m = m$
1	0	1	$R_0 = R_0^2 = m^4$	$R_1 = R_1 m = m^2$
0	1	0	$R_0 = R_0^2 = m^8$	$R_0 = R_0 m = m^9$

- Result: $R_0 = m^9$

- Total of 4 squarings and 4 multiplications

Montgomery Powering Ladder Algorithm Example

- $e = 9 = (1001)_2$
- Montgomery Powering Ladder Algorithm
- Start with $R_0 = 1$ and $R_1 = m$

i	d_i	b	Step 2a	Step 2b
3	1	0	$R_0 = R_0 R_1 = m$	$R_1 = R_1^2 = m^2$
2	0	1	$R_1 = R_0 R_1 = m^3$	$R_0 = R_0^2 = m^2$
1	0	1	$R_1 = R_0 R_1 = m^5$	$R_0 = R_0^2 = m^4$
0	1	0	$R_0 = R_0 R_1 = m^9$	$R_1 = R_1^2 = m^{10}$

- Result: $R_0 = m^9$

- Total of 4 squarings and 4 multiplications

Comparing Exponentiation Algorithms

Algorithm	Temporary Variables	Number of Squ & Mul
Square-and-Multiply	2	$k + k/2$
Square-and-Multiply-Always	3	$k + k$
Montgomery Powering Ladder	2	$k + k$

- Are there better algorithms?
- Is it possible to compute $m^e \pmod{n}$ in a secure way, without introducing extra multiplications?
- The **Atomic Square-and-Multiply** algorithms by Marc Joye require $k + k/2$ squarings and multiplications as in the classical (unprotected) algorithm

Atomic Square-and-Multiply Algorithm

- Atomic Square-and-Multiply Algorithm by Marc Joye

Input: $m,\ d = (d_{k-1}, \ldots, d_0)_2,\ n$
Output: $s = m^d \pmod{n}$

1. $R_0 \leftarrow 1\ ;\ R_1 \leftarrow m\ ;\ i \leftarrow k-1\ ;\ b \leftarrow 0$

2. While $i \geq 0$

$R_0 \leftarrow R_0 \cdot R_b \pmod{n}$

$b \leftarrow b \oplus d_i\ ;\ i \leftarrow i - \bar{b}$

3. Return R_0

- This algorithm behaves regularly without dummy operations
- 2 Temporary variables: R_0 and R_1

Atomic Square-and-Multiply Algorithm Example

- $e = 9 = (1001)_2$
- Atomic Square-and-Multiply Algorithm by Marc Joye
- Start with $R_0 = 1$, $R_1 = m$, $i = k - 1 = 3$, and $b = 0$

i	d_i	b	Step 2a	Step 2b
3	1	0	$R_0 = R_0 R_0 = 1$	$b = b \oplus d_i = 1$; $i = i - \bar{b} = 3$
3	1	1	$R_0 = R_0 R_1 = m$	$b = b \oplus d_i = 0$; $i = i - \bar{b} = 2$
2	0	0	$R_0 = R_0 R_0 = m^2$	$b = b \oplus d_i = 0$; $i = i - \bar{b} = 1$
1	0	0	$R_0 = R_0 R_0 = m^4$	$b = b \oplus d_i = 0$; $i = i - \bar{b} = 0$
0	1	0	$R_0 = R_0 R_0 = m^8$	$b = b \oplus d_i = 1$; $i = i - \bar{b} = 0$
0	1	1	$R_0 = R_0 R_1 = m^9$	$b = b \oplus d_i = 0$; $i = i - \bar{b} = -1$

- Result: $R_0 = m^9$

- Total of 4 squarings and 2 multiplications

Right-to-Left Binary Algorithm

- The classical Right-to-Left Binary Algorithm

Input: $m,\ d = (d_{k-1}, \ldots, d_0)_2,\ n$
Output: $s = m^d \pmod{n}$

1. $R_0 \leftarrow 1\ ;\ \ R_1 \leftarrow m\ ;\ i \leftarrow 0$

2. While $i \leq k-1$

If $d_i = 1$ then $R_0 \leftarrow R_0 \cdot R_1 \pmod{n}$

$R_1 \leftarrow R_1^2 \pmod{n}\ ;\ \ i \leftarrow i+1$

3. Return R_0

Right-to-Left Binary Algorithm Example

- $e = 9 = (1001)_2$
- The classical Right-to-Left Binary Algorithm
- Start with $R_0 = 1$, $R_1 = m$, and $i = 0$

i	d_i	Step 2a	Step 2b
0	1	$R_0 = R_0 R_1 = m$	$R_1 = R_1^2 = m^2$; $i = i + 1 = 1$
1	0		$R_1 = R_1^2 = m^4$; $i = i + 1 = 2$
2	0		$R_1 = R_1^2 = m^8$; $i = i + 1 = 3$
3	1	$R_0 = R_0 R_1 = m^9$	$R_1 = R_1^2 = m^{16}$; $i = i + 1 = 4$

