# Overview

A. Execution model

*language features, sandbox,*
*applet firewall: object isolation and sharing*

B. On-card Cryptography

*algorithms and protocols, good cryptographic practice*

C. Protecting against attacks

*timing attacks, SPA/DPA, fault injection*





# C. Attacks: Timing

- "Optimized" Comparison Operations
  - *attackers learn from early exits (e.g., testing a PIN digit-wise)*
    ☛ *code should have constant run time*

OOPS

```
boolean arrayCompare(byte[] ba1, byte[] ba2) {
   short l;
   if ((l = ba1.length) != ba2.length)
     return false;

   while (--l >= 0)
     if (ba1[l] != ba2[l])
       return false;

   return true;
}
```

# C. Attacks: Timing

- "Optimized" Comparison Operations
  - *attackers learn from early exits (e.g., testing a PIN digit-wise)*
    ☛ *code should have constant run time*

OOPS

```
boolean arrayCompare(byte[] ba1, byte[] ba2) {
   short l;
   if ((l = ba1.length) !=
     return false;

   while (--l >= 0)
     if (ba1[l] != ba2[l])
       return false;

   return true;
}
```

GOOD

```
boolean arrayCompare(byte[] ba1, byte[] ba2) {
   short l1, l2;
   boolean result = true;

   if ((l1 = ba1.length) != (l2 = ba2.length))
     result = false;

   l1 = MIN(l1,l2);
   while (--l1 >= 0)
     if (ba1[l1] != ba2[l1])
       result = false;

   return result;
}
```





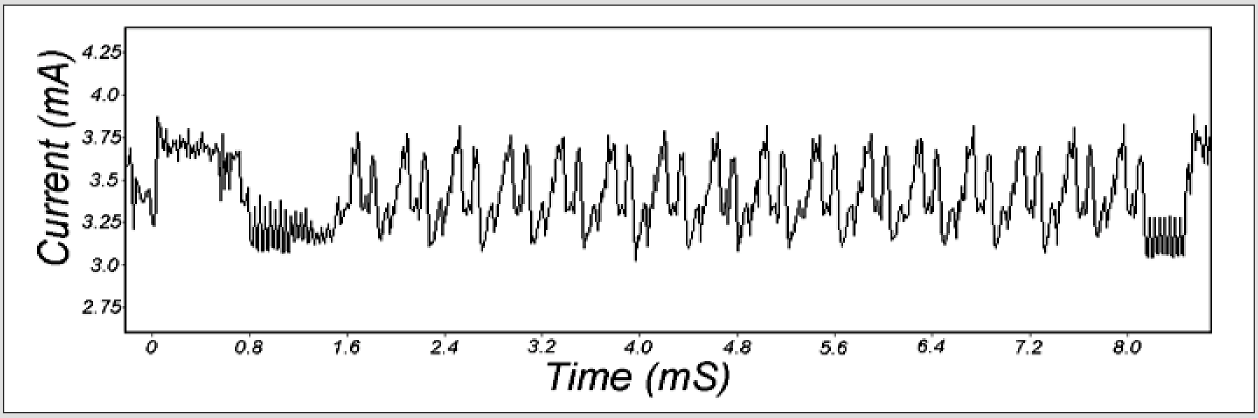
# C. Attacks: Power Analysis

- semiconductor logic gates are constructed out of transistors
  ☛ electrons flow across the silicon substrate when charge is applied to (or removed from) a transistor's gate, consuming power and producing electromagnetic radiation

- Simple Power Analysis (SPA)
  - *measures a circuits power consumption by inserting a resistor in series with power or ground*
    *(the voltage difference across a resistor divided by the resistance yields the current)*
  - *large-scale power variations due to the instruction sequence*
  - *digitally samples at rates over 1 Ghz with less than 1% error are possible*
    *(devices capable of sampling at 20 Mhz cost less than $400)*

