Einführung in die Informationssicherheit

Florian Mendel

Institute for Applied Information Processing and Communications (IAIK)
Graz University of Technology
Inffeldgasse 16a, A-8010 Graz, Austria

http://www.iaik.tugraz.at/

L6 – Implementation Security

Einführung in die Informationssicherheit

Übersicht

- Implementation Security
  - Buffer Overflows
  - Unchecked Data Input
  - SQL Injection
- Side-Channel Attacks
  - Simple Power Analysis (SPA)
  - Differential Power Analysis (DPA)
  - Andere Attacken

Buffer Overflows (History)

- In den 70er Jahren wurde die Problematik von Buffer Overflows in der Programmiersprache C erstmals erkannt
- 1988 Morris Wurm verwendete einen Buffer Overflow im Unix-Programm finger um sich zu verbreiten
- Trotz ihrer langen Geschichte sind Buffer Overflows noch immer eine großes Sicherheitsproblem

Buffer Overflow Attack

Definition (Microsoft)

A buffer overflow attack is an attack in which a malicious user exploits an unchecked buffer in a program and overwrites the program code with his own data. If the program code is overwritten with new executable code, the effect is to change the program's operation as dictated by the attacker. If overwritten with other data, the likely effect is to cause the program to crash.

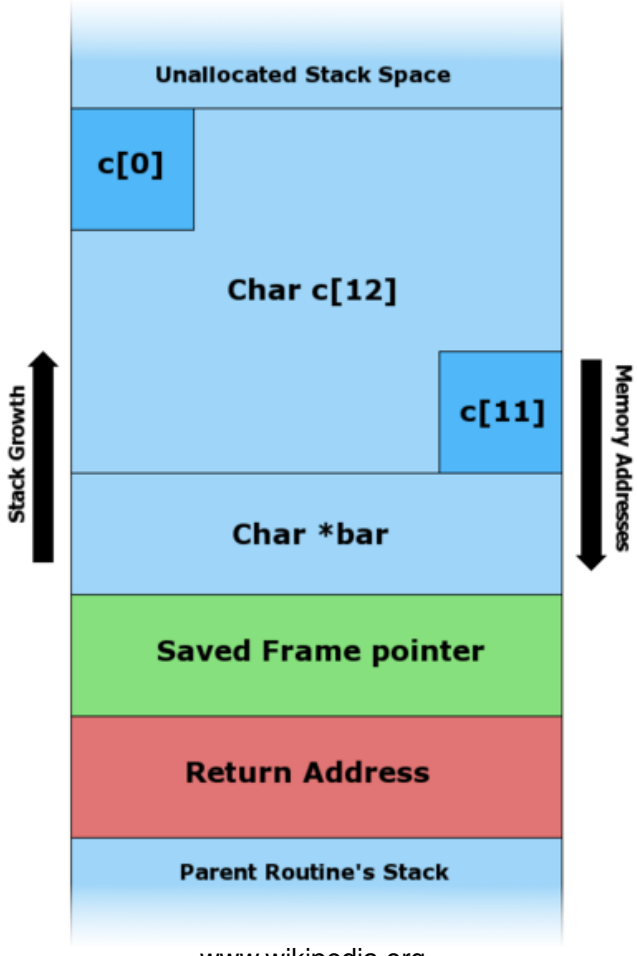
Beispiel

```
#include <string.h>

void foo (char *bar)
{
    char  c[12];

    strcpy(c, bar);  // no bounds checking
}
```

```
int main (int argc, char **argv)
{
    foo(argv[1]);
}
```