- Result: $R_0 = m^9$

- Total of 4 squarings and 2 multiplications

Atomic Right-to-Left Binary Algorithm

- The atomic Right-to-Left Binary Algorithm by Marc Joye

Input: $m,\ d = (d_{k-1}, \ldots, d_0)_2,\ n$
Output: $s = m^d \pmod{n}$

1. $R_0 \leftarrow 1\ ;\ R_1 \leftarrow m\ ;\ i \leftarrow 0\ ;\ b \leftarrow 1$

2. While $i \leq k-1$

$b \leftarrow b \oplus d_i$

$R_b \leftarrow R_b R_1 \pmod{n}\ ;\ i \leftarrow i + b$

3. Return R_0

Atomic Right-to-Left Binary Algorithm Example

- $e = 9 = (1001)_2$
- Atomic Right-to-Left Binary Algorithm by Marc Joye
- Start with $R_0 = 1$, $R_1 = m$, $i = 0$, and $b = 1$

i	d_i	b	Step 2a	Step 2b
0	1	1	$b = b \oplus d_i = 0$	$R_0 = R_0 R_1 = m$; $i = i + b = 0$
0	1	0	$b = b \oplus d_i = 1$	$R_1 = R_1 R_1 = m^2$; $i = i + b = 1$
1	0	1	$b = b \oplus d_i = 1$	$R_1 = R_1 R_1 = m^4$; $i = i + b = 2$
2	0	1	$b = b \oplus d_i = 1$	$R_1 = R_1 R_1 = m^8$; $i = i + b = 3$
3	1	1	$b = b \oplus d_i = 0$	$R_0 = R_0 R_1 = m^9$; $i = i + b = 3$
3	1	0	$b = b \oplus d_i = 1$	$R_1 = R_1 R_1 = m^{16}$; $i = i + b = 4$

- Result: $R_0 = m^9$

- Total of 4 squarings and 2 multiplications

Preventing Side-Channel Attacks

- For SPA-type attacks: Use Montgomery ladder or Atomic algorithms of Marc Joye
- However, these algorithms are not sufficient to thwart DPA-like attacks
- To circumvent the DPA-type attacks, we use data whitening, or randomization, or blinding
- For RSA, randomization of m, d, or n is used in the computation of $s = m^d \pmod{n}$

DPA-Type Countermeasures — Randomizing m

- For a random r compute

$$\begin{aligned} m^* &= r^e \cdot m \pmod{n} \\ s^* &= (m^*)^d \pmod{n} \\ s &= s^* \cdot r^{-1} \pmod{n} \end{aligned}$$

- If e is unknown, compute

$$\begin{aligned} m^* &= r \cdot m \pmod{n} \\ s^* &= (m^*)^d \pmod{n} \\ s &= s^* \cdot r^{-d} \pmod{n} \end{aligned}$$

- For a short random $r < 2^u$, compute

$$\begin{aligned} m^* &= m + r \cdot n \\ n^* &= 2^u \cdot n \\ s^* &= (m^*)^d \pmod{n^*} \\ s &= s^* \cdot r^{-d} \pmod{n} \end{aligned}$$

DPA-Type Countermeasures — Randomizing d

- For a random r compute

$$\begin{aligned} d^* &= d + r \cdot \phi(n) \\ s &= m^{d^*} \pmod{n} \end{aligned}$$

- If $\phi(n)$ is unknown, compute

$$\begin{aligned} d^* &= d + r \cdot (e \cdot d - 1) \\ s &= s^{d^*} \pmod{n} \end{aligned}$$

- If e is unknown, for random $r < d$, compute

$$\begin{aligned} d^* &= d - r \\ s_1^* &= m^{d^*} \pmod{n} \\ s_2^* &= m^{r} \pmod{n} \\ s &= s_1^* \cdot s_2^* \pmod{n} \end{aligned}$$

DPA-Type Countermeasures — Randomizing n

- This technique can be combined with previous ones
- Randomizing n also protects against fault attacks
- For short random numbers r_1 and $r_2 > r_1$, compute

$$\begin{aligned} m^* &= m + r_1 \cdot n \\ n^* &= r_2 \cdot n \\ s^* &= (m^*)^d \pmod{n^*} \\ s &= s^* \pmod{n} \end{aligned}$$

- For short random numbers r_1 and $r_2 > r_1$, compute

$$\begin{aligned} m^* &= m + r_1 \cdot n \\ n^* &= r_2 \cdot n \\ s^* &= (m^*)^d \pmod{n^*} \\ Y &= (m^*)^{d \bmod \phi(r_2)} \pmod{r_2} \\ c &= (S^* - Y + 1) \pmod{r_2} \\ s &= (s^*)^c \pmod{n} \end{aligned}$$

Fault Attacks

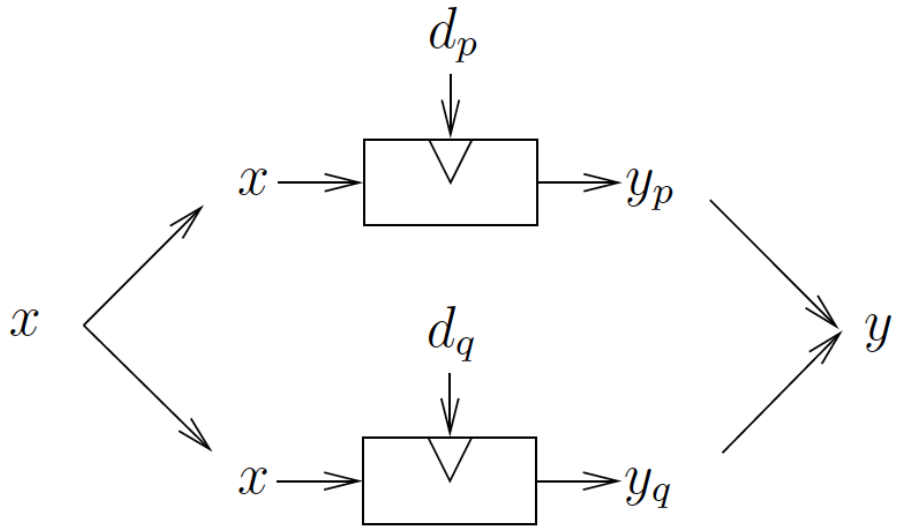
- Safe-error attack was a type of fault attack
- Timely induce a fault into ALU during multiply operation at step i
- Check the output
  - If the result is incorrect (invalid signature or error notification), then the error was effective $\Rightarrow d_i = 1$
  - If the result is correct, then the multiplication was dummy (safe error) $\Rightarrow d_i = 0$
- Re-iterate the attack for another value of i
- It was introduced into the RSA when Square-and-Multiply-Always algorithm was used

