Seitenkanalangriffe verständlich erklärt

Hunz
cyber AT firlefa.nz

21. April 2018

Über mich

- Informatik-Studium in Augsburg
- danach bei Fraunhofer AISEC in Garching
- dort mit Seitenkanalangriffen auf Mikrocontroller gearbeitet / geforscht
- parallel privat eigene Analysesoftware entwickelt
- low-cost Messplatz zu Hause aufgebaut
- seit Anfang 2015 selbständig (HW/SW Entwicklung und IT Security)

Seitenkanäle - Beispiele

- Wasserdruck bei Halbzeitpause von Deutschlandspiel
- Nächtliche Pizzalieferungen ins Pentagon - Urban legend?
- Summen von Trafos is lastabhängig
- Licht, bzw. Fernseherflackern in Fenstern

Seitenkanäle in der IT

- Brute force: Alle Bits bzw. Zeichen müssen gleichzeitig richtig geraten werden
- bei 128Bit Schlüssel: $2^{128} = 340282366920938463463374607431768211456$ Möglichkeiten

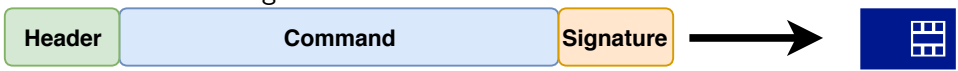
- Abkürzung durch Seitenkanalangriffe
- Teile des Schlüssels (z.B. einzelne Bits oder Bytes) lassen sich getrennt erraten
- Seitenkanal liefert Antwort ob Schlüsselteil korrekt geraten wurde
- Beispiel: bei 128Bit Schlüssel: Bit 1 erraten, Bit 2 erraten, ...
  $\rightarrow 128 * n$ Versuche

- n in der Praxis meist größer als 1 wegen Rauschen
- selbst bei $n = 10^6$ nur 128 Mio Versuche vs.
  340282366920938463463374607431768 Mio bei brute force

Timing-Angriffe

- Seitenkanal: Messung der Ausführungszeit
- Angreifer schickt Anfrage an System, misst Zeit bis zur Antwort

- Beispiel: PayTV-Smartcard
- Kommandos an Smartcard müssen signiert sein
- HMAC-Verfahren - Smartcard und PayTV-Anbieter kennen beide geheimen (symmetrischen) Schlüssel
- Signatur über Smartcard-Kommando, Signatur wird an Kommando angehängt und an Karte übertragen



- Karte erkennt an der Signatur, ob Kommando mit geheimen Schlüssel signiert wurde

Signaturprüfung

- Karte empfängt Daten
- Karte berechnet selbst Signatur über Kommando mit geheimen Schlüssel
- Karte vergleicht empfangene Signatur mit der berechneten

```
int result = memcmp(rcvd_signature, good_signature, signature_size);
```

- Bei Übereinstimmung (result=0) ist das Kommando authentisch

memcmp

Naive memcmp Implementierung:

```
int memcmp(uint8_t *a, uint8_t *b, size_t n) {
  for(;n;n--,a++,b++) {
    int diff = *a - *b;
    if (diff)
      return diff;
  }
  return 0;
}
```

- Pointer auf zwei Buffer **a** und **b**
- Jeweils ein Byte aus **a** und **b** laden und vergleichen
- Sobald Unterschiedliche Werte Unterschied zurückgeben
- Ansonten mit den nächsten Bytes fortfahren
- Stimmen alle überein wird 0 zurückgegeben

memcmp - Timing Analyse

- Schleife wird nur so lange ausgeführt wie Bufferinhalte übereinstimmen
- memcmp bricht bei erstem Unterschied ab
- Je früher ein Unterschied in den Daten, desto schneller ist memcmp fertig
  → **Laufzeit von memcmp ist datenabhängig!**
- Angreifer kann Zeit messen um richtig geratene Bytes zu erkennen
  → **Timing-Angriff**

- Grundsätzlich: **Kein Bug** in memcmp, sondern **erwünschtes Verhalten**
- **Grund: Performance-Gewinn** bei Vergleich großer Datenmengen