Program Stack

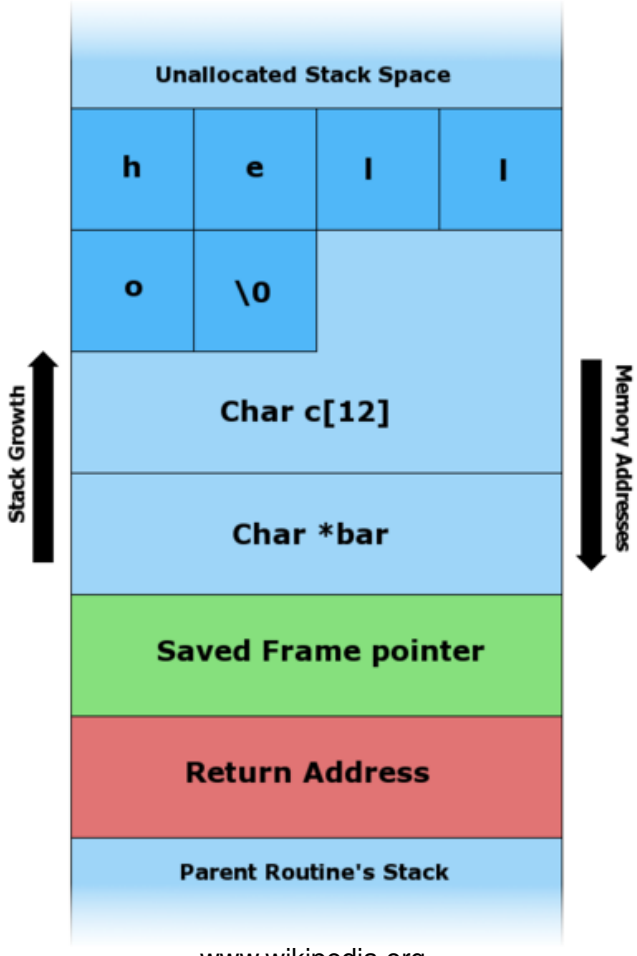
- Bevor Daten kopiert werden



www.wikipedia.org

Program Stack

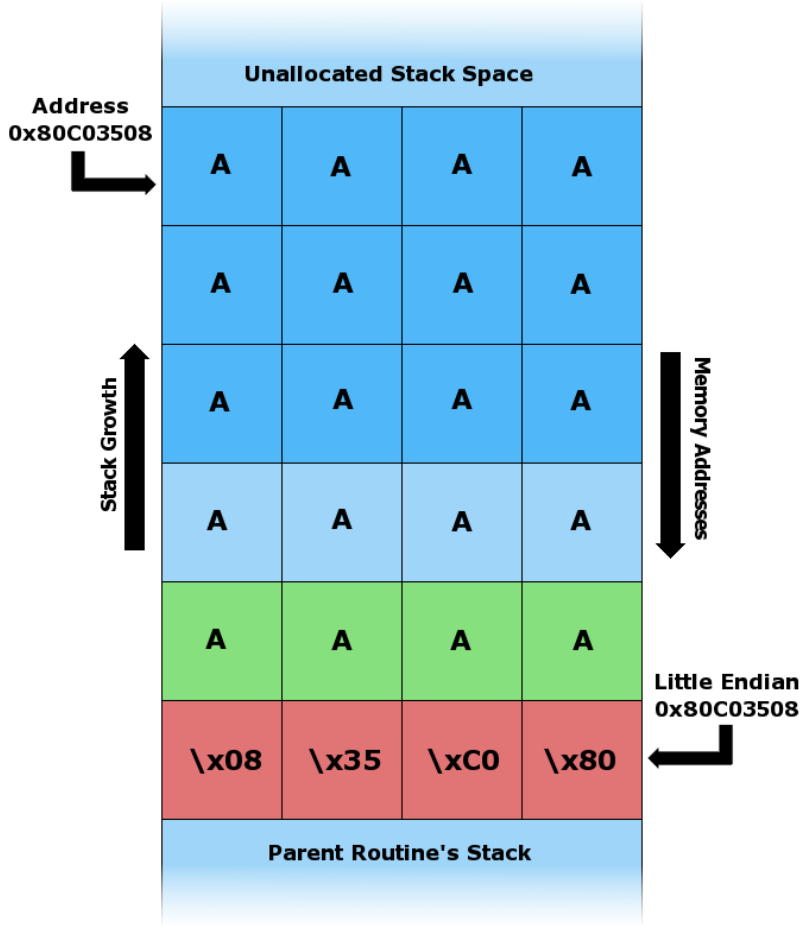
- `hello` ist das erste Argument



www.wikipedia.org

Program Stack

- `AA...A\x08\x35\xC0\x80` ist das erste Argument
- In der Praxis wäre `AA...A` geeigneter Shellcode



www.wikipedia.org

Buffer Overflow Attack

Notwendig für einen erfolgreichen Angriff:

- Vulnerability (Buffer Overflow) muss identifiziert werden
- Die Größe des Buffers muss bekannt sein
- Der Angreifer muss in der Lage sein, die Eingabedaten zu kontrollieren
- Die Return-Adresse muss ersetzt werden (control flow corruption)
- Eigenen Code einfügen (code injection)

Gegenmaßnahmen

- Verwenden Sie keine unsicheren Funktionen
  - ☒ strcpy, strcmp, ...
  - ☑ strncpy, strncmp, ...
- Verwenden von sicheren Bibliotheksmodulen
  - z.B. String-Klasse (bound checking)
- Quellcode Scan-Tools
  - z.B. IBM Rational PurifyPlus
- Compiler-Tools
- Verwenden Sie anderen Programmiersprachen ☺
  - Java, Perl, ...

Übersicht

- Implementation Security
  - Buffer Overflows
  - Unchecked Data Input
  - SQL Injection
- Side-Channel Attacks
  - Simple Power Analysis (SPA)
  - Differential Power Analysis (DPA)
  - Andere Attacken

Unchecked Data Input (Server side)

- Ungeprüfte Daten können verwendet werden, um Informationen über das System zu erlangen
  - Eine Mailing-Skript kann verwendet werden, um System-Dateien zu verschicken
  - Ein CGI-Skript kann auch verwendet werden, um Daten in System-Dateien hinzuzufügen
- Versteckte Felder sind nicht wirklich versteckt
  - Daten in diesen Feldern können leicht geändert werden
- Gegenmaßnahme
  - Entfernen von Meta-Zeichen aus der Eingabe des Anwenders

Unchecked Data Input (Client Side)

- JavaScript
  - Wird oft eingesetzt um die Eingabe des Uesrs zu überprüfen
  - Aber: JavaScript ist leicht zu umgehen
    - Laden Sie die Seite und entfernen Sie den Skript Teil
    - Deaktivieren Sie JavaScript in Ihrem Web-Browser
- Gegenmaßnahme
  - Überprüfung der Eingabe auch auf dem Server

Übersicht

- Implementation Security
  - Buffer Overflows
  - Unchecked Data Input
  - SQL Injection
- Side-Channel Attacks
  - Simple Power Analysis (SPA)
  - Differential Power Analysis (DPA)
  - Andere Attacken

SQL Injection

- Was ist SQL Injection?

- SQL Injection ist eine Technik, um SQL-Befehle in benutzerdefienierten Eingaben/Parametern einzubetten

- Das Ergebnis ist, dass ein Angreifer beliebigen SQL Abfragen oder Befehle auf dem Datenbankserver ausführen kann

Beispiel 1

```
String query =
"Select * from users WHERE user='" + user + "';"
```

Set user = a'; DROP TABLE users; –

```
Select * from users WHERE user='a';
DROP TABLE users;
--';
```

Beispiel 2

```
String query =
"Select * from users WHERE user='" + user + "'
and passwd='" + passwd + "';";
```

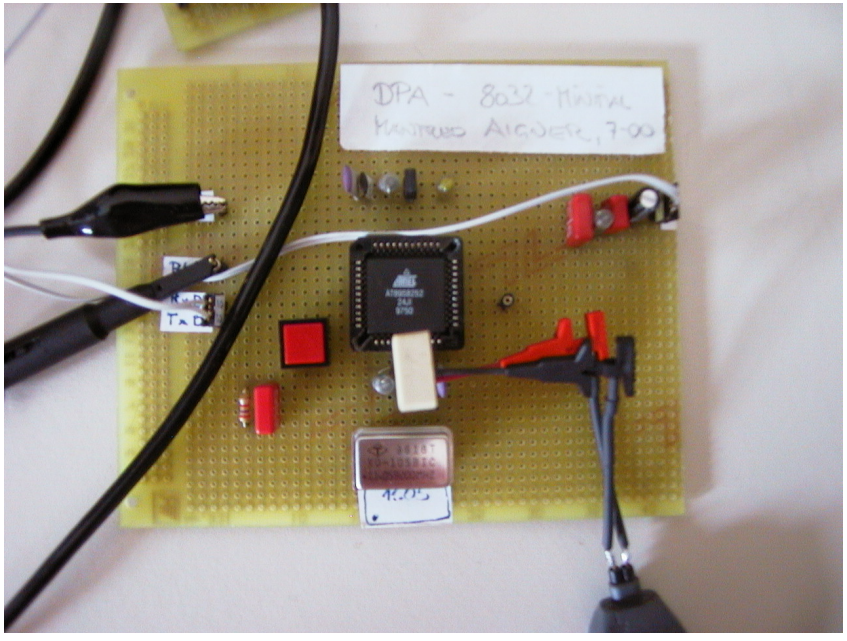
Set user = ' or 1 = '1 and passwd = ' or 1 = '1

```
Select * from users WHERE user='' or 1='1'
and passwd='' or 1='1';
```

Gegenmaßnahmen

- Filtern von “böse” Zeichen aus der Eingaben (Anführungsstriche, Schrägstriche, Beistriche, . . .)
- Überprüfen von numerischen Werten
- Benutzerberechtigungen einschränken
- Löschen von nicht verwendete Prozeduren
- . . .