Fault Attack Assumptions

- Precise bit errors
  - The attacker can cause a fault in a single bit
  - Full control over the timing and location of the fault
- Precise byte errors
  - The attacker can cause a fault in a single byte
  - Full control over the timing but only partial control over the location (e.g., which byte is affected)
- Unknown byte errors
  - The attacker can cause a fault in a single byte
  - Partial control over the timing and location of the fault
- Random errors
  - Partial control over the timing and no control over the location

GCD Attack

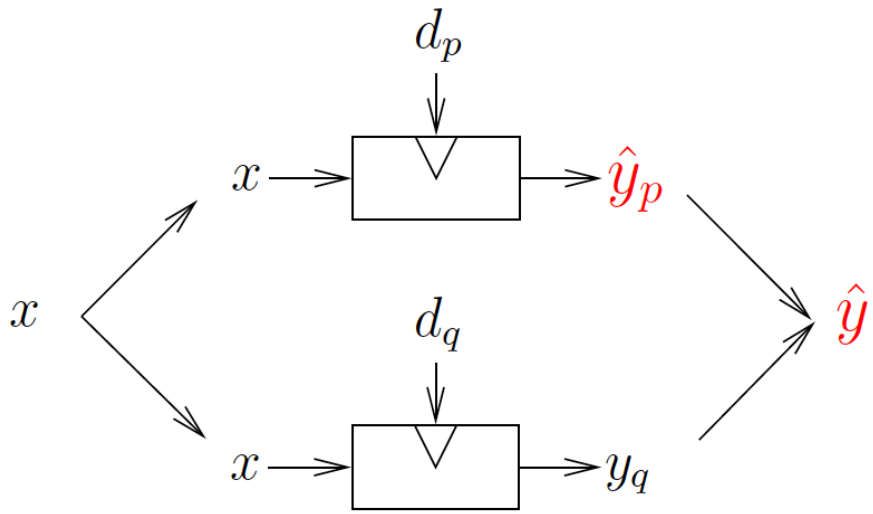
- GCD attack is possible if the CRT version of RSA signature computation is used
- Computation of a signature $y = x^d \pmod{n}$ using CRT
- $y_p = x^{d_p} \pmod{p}$ with $d_p = d \pmod{p-1}$
- $y_q = x^{d_q} \pmod{q}$ with $d_q = d \pmod{q-1}$



- $y = \mathrm{CRT}(y_p, y_q) = y_p + p \cdot [i_p \cdot (y_q - y_p) \bmod q]$

GCD Attack

- Assume that due to the fault in ALU, y_p is incorrectly computed



- The prime factor q can be obtained from the incorrect $\hat{y}_p$ using

$$\gcd(\hat{y}^e - x \bmod n,\ n) = q$$

- Because

$$\hat{y}^e - x = \hat{y}_p^e - x \neq 0 \pmod{n} \iff p \nmid (\hat{y}_p^e - x)$$
$$\hat{y}^e - x = \hat{y}_q^e - x = 0 \pmod{n} \iff q \mid (\hat{y}_p^e - x)$$

GCD Attack Demonstration

- Let $p = 17$ and $q = 19$, which gives $n = p \cdot q = 323$ and $\phi(n) = (p-1) \cdot (q-1) = 288$
- Select $e = 23$, since $\gcd(e, \phi(n)) = \gcd(23, 288) = 1$
- Compute $d = e^{-1} \pmod{n}$, which gives $d = 263$
- Select $x = 100$
- We compute $y = x^d \pmod{n}$, which gives $y = 25$

GCD Attack Demonstration

- In order to apply CRT, we first compute

$$\begin{array}{lcll} d_p = d \bmod (p-1) & \rightarrow & d_p = 263 \bmod 16 & \rightarrow \quad d_p = 7 \\ d_q = d \bmod (q-1) & \rightarrow & d_q = 263 \bmod 18 & \rightarrow \quad d_q = 11 \\ i_p = p^{-1} \bmod q & \rightarrow & i_p = 17^{-1} \bmod 19 & \rightarrow \quad i_p = 9 \end{array}$$

- Mod p exponentation:

$$y_p = x^{d_p} \pmod{p} \quad \rightarrow \quad y_p = 100^7 \bmod 17 \quad \rightarrow \quad y_p = 8$$

- Mod q exponentation:

$$y_q = x^{d_q} \pmod{q} \quad \rightarrow \quad y_q = 100^1 1 \bmod 19 \quad \rightarrow \quad y_q = 6$$

- The CRT produces

$$\begin{aligned} y &= y_p + p \cdot [i_p \cdot (y_q - y_p) \bmod q] \\ &= 8 + 17 \cdot [9 \cdot (6-8) \bmod 19] \\ &= 25 \end{aligned}$$