Realer Angriff auf memcmp timing

- Dieser Angriff wurde früher auf Premiere-Karten durchgeführt
- Karte wurde über seriellen Port mit Rechner verbunden
- Antwortzeit der Karte wurde mittels DOS-Programm gemessen
- Für jedes Byte Werte 0..255 durchprobieren
- Der Wert mit der längsten Antwortzeit ist korrekt

Timing Demo

- UDP Kommunikation mit localhost
- Server vergleicht String mit Referenz (HMAC, Passwort, ...)
- Server antwortet ob Strings übereinstimmen
- Client misst Antwortzeit

Spectre/Meltdown

- Spectre und Meltdown sind ebenfalls Timing-Angriffe
- Vergleichbar mit memcmp-Beispiel
- Zielen auf Prozessorhardware statt Software
  $\rightarrow$ grundsätzlich erstmal alle Betriebssysteme betroffen
- Grund: Optimierungen mit Ziel Geschwindigkeitszuwachs führen zu datenabhängigen Timingunterschieden

Warum?

CPUs

- Sehr, sehr schnelle Datenverarbeitung erkauft durch sehr hohe Komplexität
- **Takt lässt sich nicht beliebig erhöhen**
- **Engpass: RAM-Anbindung**
- RAM-Zugriffe dauern unglaublich lange, CPU muss dann warten
- Betrifft sowohl Code, als auch Daten

Wie baut man schnelle CPUs?

- mehrere Kerne (mehrere Threads werden parallel ausgeführt)
- **Simultaneous Multithreading**: Mehrere Threads auf einem Kern
- **Superskalarität**: CPUs können mehrere Befehle parallel abarbeiten
- **out-of-order execution**: Unabhängige Befehle werden parallel ausgeführt
- Häufig benötigte Daten in **Caches** innerhalb der CPU halten (schneller Zugriff)
- **Vorhersagen** welche Daten benötigt werden und diese Daten vor-laden
- **Spekulative Ausführung**: Code dessen Ausführung von bisher noch unklarer Vorbedingung abhängt wird auf Verdacht ausgeführt. Ergebnis wird übernommen oder verworfen wenn klar ist, ob Vorbedingung zutrifft.
  Beispiel:

  ```
  if (vorbedingung) { ... }
  ```

Spectre Kurzfassung

- Angriff nutzt **spekulative Ausführung** und **Caches**

- Durch spekulative Ausführung lädt Code Daten A in den Cache auf die er gar nicht zugreifen dürfte
- Basierend auf Wert A werden weitere Daten B in den Cache geladen (auf Daten B darf der Code regulär zugreifen)
- Welche Elemente von B geladen werden hängt von Daten A ab
- Vorbedingung erweist sich nach Ausführung als falsch
  $\rightarrow$ keine Exception da Code ja gar nicht ausgeführt werden sollte
- Daten bleiben trotzdem in den Caches
- Code greift auf alle möglichen Elemente von B zu, misst Zugriffszeit
  $\rightarrow$ kurze Zugriffszeit: Daten waren schon im Cache
- Aus den gecachten Daten B lassen sich die Daten A rekonstruieren

Meltdown Kurzfassung

- **Spectre**: Nur Zugriff auf Speicher **innerhalb eines Prozesses** (z.B. Webbrowser)
- **Meltdown**: Zugriff auf Speicher von **anderen Prozessen** (z.B. Kernel)
- Meltdown nutzt **out-of-order execution** und **Caches**

- Zugriffsversuch auf Datum A aus fremden Speicher führt zu Exception
- Durch out-of-order execution werden dennoch weitere Befehle ausgeführt (die Ergebnisse dieser Befehle werden später verworfen)
- Basierend auf Wert A werden weitere Daten B in den Cache geladen (auf Daten B darf der Code regulär zugreifen)
- Code greift auf alle möglichen Elemente von B zu, misst Zugriffszeit $\rightarrow$ kurze Zugriffszeit: Daten waren schon im Cache
- Aus den gecachten Daten B lässt sich Wert A rekonstruieren

Gegenmaßnahmen?