## C. Attacks: Power Analysis

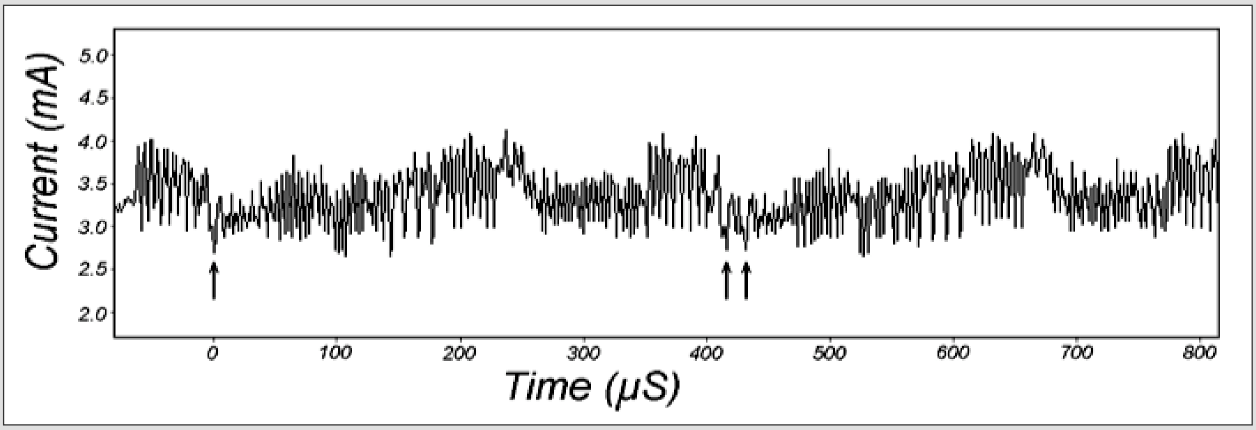
- SPA: DES
  - *full trace, 16 rounds*



## C. Attacks: Power Analysis

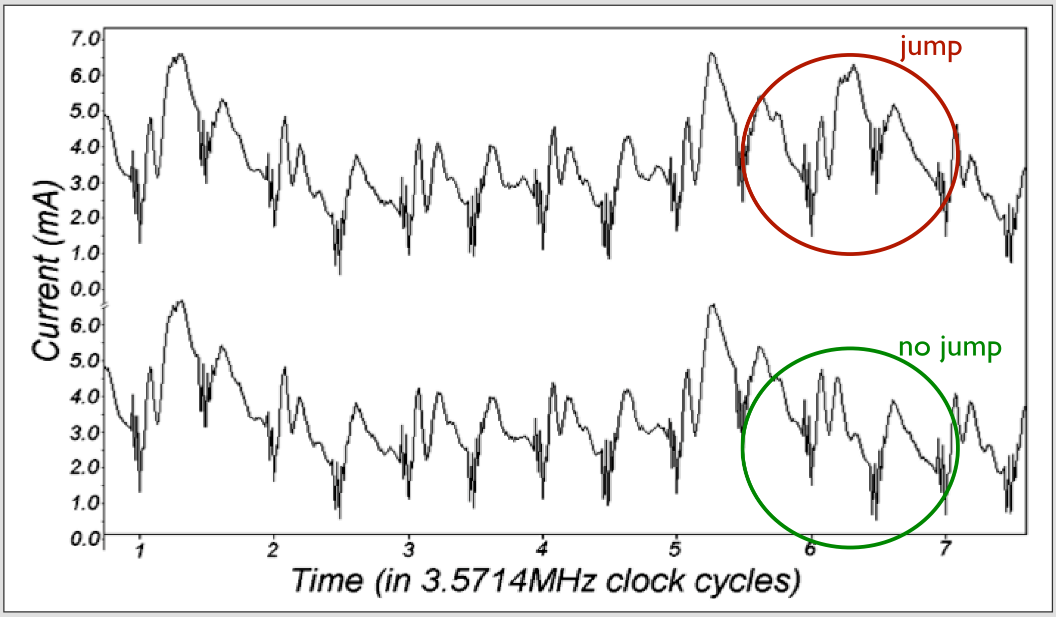
- SPA: DES (cont'd)
  - *shows 2nd and 3rd round*
  - *28-bits key registers C and D are rotated once in round 2 and twice in round 3*



# C. Attacks: Power Analysis

- SPA: DES (cont'd)








# C. Attacks: Power Analysis

- SPA: Areas of Attack
  - ***key schedules:*** *DES involves rotating 28-bits key registers with conditional branches to check the bit shifted of the end*
  - ***permutations:*** *conditional branches can cause power consumption differences*
  - ***comparisons:*** *memory comparisons typically perform a conditional branch when a mismatch is found*
  - ***multipliers:*** *modular multiplication tends to leak information about the data*
  - ***exponentiators:*** *simple modular exponentiation scans across the exponent and performs a squaring operation in every iteration with an additional multiplication operation for each 1 exponent bit*