GCD Attack Demonstration

- Now assume that y_p was incorrectly computed
- Instead of $y_p = 8$, we compute $\hat{y}_p = 10\text{d}$ due to a fault in ALU
- This incorrect value $\hat{y}_p = 10$ would be used in the CRT computation

$$\begin{aligned} \hat{y} &= \hat{y}_p + p \cdot [i_p \cdot (y_q - \hat{y}_p) \bmod q] \\ &= 10 + 17 \cdot [9 \cdot (6 - 10) \bmod 19] \\ &= 44 \end{aligned}$$

- This resulting incorrect value $\hat{y} = 44$ allows us to obtain q using

$$\begin{aligned} q &= \gcd(\hat{y}^e - x \bmod n, n) \\ &= \gcd(44^{23} - 100 \bmod 323, 323) \\ &= \gcd(5 - 100 \bmod 323, 323) \\ &= \gcd(-95 \bmod 323, 323) \\ &= \gcd(238, 323) \\ &= 17 \end{aligned}$$

Countermeasures Against GCD Attack

- Recomputation
  - It does not detect permanent errors
  - It doubles the computation time
- Verification
  - It may double the computation time
  - It requires the knowledge of e

Countermeasures Against GCD Attack

- Shamir's method
- Choose a small random r
- Compute $y_{rp} = x^{d \bmod \phi(rp)} \bmod rp$
- Compute $y_{rq} = x^{d \bmod \phi(rq)} \bmod rq$
- If $y_{rp} \neq y_{rq} \pmod{r}$, output ERROR and stop
- Output $y = \mathrm{CRT}(y_{rp} \bmod p,\ y_{rq} \bmod q)$

- Shamir's method requires the knowledge of d
- However, in CRT, only d_p and d_q are available

RSA Error Detection – The Standard Mode

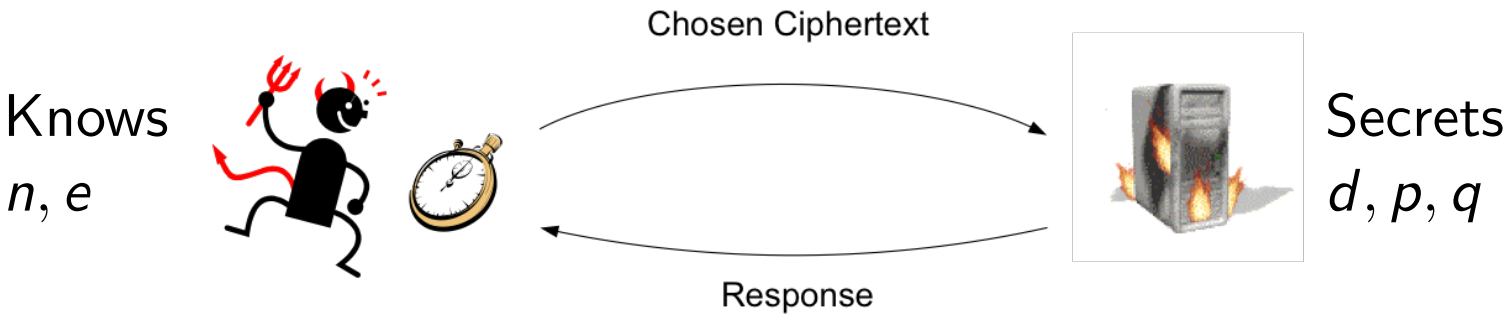
- Compute $z = x^d \pmod{rn}$
- Compute $y_r = x^{d \bmod \phi(r)} \pmod{r}$
  (Note that r can be chosen prime, this $\phi(r) = r - 1$)
- If $y_r \neq z \pmod{r}$, output ERROR and stop
- Output $y = z \pmod{n}$

RSA Error Detection – The CRT Mode

- Compute $z_1 = x^{d_p} \pmod{r_1 p}$
- Compute $y_{r_1} = x^{d_p \bmod \phi(r_1)} \pmod{r_1}$
- If $y_{r_1} \neq z_1 \pmod{r_1}$, output ERROR and stop
- Compute $z_2 = x^{d_q} \pmod{r_2 q}$
- Compute $y_{r_2} = x^{d_q \bmod \phi(r_2)} \pmod{r_2}$
- If $y_{r_2} \neq z_2 \pmod{r_2}$, output ERROR and stop
- Output $y = \mathrm{CRT}(z_1 \bmod p, z_2 \bmod q)$

Micro-Architectural Side-Channel Attacks

- Historical targets of side-channel attacks were smart cards
- The vulnerability of computer systems (e.g., remote servers) was not known until Brumley and Boneh Attack (2003)
- This remote timing attack on RSA (OpenSSL implementation on a web server) shows the practicality of such attacks
- The attack revealed 1024-bit RSA key of a server over a LAN
- Remote RSA attack was improved, now requiring $< 100,000$ queries (while the original attack needed 1.4 million queries)



Micro-Architectural Side-Channel Attacks

- Side-channel attacks can be applied to the PC as well
- This is rather interesting since maturing Trusted Computing efforts promise a "trusted environment" with isolated execution for applications, etc.
- These new side-channel attacks are different from embedded platforms
- The PC platform environment is quite different from the embedded security platforms.
- Only pure "unprivileged" software-based attacks are really interesting

Micro-Architectural Side-Channel Attacks

- Since the power is not easily observable in a complicated device such as PC, the timing variations are targeted
- Timing variances exploited in timing attacks are caused by:
  - Different data-dependent execution paths
  - Different (number of) instructions executed in those paths
- In general, micro-architectural attacks exploit timing and access variations caused by the components of the CPU (even if the same sequence instructions are always executed)