- Zusätzlich zufällige Wartezeit einbauen
- memcmp-Beispiel:

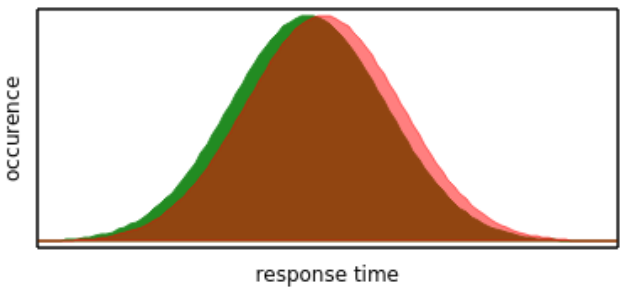
```
int result = memcmp(rcvd_signature, good_signature, signature_size);
usleep(getrandom());
...
```

NOPE!

- Zufällige Wartezeit gibt es bereits durch Messungenauigkeit, Taskwechsel, etc.
- Löst das Problem nicht, erhöht nur die Anzahl der Messungen die Angreifer machen muss

Warum?

- Gauß-Verteilung der Messwerte
- Messwert-Histogramm (auch Mittelwert) für 'schnelle' Antworten hebt sich dennoch vom Rest ab



Neuer Versuch

```
uint8_t memcmp_consttime(uint8_t *a, uint8_t *b, size_t n) {
  uint8_t diff = 0;
  for(;n;n--,a++,b++)
    diff |= (*a) ^ (*b);
  return diff;
}
```

- Gibt 0 zurück wenn Daten A und B gleich
- konstante Laufzeit
- Probleme?

Neuer Versuch

```
uint8_t memcmp_consttime(uint8_t *a, uint8_t *b, size_t n) {
  uint8_t diff = 0;
  for(;n;n--,a++,b++)
    diff |= (*a) ^ (*b);
  return diff;
}
```

- Gibt 0 zurück wenn Daten A und B gleich
- konstante Laufzeit
- Probleme?
- Zukünftige Compileroptimierungen?

I have a bad feeling about this...

```
uint8_t memcmp_consttime(uint8_t *a, uint8_t *b, size_t n) {
  uint8_t diff = 0;
  for(;n;n--,a++,b++)
    diff |= (*a) ^ (*b);
  return diff;
}
```

Zukünftige Optimierungen (1):

- Nutzung der Funktion:

  ```
  if(memcmp_consttime(a,b,n)) { ... } else { ... }
  ```

- If prüft nur ob Rückgabewert 0 oder nicht 0 ist
- Genauer Wert nicht relevant
- Compileroptimierung: Abbruch der Schleife in memcmp_consttime sobald diff nicht mehr 0 ist

I have a bad feeling about this... (2)

```
uint8_t memcmp_consttime(uint8_t *a, uint8_t *b, size_t n) {
  uint8_t diff = 0;
  for(;n;n--,a++,b++)
    diff |= (*a) ^ (*b);
  return diff;
}
```

Zukünftige Optimierungen (2):

- Wann lässt sich for-Schleife vorzeitig abbrechen ohne dass sich Rückgabewert der Funktion ändert?

I have a bad feeling about this... (2)

```
uint8_t memcmp_consttime(uint8_t *a, uint8_t *b, size_t n) {
  uint8_t diff = 0;
  for(;n;n--,a++,b++)
    diff |= (*a) ^ (*b);
  return diff;
}
```

Zukünftige Optimierungen (2):

- Wann lässt sich for-Schleife vorzeitig abbrechen ohne dass sich Rückgabewert der Funktion ändert?
  $\rightarrow$ **Sobald diff = 0xff ist**
- Optimierung durch zukünftige Compilerversion kann nicht ausgeschlossen werden

Zwischenfazit

- Plötzlich Seitenkanal-Angriff gegen eigentlich gehärteten Code möglich
- Wie lässt sich sowas automatisiert testen/erkennen bzw. verhindern?