Side-Channel Attacks



Überblick

- Konzept
- Simple Power Analysis (SPA)
- Differential Power Analysis (DPA)
- Andere Attacken

Prinzip

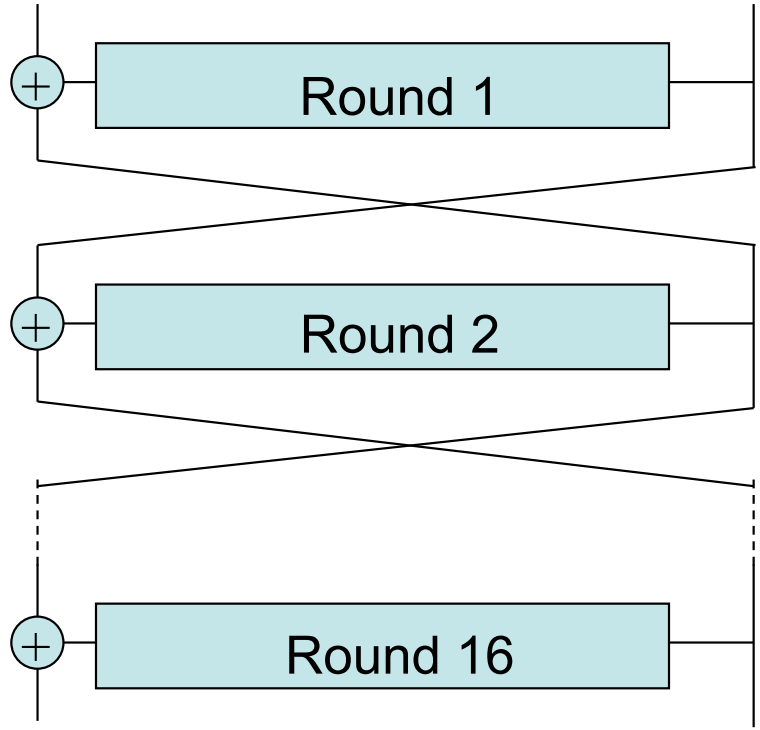
- ‘Pure’ Kryptoanalyse:
  - Beobachtet Ciphertexts, (Plaintexts)
  - Berechnet den geheimen Schlüssel
  - Rein mathematisch, Papier und Bleistift (naja)
- Seitenkanal-Kryptoanalyse:
  - Benutzen das Wissen um Zwischenwerten (interm. Variables)
  - Über Seitenkanäle
  - Real-life Szenarien

Warum funktioniert das?

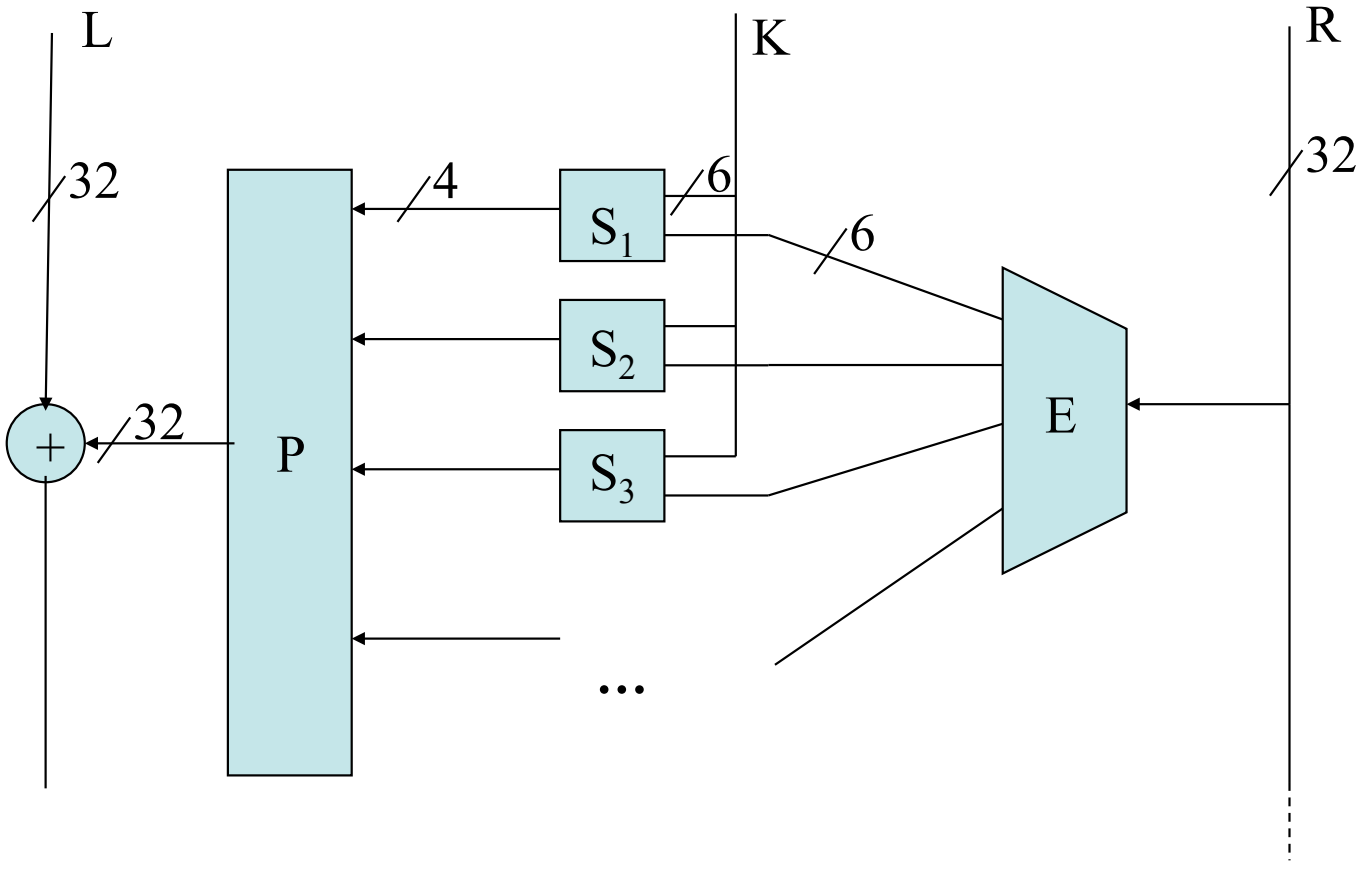
- Beispiel 1: 3-DES
- Product-Cipher:
  - Starke Cipher
  - Zusammengesetzt aus schwachen Komponenten
  - Seitenkanalattacken benutzen Information über Zwischenergebnisse die von den schwachen Komponenten kommen!