# C. Attacks: Power Analysis

- SPA: Countermeasures
  - *avoid using keys for conditional branching operations (may require "creative" coding and cause serious performance penalties)*
  - *constant execution paths (possibly introducting dummy operations)*
  - *hardware implementations have sufficiently small power consumption variations that SPA does not yield key material*





# C. Attacks: Power Analysis

- Differential Power Analysis (DPA)
  - *measures effects correlated to data values being manipulated (much smaller, often overshadowed by measurement errors)*
    ☛ *statistical functions tailored to the target algorithm*
  - *signals leaking during public-key operations tend to be stronger than for symmetric key operations*
  - *can be used to break implementations of virtually all algorithms*
- DPA: Countermeasures
  - *reduce signal sizes (e.g., constant execution path code, use operations that leak less, physically shielding the device)*
  - *introduce noise and temporal obfuscation (randomize execution time/ordering)*
- Related Attacks
  - *examine the electromagnetic radiation*

# C. Attacks: PIN Counter

- **Example:** Circumvent a PIN try count

OOPS

```
class PIN {
   byte[] _pin;
   byte tryCount = 3;

   boolean verify(byte[] pin) {
      boolean result = true;

      for (short i = 0; i < _pin.length; ++i)
        result = result && (_pin[i] == pin[i]);

      if (!result)
        --tryCount;

      return result;
   }
};
```





# C. Attacks: PIN Counter

- **Example:** Circumvent a PIN try count

OOPS

```
class PIN {
   byte[] _pin;
   byte tryCount = 3;

   boolean verify(byte[] p
      boolean result = tru

      for (short i = 0; i
        result = result &&

      if (!result)
        --tryCount;

      return result;
   }
};
```

GOOD

```
class PIN {
   byte[] _pin;
   byte tryCount = 3;

   boolean verify(byte[] pin) {
      boolean result = true;

      --tryCount;
      for (short i = 0; i < _pin.length; ++i)
        result = result && (_pin[i] == pin[i]);

      if (result)
        ++tryCount;

      return result;
   }
};
```

# C. Attacks: Fault Injection

- light attacks may erase/modify individual memory cells
- two types of attacks:
  - *code/PC manipulation*
  - *modification of data (e.g., return values, key material)*

- Code/PC manipulation
  - *"erased" instructions usually become* **`nop`** *instructions*
  - *may eliminate conditional jumps or erase security checks*
  - *countermeasures: default to error handling code, jump to "good" cases*
    *code traces (very complex)*

# C. Attacks: Fault Injection

- **Example:** Avoid PIN check result

OOPS

```
if (pinOK) {
   // allow access


} else {
   // error handling
}
```

```
sload ...
ifeq ...
...


...
```

# C. Attacks: Fault Injection

- **Example:** Avoid PIN check result

OOPS

```
if (pinOK) {
   // allow access


} else {
   // error handling
}
```

GOOD

```
if (pinOK == false) {
   // error handling


} else {
   // allow access
}
```

*generating a jump to the "good" case is much more difficult*





# C. Attacks: Fault Injection

- Modification of Data
  - *manipulation of return values (e.g., after a PIN check)*
    - ☛ avoid return values which may be "easily" generated such as `0x00` or `0xFF`
    - ☛ store important values redundantly
  - *attacking key material*
    - calculations with carefully modified key material may leak key data
    - ☛ avoid crypto operations with potentially modified key material (e.g., using CRC)
    - ☛ store key material encrypted to make the effects of modification unpredictable

# C. Attacks: `javacardx.crypto`

- `javacardx.crypto.KeyEncryption`
  – *methods to enable encrypted key data access to a key implementation*
- `javacard.security.KeyBuilder`
  – *the key object factory*





# C. Attacks: `javacardx.crypto`

- `javacardx.crypto.KeyEncryption`
  – *methods to enable encrypted key data access to a key implementation*
- `javacard.security.Key`
  – *the key object factory*

```
public interface KeyEncryption {
   public Cipher getKeyCipher();
   public void setKeyCipher(Cipher keyCipher);
};
```

# C. Attacks: `javacard.security`

- `javacardx.crypto.KeyEncryption`
  - *methods to enable encrypted key data access to a key implementation*
- `javacard.security.Key`
  - *the key object factory*

```
public class KeyBuilder {
   public Key buildKey(byte keyType, short keyLength,
                       boolean keyEncryption);
};
```