Micro-Architectural Side-Channel Attacks

- Micro-architectural side-channel attacks are a new class of attacks that exploit the micro-architecture and throughput-oriented internal functionality of modern processor components
- Micro-architectural attacks exploit the execution time variations caused by CPU components
- Currently there are 4 types of Micro-Architectural Attacks:
  - Cache Analysis
  - Branch Prediction Analysis
  - Instruction Cache
  - Shared Functional Units

Micro-Architectural Side-Channel Attacks

- These attacks capitalize on the situations where several applications share the same processor resources
- The shared usage between spy and crypto process allows a spy process running in parallel to the victim process to extract critical information like secret keys.
- On powerful PC-platforms many applications can run in parallel:
  - Either quasi-parallel enabled by OS scheduling, or
  - More or less explicitly parallel depending on the degree of additional hardware, such dual processors, multicores, and simultaneous multi threading

Micro-Architectural Side-Channel Attacks

- Thus, several applications share the same processor and its resources, and also at more or less the same time
- Therefore, when a highly critical crypto algorithm is executed, there is the potential threat that a malicious or so called spy process is executed in parallel with the crypto process which might try to extract critical or secret information by "spying" on the crypto process during its execution

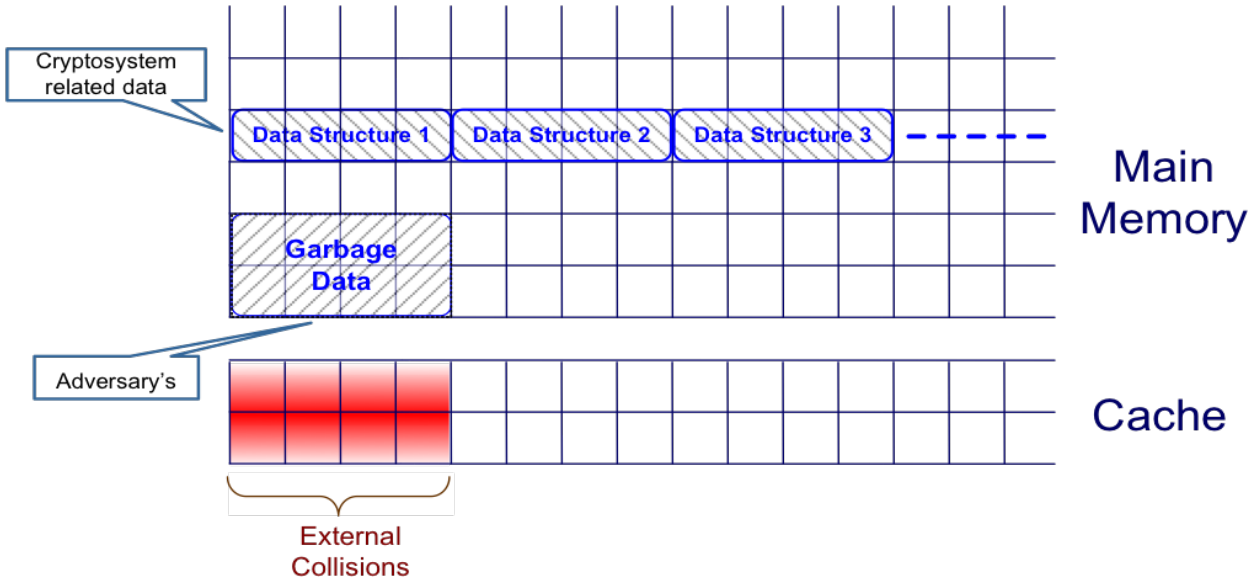
Cache Attacks

- Cache is a small and fast storage area used by the CPU to reduce the average time to access main memory.
- It stores copies of the most frequently used data
- When a CPU needs to read a location in main memory, it first checks to see if the data is already in the cache
  - Cache Hit: data is already in the cache; CPU immediately uses this data in cache.
  - Cache Miss: data is not in the cache; CPU reads it from the memory and stores a copy in the cache.
- Cache Block: The minimum amount of data that can be read from the main memory into the cache at once
- Each cache miss causes a cache block to be retrieved from a higher level memory.

Cache Attacks

- Cache attacks exploit the cache behavior (i.e., cache hit/miss statistics) of cryptosystems
- Cache architecture leaks information about memory access patterns
- The sources of information leakage:
  - Execution Time: cache misses take more time than a cache hit
  - Power Consumption: cache misses require more power than a cache hit
- Cryptosystems have data dependent memory access patterns
- Once the access patterns are extracted, an adversary may recover the secret key

Cache Attacks

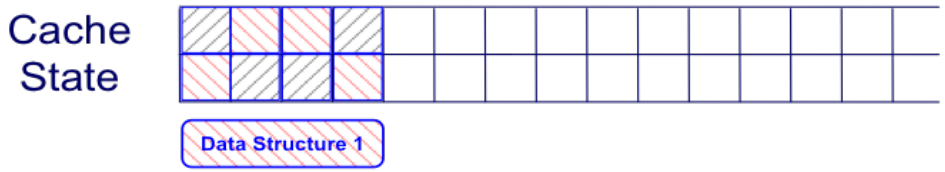


- An access to "Data Structure 1" may evict adversary's data and vice versa
- An adversary can infer if/when "Data Structure 1" is accessed during the encryption

Cache Attacks

- Adversary reads the garbage data via the "Spy Process"

- Case 1: "Data Structure 1" is accessed



- Case 2: "Data Structure 1" is not accessed

- Spy process reads the garbage data again
  - Case 1: takes more time to read it
  - Case 2: takes less time to read it