Beispiel 1: 3-DES

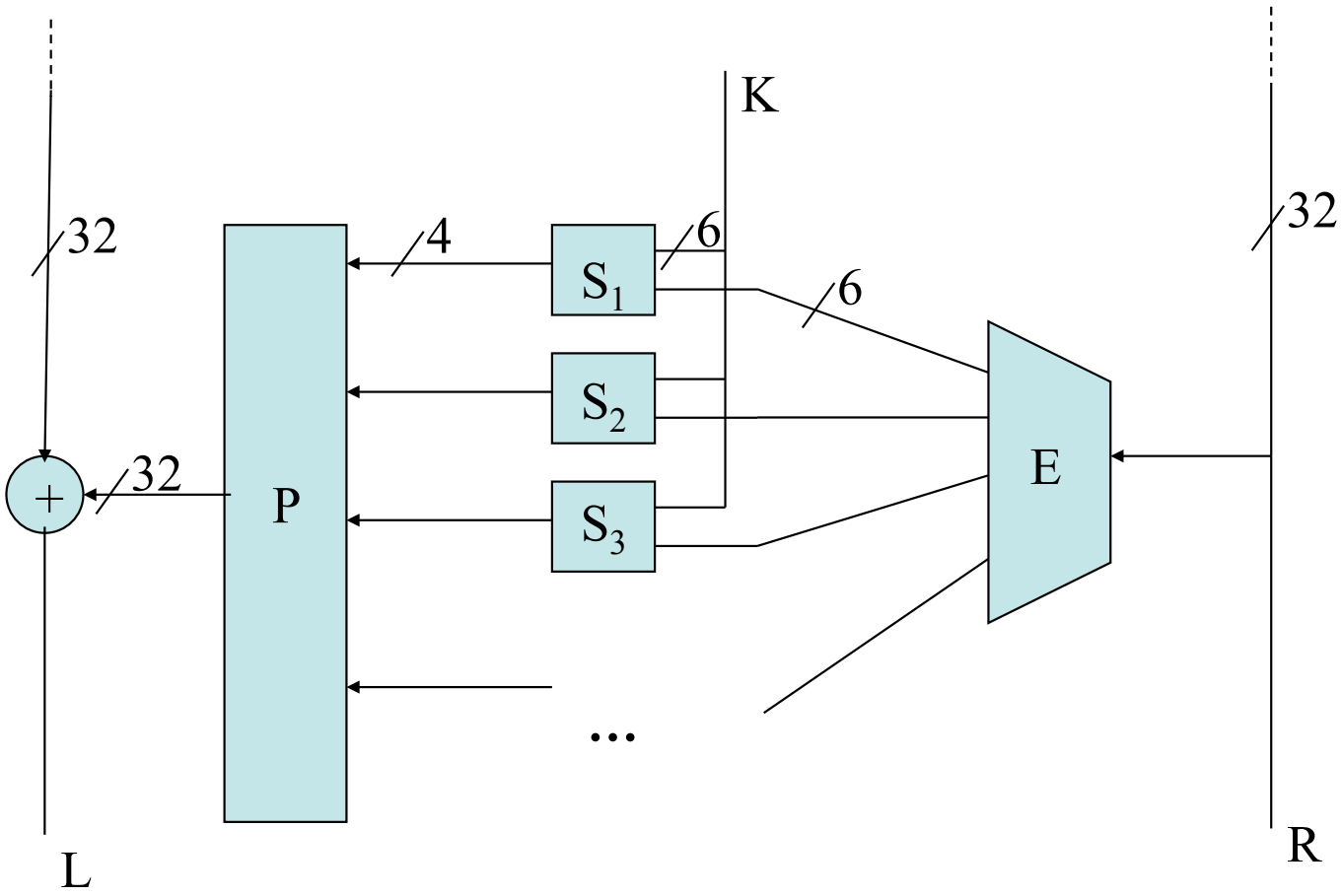
- Triple-DES:
  - Dreifache Anwendung mit untersch. Schlüsseln
- DES:
  - 16 Runden
  - Feistel Netzwerk
  - Jede Runde besteht aus mehreren kleinen Komp.
  - 56-bit effektive Key-Länge
  - 64-bit Input