Resistenz gegen solche Angriffe hängt ab von:

- Anwendungsspezifischem Code
- Compiler, Assembler, Toolchain
  (auch Linker - legt fest wo Daten im Speicher liegen $\rightarrow$ Zuordnung zu Cachelines)
- Hardware

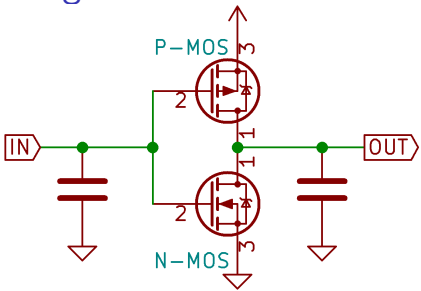
Was tun?

- etwas besser handhabbar mit lowlevel-Programmiersprachen
- Spezielle Bibliotheken?
  (gibt es teilweise schon, aber ausreichend für Zukunft?)
- Marker für Funktionen die konstante Laufzeit haben sollen?
- Spezielle Hardware? (z.B. AES-NI)

Weitere Seitenkanäle

- elektromagnetische Abstrahlung
- Leistungsaufnahme
- DPA/CPA: Angriffe auf kryptografische Verfahren (z.B. AES, RSA, ECC, ...)

Leistungsaufnahme

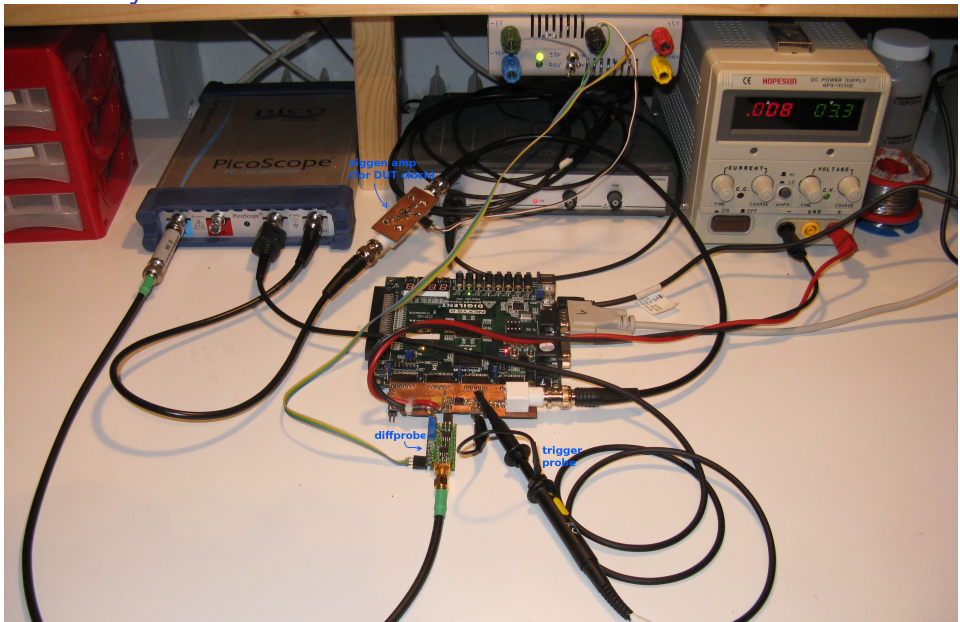


- Heutige Digitaleketronik in CMOS-Technologie
- komplementäre Feldeffekt-Transistoren
- nahezu kein Ruhestrom
- **Stromfluss bei Änderung von Werten**
- **Umladevorgänge** (z.B. der Gate-Ladung)
- Je mehr Bits sich in einem Takt ändern, desto mehr Strom fliesst
- annähernd linearer Zusammenhang zwischen Anzahl der geänderten Bits und Stromfluss