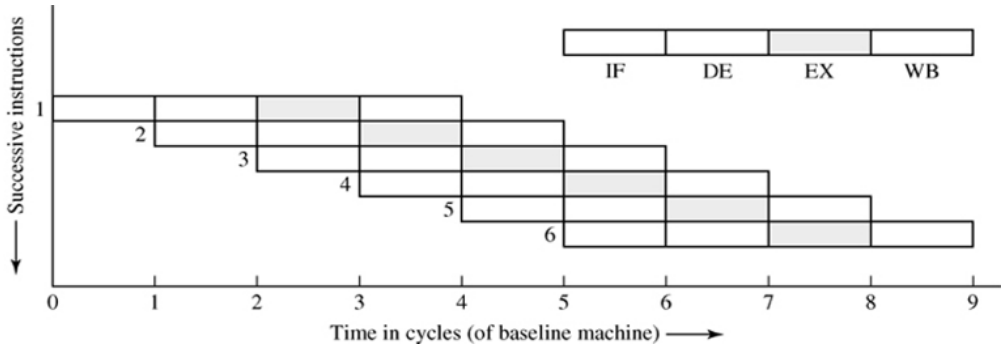
Branch Prediction Attack (BPA)

- A very new software side-channel enabled by the branch prediction capability common to all modern CPUs
- The penalty paid (extra clock cycles) for a mispredicted branch can be used for cryptanalysis of cryptographic primitives that employ a data-dependent program flow
- BPA allows an unprivileged process to attack other processes running in parallel on the same processor
- Works despite of sophisticated partitioning methods such as memory protection, sandboxing or even virtualization
- Public-key ciphers like RSA and ECC are susceptible to BP attacks

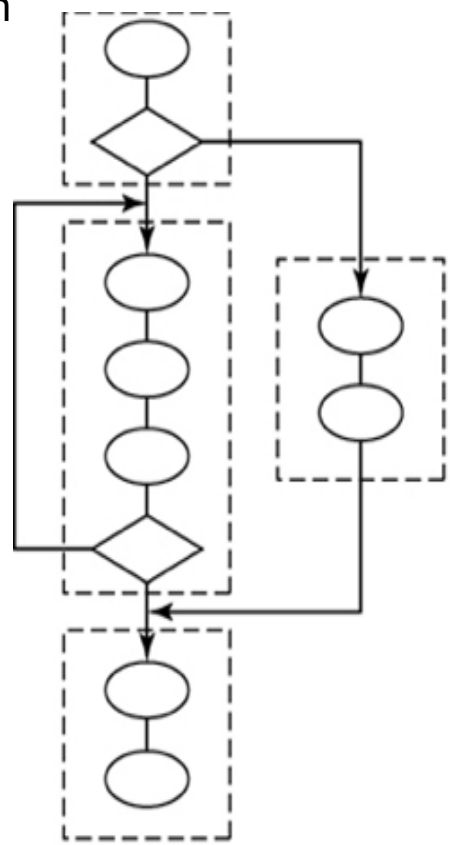
Branch Prediction Attack

- Superscalar processors have to execute instructions speculatively to overcome control hazards
- Branch prediction units try to predict the most likely execution path after a branch
- A branch instruction is a point in the instruction stream of a program where the next instruction is not necessarily the next sequential one
- For conditional branches, the decision to take the branch or not to take depends on some condition that must be evaluated in order to make the correct decision
- During this evaluation period, the processor speculatively executes instructions from one of the possible execution paths instead of stalling and awaiting for the decision to come through

Branch Prediction Attacks



Due to the deep pipelining of the instruction sequences



A branch operation could stall the pipeline

Introducing misprediction delays up to 150 cycles

Basic Pentium III Processor Misprediction Pipeline

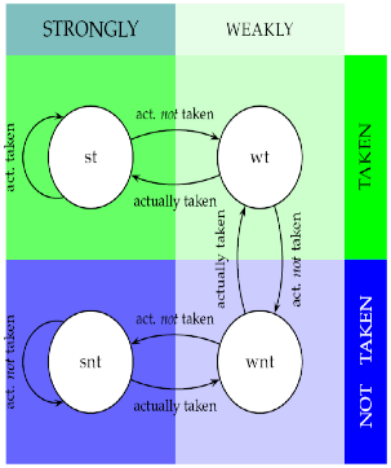
1	2	3	4	5	6	7	8	9	10
Fetch	Fetch	Decode	Decode	Decode	Rename	ROB Rd	Rdy/Sch	Dispatch	Exec

Basic Pentium 4 Processor Misprediction Pipeline

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
TC Nxt IP		TC Fetch		Drive	Alloc	Rename		Que	Sch	Sch	Sch	Disp	Disp	RF	RF	Ex	Flgs	Br Ck	Drive

Branch Prediction Attack

- A branch predictor determines whether a conditional branch in the instruction flow of a program is likely to be taken or not