Die erste Runde



Die letzte Runde

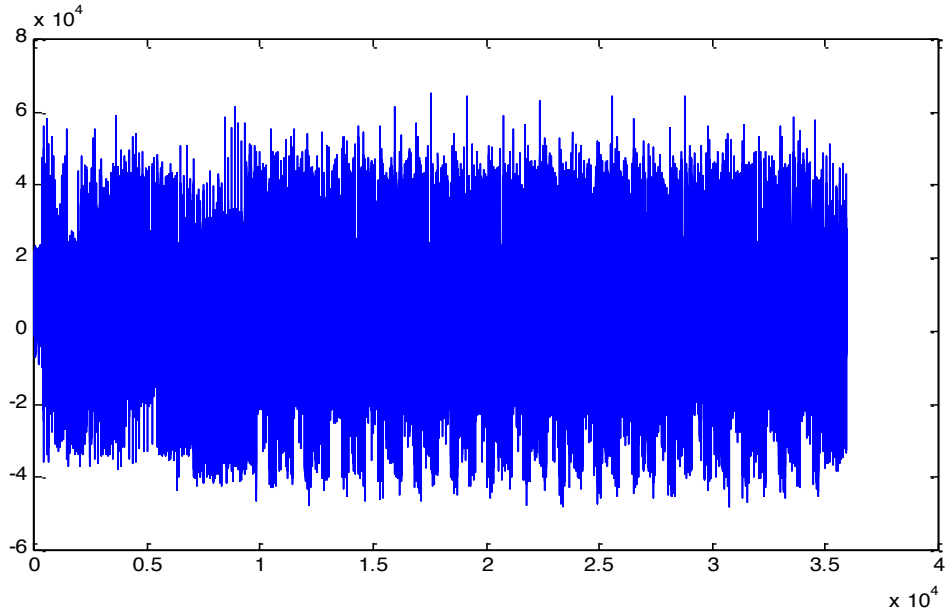


Iterierte Block-Ciphers

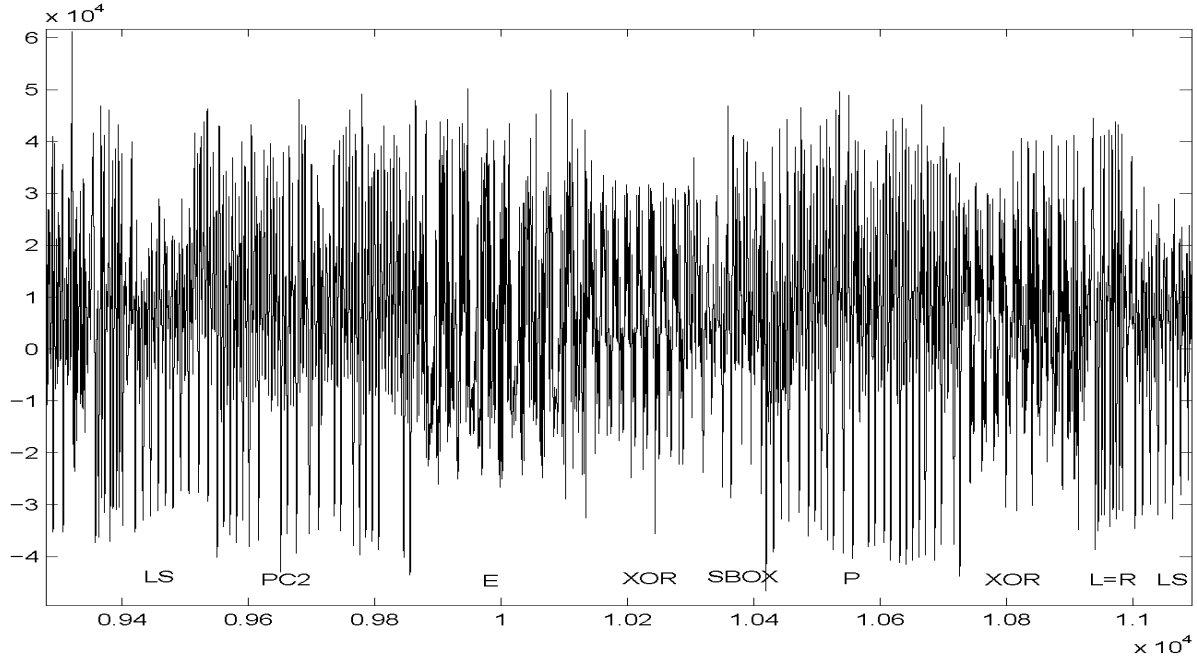
- Nach der ersten Runde hängt jedes Output-Bit nur von ein paar Plaintext-Bits und Key-Bits ab:
  - DES: 6 Plaintext-Bits & 6 Key-Bits
- Ähnlich bei Ciphertext: Die Inputs der letzten Runde können aus der Kenntnis von wenigen Ciphertext-Bits und Key-Bits berechnet werden
- Seitenkanal-Attacken funktionieren weil man Zwischenergebnisse vorhersagen kann indem man nur wenige Key-Bits erratet (durchprobiert).

Simple Power Analysis

- Ausführung einer Instruktion verbraucht Strom
- Untersch. Instr. verbrauchen untersch. viel Strom
- Beispiel: 16 Runden DES



Eine DES-Runde

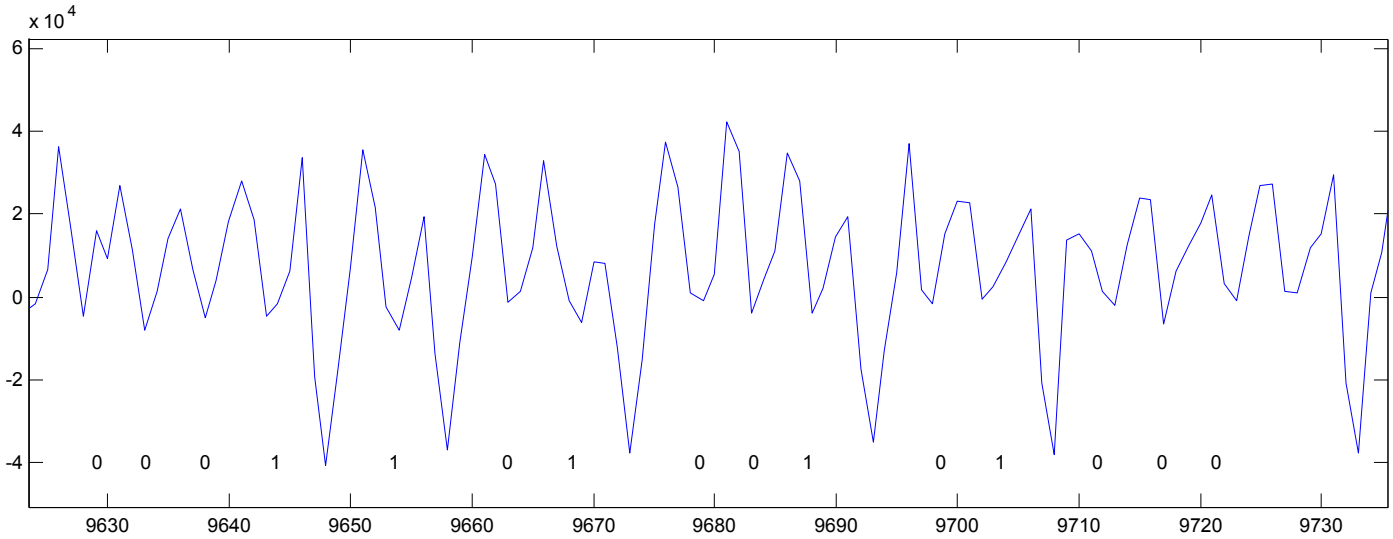


- Klar erkennbare Muster die zu den Komponenten einer Runde korrespondieren.

Zoom in eine der Permutationen

Permutation:

```
for i = 1 to 32
  if input[i] == 1              /* small valley */
    then out[P[i]] = 1          /* deep valley  */
```



Simple Power Analysis: Public Key Kryptosystem

- Public-key Kryptographie
  - Modulare Arithmetik (sehr große Zahlen) wird häufig eingesetzt
- Exponentiation mit geheimen Schlüssel (RSA-Sig) oder Multiplikation mit geheimen Skalar (ECC)
- Effiziente Variante: binary square-and-multiply

Modulare Exponentiation

Binary square-and-multiply algorithm:

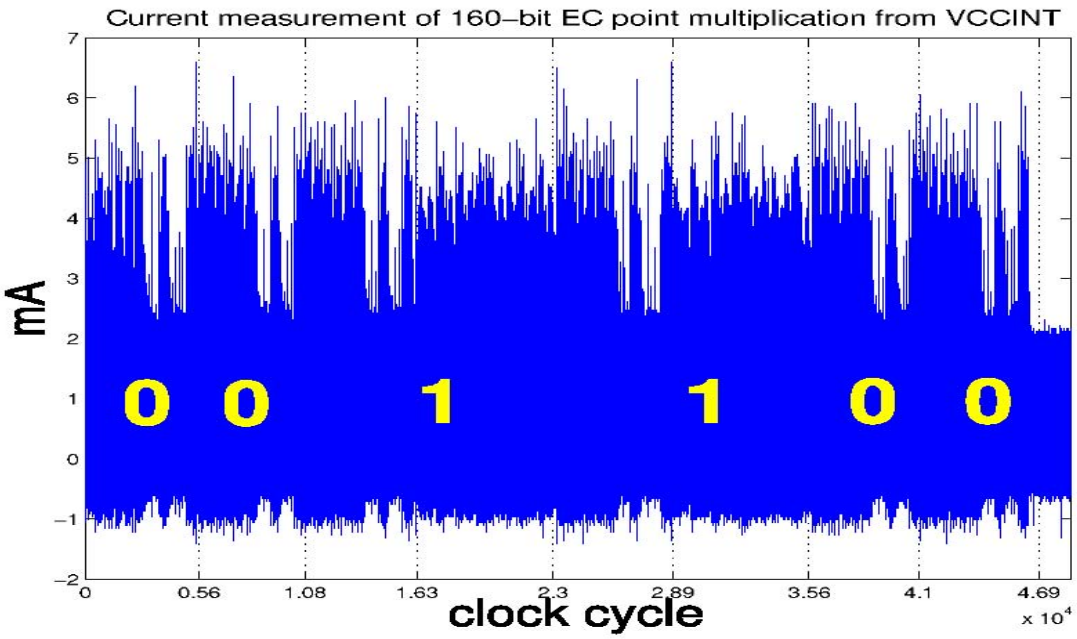
```
s = 1;
for k = w to 0
  s = s^2 mod n
  if (bit k of x) = 1
    then s = s * y mod n

return s        /* = y^x */
```

Example:

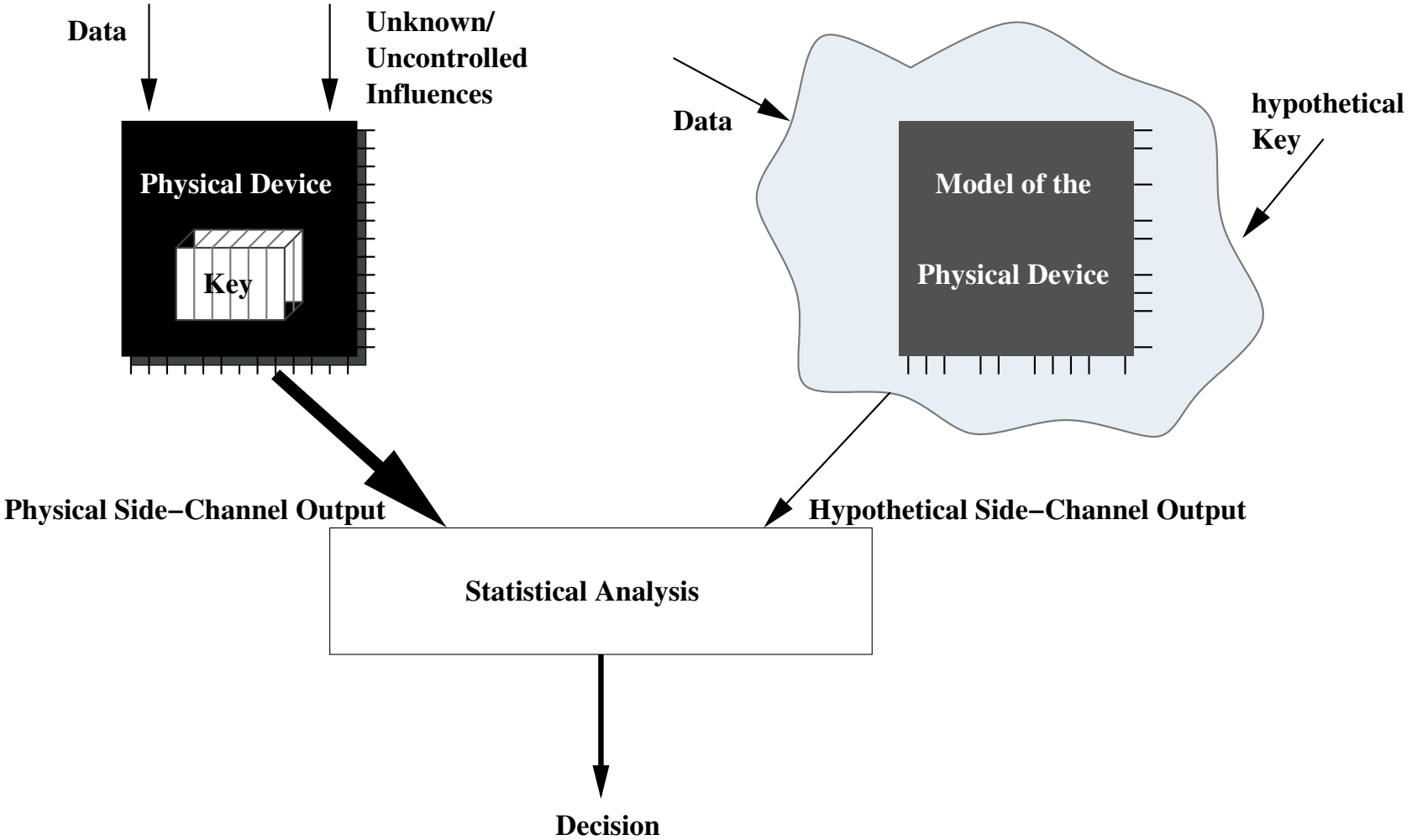
$y^{26} = ((((1^2 * y)^2 * y)^2)^2 * y)^2$ Binary(26) = 11010