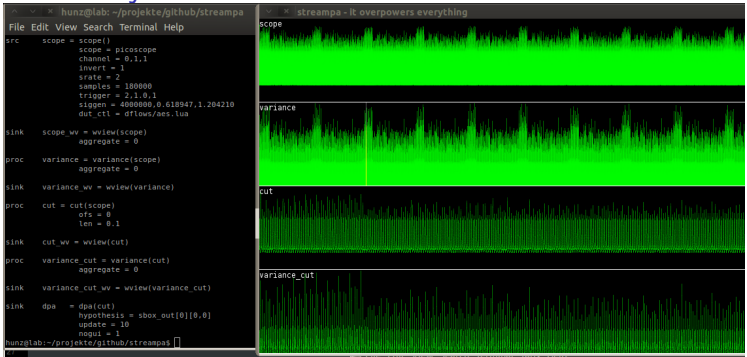
Differential/Correlation Power Analysis

- Stromfluss lässt sich mit Speicheroszilloskop aufzeichnen
- Wieder viele Durchläufe (einige tausend bis Millionen), dabei Aufzeichnung der Stromaufnahme
- Entweder Klartext oder Ciphertext muss bekannt sein (und sich ständig ändern)
- Bekannter Cipher-/ oder Plaintext, Annahme Keybit[1] $= 0$
- Damit lassen sich Zwischenwerte bei der Ver-/Entschlüsselung vorhersagen
- Basierend auf Zwischenwerten wird Stromverbrauch vorhergesagt
- Kontrolle wie gut das mit realem Stromverbrauch übereinstimmt mit **Pearson-Korrelation**
- So lange durchführen bis Schlüssel rekonstruiert

Power Analysis Messaufbau



Power Analysis Demo




```
function init()
    atr = iso7816.reset()
    print ("ATR: " .. tohex(atr))

    -- setkey
    sw = iso7816.apdu(fromhex("80 30 00 00 10 11 22 33 44 55 66 77 88
    print ("SW: " .. tohex(sw))
end

require "iso7816"
require "math"

iso7816.new("fpga:/dev/ttyS0")

init()

while true do
    rnd = prng(16)
    -- encrypt w/ trigger enable
    sw = iso7816.apdu(fromhex("80 32 00 00 10") .. rnd, true)
    if (sw == nil) or (sw ~= fromhex("90 00")) then
--      init()
    else
--      print ("SW: " .. tohex(sw))
        stream_commit{indata = rnd}
    end
end
```

```
hunz@lab:~/proj... ⌘  hunz@lab:~/proj... ⌘  hunz@lab: ~/proj... ⌘  hunz@lab: ~
node scope (src scope) created @0x8aa63e8
node scope_wv (sink wview) created @0xb6f00fd0
node variance (proc variance) created @0xb6f42780
node variance_wv (sink wview) created @0xb6f42898
node cut (proc cut) created @0xb6f42b20
node cut_wv (sink wview) created @0xb6f42c48
node variance_cut (proc variance) created @0xb6f61898
node variance_cut_wv (sink wview) created @0xb6f619d8
node dpa (sink dpa) created @0xb6f62fc8
ATR: 3b 00
SW: 90 00
sbox_out[0][0,0]
OPA init (256 hypotheses, tracelen 18000)
memory usage: 17 MByte
[    10] max_hypo: 25 val: 0.964773 / signif: 1.644384
[    20] max_hypo: a0 val: -0.866666 / signif: 1.162755
[    30] max_hypo: 61 val: -0.769309 / signif: 0.949386
[    40] max_hypo: 61 val: -0.696236 / signif: 0.822192
[    50] max_hypo: 11 val: -0.658481 / signif: 0.735391
[    60] max_hypo: 11 val: -0.656230 / signif: 0.671317
[    70] max_hypo: 10 val: -0.640474 / signif: 0.621519 !
```

Fazit

- Seitenkanalangriffe sind keine trivialen Angriffe
- Meist sehr viele Messdurchläufe nötig
- Aber deutlich weniger als bei brute force
- Schlüssel muss nicht auf einmal korrekt erraten werden
- stattdessen Erraten von Teilen eines Schlüssels möglich

- Sehr komplexes Thema
- Schutz gegen Seitenkanalangriffe aufwendig
- Kleinste Fehler machen Schutz unwirksam
- Optimierung bzgl. Geschwindigkeit vs. Sicherheit gegen Seitenkanalangriffe