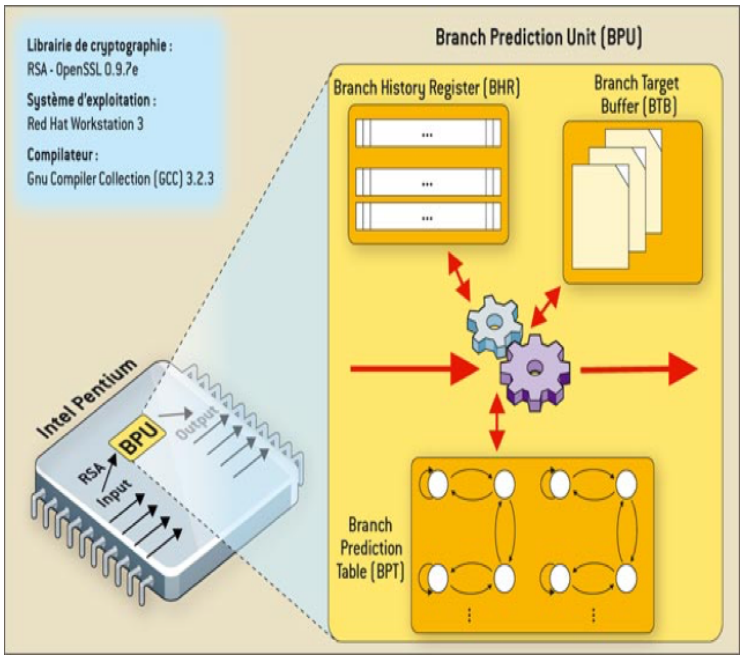
- Branch predictors are crucial in today's modern, superscalar processors for achieving high performance
- They allow processors to fetch and execute instructions without waiting for a branch to be resolved

- Almost all pipelined processors do branch prediction of some form, because they must guess the address of the next instruction to fetch before the current instruction has been executed
- Branch prediction is not the same as branch target prediction
- Branch target prediction attempts to guess the target of the branch or unconditional jump before it is computed by parsing the instruction itself

Branch Prediction Attack

- BPU consists of mainly two "logical" parts: the branch target buffer (BTB) and the branch predictor logic

- BTB is the buffer where the CPU stores the target addresses of the previously executed branches
- BTB is limited in size, the CPU can store only a number of such target addresses, and previously stored addresses are evicted from the BTB if a new address needs to be stored instead
- The predictor is that part of the BPU that makes the prediction on the outcome of the branch



Branch Prediction Attack

- BPA uses the branch misprediction delays to break cryptographic primitives using a data-dependent program flow
- BPA also allows an unprivileged process to attack other processes running in parallel on the same processor even in the presence of sophisticated partitioning methods such as memory protection, sandboxing or even virtualization
- 4 different branch prediction attacks are proposed:
  - Exploiting the Predictor directly (Direct Timing Attack)
  - Forcing the BPU to the Same Prediction (Asynchronous Attack)
  - Forcing the BPU to the Same Prediction (Synchronous Attack)
  - Trace-driven Attack against the BTB (Simple Prediction Attack)

Direct Timing Attack (DTA)

- DTA relies on the fact that the prediction algorithms are deterministic (i.e., predictable)
- DTA assumes that an adversary attacks an RSA cipher with a private exponent d and knows the first (the most significant) i bits of d and is trying to reveal d_i.

$$
\begin{aligned}
&S = A * B \\
&S = (S - (S * N^{-1} \bmod R) * N)/R \\
&\textbf{if } S > N \textbf{ then } S = S - N \\
&\textbf{return } S
\end{aligned}
$$

Direct Timing Attack

- For any message m, the adversary can simulate the first i steps of the operation and obtain the intermediate result that will be the input of the $(i+1)$th squaring
- Then, the attacker creates 4 different message sets M_1, M_2, M_3, and M_4, such that

$$
\begin{aligned}
M_1 &= \{m \mid m \text{ causes a misprediction during MM of } (i+1)^{\text{th}} \text{ squaring if } d_i = 1\} \\
M_2 &= \{m \mid m \text{ does not cause a misprediction during MM of } (i+1)^{\text{th}} \text{ squaring if } d_i = 1\} \\
M_3 &= \{m \mid m \text{ causes a misprediction during MM of } (i+1)^{\text{th}} \text{ squaring if } d_i = 0\} \\
M_4 &= \{m \mid m \text{ does not cause a misprediction during MM of } (i+1)^{\text{th}} \text{ squaring if } d_i = 0\}
\end{aligned}
$$

- If the difference between the average execution time of M_1 and M_2 is more significant than that of M_3 and M_4, then the attacker guesses that $d_i = 1$. Otherwise d_i should be 0

Implications of Micro-Architectural Attacks (MAA)

- The micro-architectural attacks work despite of sophisticated partitioning and protection methods such as memory protection, sandboxing, and virtualization
- They can impact: Multiuser systems, VPNs, Virtual machines, Trusted computing, Sandboxes (JVM, JavaScript), and Remote attacks
- They are:
  - Easy to deploy – pure software attacks
  - Hard to detect
  - Hard to protect efficiently

Software Countermeasures against MAA

- OpenSSL had gone into several revisions and implemented many software countermeasures
- Yet, new micro-architectural vulnerabilities of OpenSSL are being discovered
- Are software countermeasures sufficient?
  - Solving the problem in software means solving it one by one
  - Software solutions are algorithm- and attack-specific
  - They incur high performance overhead