Strom-Kurve

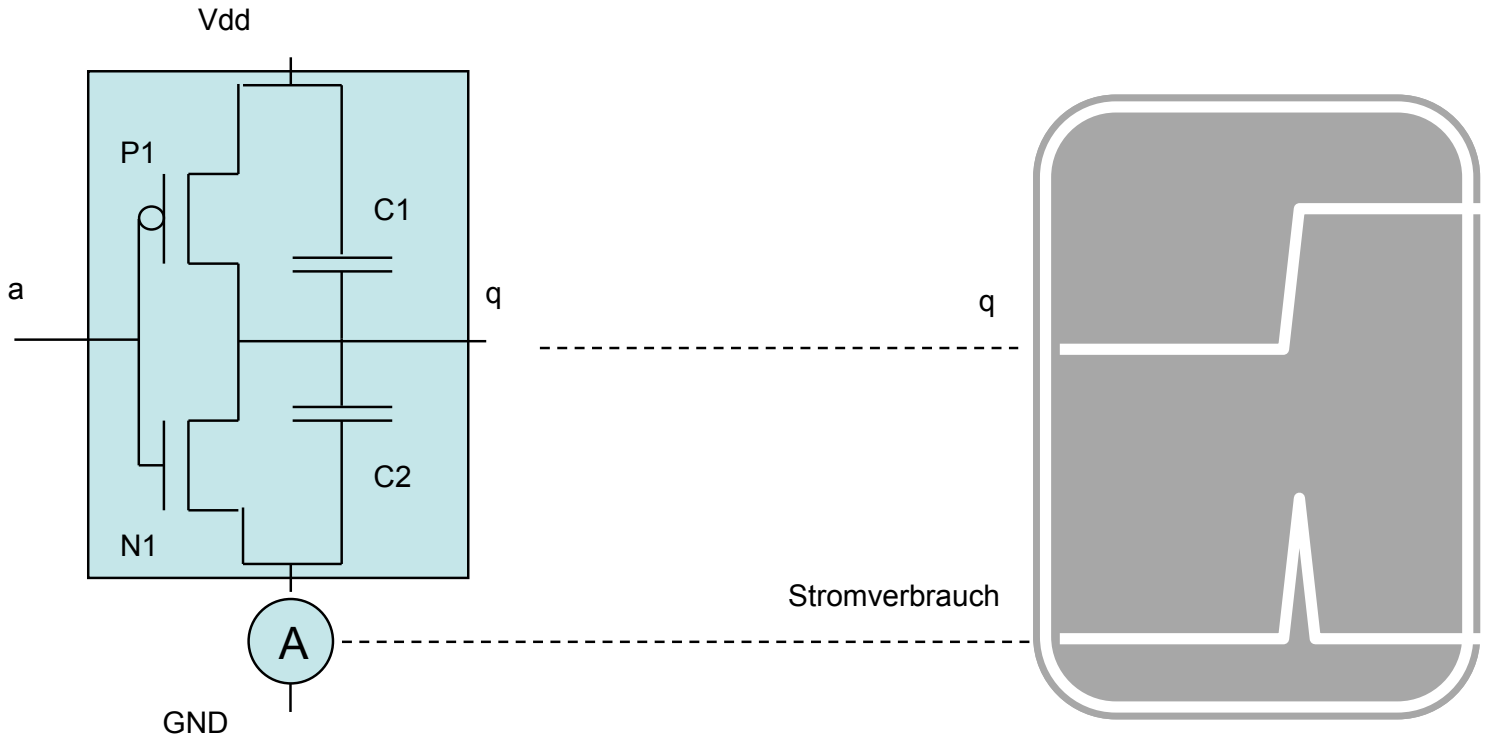
- Quadrieren/Verdoppeln in jedem Schritt
- Multiplikation/Addieren nur wenn geheimes Bit = 1
- Secret-Key direkt aus der Stromkurve



Differential Power Analysis



Power Analysis Attacken



Stromverbrauch eines CMOS Gatters hängt von den Daten ab:

- $q : 0 \rightarrow 0$ fast kein Stromverbrauch
- $q : 1 \rightarrow 1$ fast kein Stromverbrauch
- $q : 0 \rightarrow 1$ hoher Stromverbrauch (proportional zu C2)
- $q : 1 \rightarrow 0$ hoher Stromverbrauch (proportional zu C1)

Prinzip einer DPA-Attacke

- Auswahl des Zwischenergebnisses das man voraussagt
  - Bestimmen des Stromverbrauchs dieses Zw.Ergb. (Strommodell)
- Sammeln von Strommessungen für versch. Inputs (echtes Device)
- Man wählt einen Teil des Schlüssels k und berechnet den
- Stromverbrauch des Zw. Ergb. für diesen geratenen Schlüssel
- Berechne die Korreliertheit zwischen der tatsächlichen Messung und dem vorhergesagten Wert:
  - Distance-of-mean Test
  - Pearson's Korrelationskoeffizient

Prinzip: Wie benutzt man den Distance-of-Mean Test

- Für alle Werte des Teil-Schlüssels k:
  - Berechne Bit t
  - Gruppiere die Messungen in 2 Gruppen: $t = 0$, $t = 1$
  - Berechne den durschnittl. Stromverbrauch jeder Gruppe: P_0, P_1
  - Differenz der Durchschnitte

Prinzip: Wie benutzt man den Distance-of-Mean Test

- Idee: Wenn der Teilschlüssel k_i korrekt war:
  - Bit t ist korrekt
  - D.h. die Gruppierung in P_0 und P_1 war korrekt
  - Dann weicht der Durchschnitt der Gruppe P_0 stark vom Durchschitt der Gruppe P_1 ab

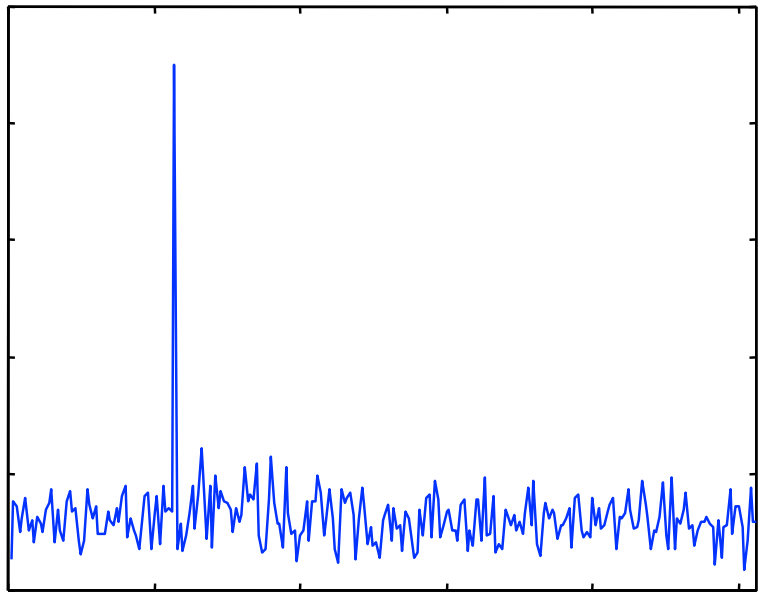
$\Rightarrow$ Größte Differenz erhält man mit korrektem k_i

Finden der Key-Bits

- Distance-of-mean Test
- Wenn die Schlüsselhypothese k korrekt ist, dann zeigt $|P_0 - P_1|$ den unterschiedlichen Stromverbrauch
- Wenn k NICHT korrekt ist, dann berechnet man das Bit t auch falsch:
  - Die Gruppierung ist falsch
  - P_0 und P_1 korrespondieren nicht zu $t = 0$ oder $t = 1$
  - P_0 und P_1 unterscheiden sich nicht so stark.

Finden des korrekten Schlüssels

- Durchführen des Distance-of-mean Tests für alle k_i
- Anschauen der resultierenden Differenz-Kurven
- Die Kurve mit dem höchsten Peak gehört zum richtigen Key
- Attacke benötigt wenig Wissen über die genaue Implementierung (im Gegensatz zu SPA)



Timing Attacken

- Voraussetzung: Störfreie Messungen der Execution-time einer Routine, für known/chosen Inputs
- Funktioniert wann immer die Execution-time vom geheimen Schlüssel abhängt:
  - DES bit Permutations
  - RSA Exponentiation mit geheimen Exponenten

Beispiel:

```
for i = 1 to 32
  if input[i] == 1
    then out[P[i]] = 1

Execution time: 32 'if' + hwt(input) * 'then'
```

Andere Seitenkanäle

- Timing
- EM radiation
- Noise, Lights, Errors . . .

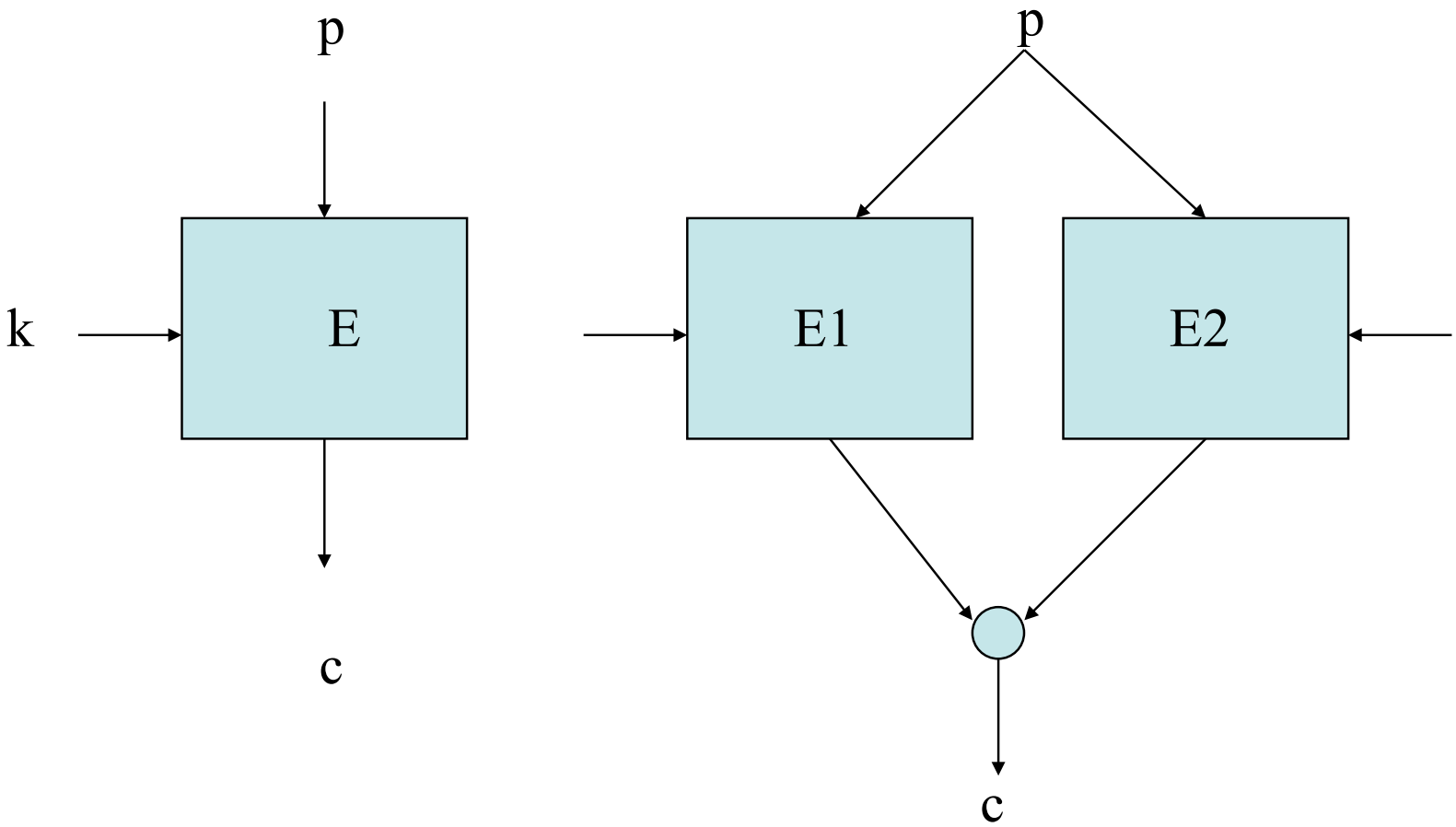
Aktive Implementierungsangriffe

- Aktive Attacken heißen oft auch fault attacks oder tamper attacks
- Bei einem Fault Attack werden Infos über das Geheimnis aus dem Output von fehlerhaften Berechnungen gewonnen
- Arten von Fehlerattacken:
  - Spike attacks
  - Glitch attacks
  - Optical attacks

Gegenmaßnahmen: Implementierungstricks

- “Playing dirty tricks”
  - Nutzlose Instruktionen
  - Nutzlose Blöcke die Strom verbrauchen
  - Nutzlose clock cycles
- Nicht schön im mathematischen Sinne ;-)
- Kein 100%iger Schutz
- Wenn clever gemacht:
  - Sehr billig
  - Sehr effektiv

Gegenmaßnahmen: Masking



Gegenmaßnahmen: Masking

- Mathematisch elegant
  - Wenn immer nur sehr kleine Shares benutzt werden
- Verspricht 100%igen Schutz
  - Leider nur theoretisch
- In der Praxis:
  - Sehr teuer
  - Mathematik bezieht sich auf idealisiertes Modell
    - Real world Implementierungen zeigen wieder Schwachpunkte

Schlussfolgerungen

- Attacken sind sehr relevant in der Praxis
  - Bis ins Jahr 2000 waren viele Smartcards im Umlauf die gegen solche Attacken anfällig waren
- 100%ig sichere Lösungen sind nur sehr schwer erreichbar
- Tradeoff zwischen Kosten und Nutzen

Vielen Dank für Ihre Aufmerksamkeit!