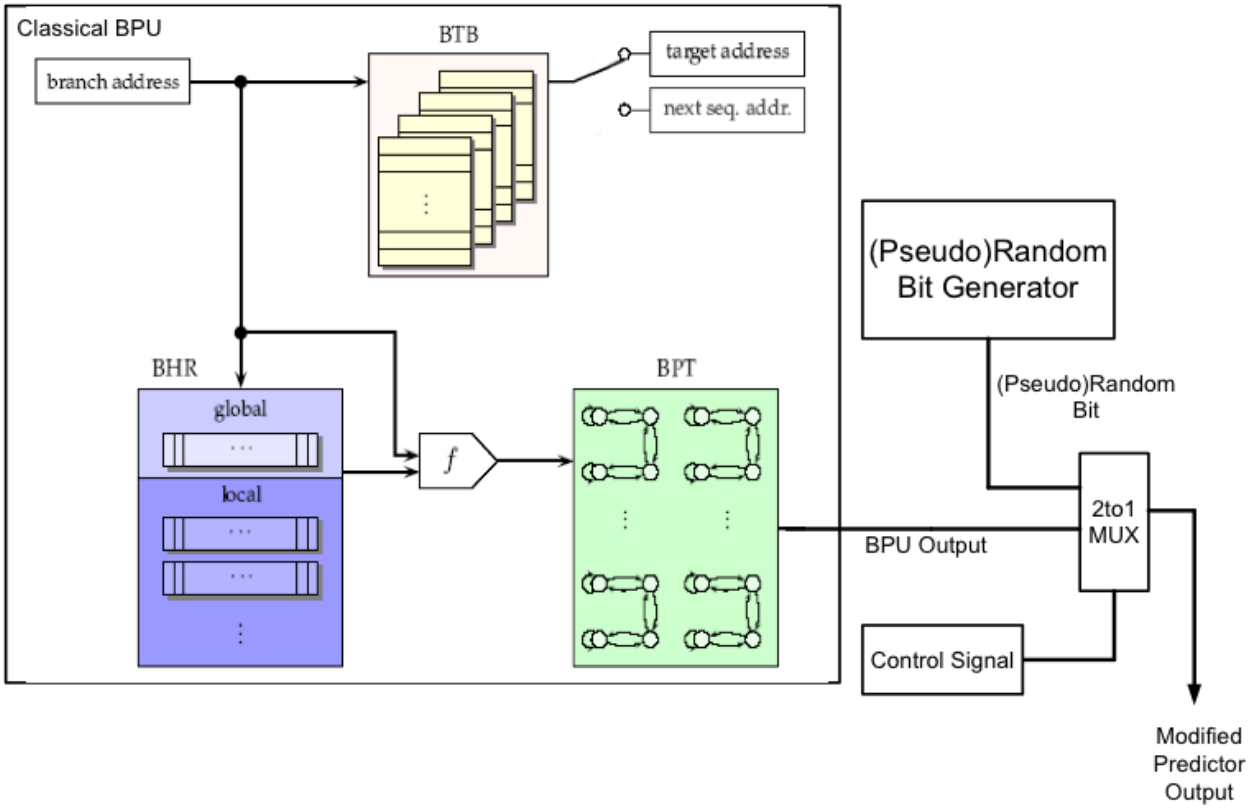
Hardware Countermeasures against MAA

- Hardware countermeasures:
  - Solving it in hardware may mean solving it for all
  - They may not be algorithm-specific
  - They may have much less overhead
- Possible branch prediction countermeasures:
  - Randomizable branch prediction
  - Partitioned Branch Target Buffers
  - Locking mechanism for BTB
  - Protected BTB area
  - Flushing mechanism for BTB
  - Dynamically disabling branch predictions

Hardware Countermeasures against MAA

Randomizable Prediction:

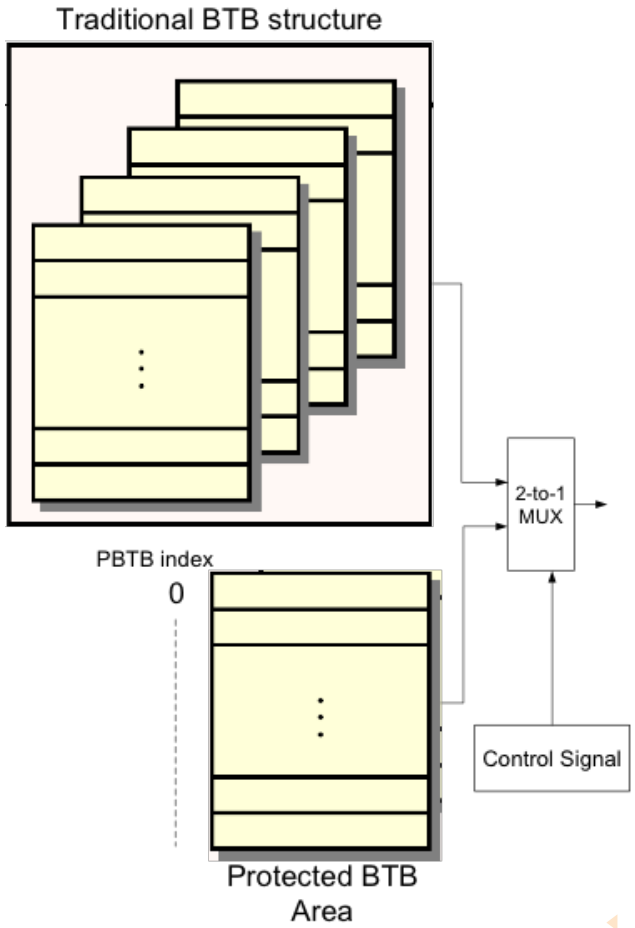
- Modify the prediction output
- BPU can output random predictions for critical branches



Hardware Countermeasures against MAA

Protected Branch Target Buffer:

- Allow critical code to benefit from using a protected buffer
- The entries in PBTB can be handled in a more secure way



Final Recommendations Against Side-Channel Attacks

- Always consider side-channel attacks when implementing cryptographic functions
- Check that the countermeasures do not introduce new vulnerabilities
- Avoid decisional tests
- Randomize execution
- Combine hardware and software protections
- Always prefer cryptographic standards