On-chip ⇒ de-packaging

- Attacks on silicon are usually expensive but not that much...
  - De-packaging (mechanical, chemical actions) without damaging the device
  - Re-connecting in a new package

Reverse engineering

- Layout reconstruction (microscope + time and skill)
- Removal of metal layers (HF acid) necessary to reconstruct deep layers

Source: Oliver Kömmerling, Marcus Kuhn
Memory content recovery

ROM content reconstruction techniques

- Implant-mask layout of a NAND ROM, made visible by a dopant-selective crystallographic etch. 16 × 14 bits plus parts of the row selector of a ROM. The threshold voltage of 0-bit P-channel transistors (stained dark here) was brought below 0 V through ion implantation.

Source: Oliver Kömmerling, Marcus Kuhn

- Power off imaging identifies active areas
- Power on imaging distinguishes closed and opened transistor channels

Source: Samyde, Skorobogatov, Anderson and Quisquater

Memory content recovery

Electromagnetic investigation

- Use of a miniature inductor wrapped around a microprobe needle
- Induce eddy currents in silicon and sense them to build a map
- Alternatively induce very small perturbations and observe different behaviour for 0 and 1

Source: Samyde, Skorobogatov, Anderson and Quisquater

- Laser or electromagnetic investigation is slow
  - Suitable when freezing the chip is an option
  - Secured devices sometimes prevent this with low clock frequency detectors
  - Freezing a static RAM when power is off allows to maintain the initial content for minutes or hours

Lest We Remember: Cold Boot Attacks on Encryption Keys
On-chip probing

- Laser attack passivation layers
- Microscope (UV for deep sub-micron process)
- Micropositioner (sub-micron)
- Microprobe (< 100 nm) connected to DSP card

9 horizontal bus lines on a depackaged smartcard processor. A UV laser (355 nm, 5 ns) was used to remove small patches of the passivation layer over the eight data bus lines to provide for microprobing access.

Source: Oliver Kömmerling, Marcus Kuhn

Memory Read-out

- Direct drive and record usually not practical (too many probes)
- Attacked chip drives address bus
  - Some implementation prevent access to all memory locations (periodically switch between multiple keys)
  - Simple modifications of decoding logic can prevent branches and jumps
- Probe some data bus lines
- Replay same sequence and change probed bus lines
  - Replay can be prevented by randomization
  - Simplified by naïve memory integrity checking after reset

Deep sub-micron ⇒ new tools

- Probing attacks become very expensive
- Focused Ion Beam (FIB)
  - 5 to 10 nm diameter
  - Beam currents from 1 pA to 10 nA
  - Very high resolution imaging
  - Material removal with very high precision
  - Combined with gas injectors material deposition (platinum or insulator)
  - Create new pads
    - Drill a hole to a metal line
    - Fill with platinum
    - Create pad on surface for easy access
  - Cost: from 500 k-euros to several M-euros
  - FIB time can be rented at much lower cost
  - Can be used to repair the test circuitry (fused after manufacturing and test) and dump memory (more on this later)

Deep sub-micron ⇒ new tools

- Electron Beam Testers (EBT)
  - Scanning Electron Microscopes (SEM) with voltage-contrast function
  - Primary electrons accelerated with voltages about 2.5 kV
  - Beam currents of about 5 nA
  - Secondary electrons recorded, number and energy indicates local electric fields on surface
  - Relatively low bandwidth (few MHz)
    - Slow down clock frequency
    - Generate periodic transactions and average to reduce noise
  - Frequently associated with a FIB for chip imaging and editing
Deep sub-micron ⇒ new tools

Countermeasures

- Random place and route to scatter the registers across the chip
  - More difficult to retrieve stored values
- Encipher the content of memories
  - More difficult to retrieve stored values
  - But this should not impose penalties on performances
- Use sensors (low and high temperatures, UV, infra-red, X-rays, ionising radiations, clock frequency, voltage, ...)
  - But the device still need a power source to react when intrusion is detected
  - For non self-powered devices (smartcards) the sensors could be destroyed when power is off ⇒ they must be tested periodically in normal operation

Countermeasures

- Introduce randomness in timings:
  - Dummy clock cycles populated with dummy computations
  - Randomized multithreading with duplicated register banks to speed-up the context switch
- Really destroy test circuitry
  - Test on wafer
  - Test circuitry cut off by wafer saw

Countermeasures

- Use restricted program counter (Page + Offset)
  - Prevent the use of PC to dump memory
  - Force to have a jump at least every $N$ (small) instructions
  - Reset system when offset counter wraps around
- Add top layer sensor meshes
  - Additional metal layer used to form a mesh on top of secured areas
  - Continuously monitored during operation
  - But FIB and EBT can be used efficiently against meshes...
Probing attack on modular exponentiation

- Let $P_k$ be the value of the probed bit of $A$ at end of iteration $k$
- Let $(P_0, ..., P_k)$ be the series of $k + 1$ first probed values
- Let $D(0...k)$, $0 \leq k < w$ be the $k + 1$ leading (leftmost) bits of $D$
- For a given hypothesis $\delta$ on $D(0...k)$
  - Attacker can simulate SM-1 algorithm for $k + 1$ first iterations...
  - ...including the probe outcomes $(P_0, ..., P_k)(\delta)$...
  - ...that is, what the probe should output if $D(0...k) = \delta$
- If $(P_0, ..., P_k)(\delta) = (P_0, ..., P_k)$, $\delta$ is plausible...
- ...else $\delta$ is incorrect and shall be discarded
- Note: plausible does not mean correct
  - Simulated and real probed values can coincide by accident...
  - ...while $\delta \neq D(0...k)$

Algorithm 1 Modular exponentiation

1. $A_{-1} \leftarrow 1$ \hspace{1cm} $\triangleright$ initial value of accumulator
2. for $k = 0, w - 1$ do \hspace{1cm} $\triangleright$ loop from MSB (bit #0) to LSB (bit #w-1) of $D$
   3. $B_k \leftarrow R_k^2 \mod N$
   4. if $D(k) = 1$ then \hspace{1cm} $\triangleright$ modular square
      5. $A_k \leftarrow (B_k \times M) \mod N$
   6. else
      7. $A_k \leftarrow B_k$
   8. end if
9. Probe \hspace{1cm} $\triangleright$ Probe and record $P_k = A_k(i)$
10. end for
11. return $A_{w-1} = M^D \mod N$

Algorithm 2 Attack algorithm

1. $\Delta_0 \leftarrow \{()\}$ \hspace{1cm} $\triangleright$ Start with empty bit-string
2. for $k = 0, w - 1$ do \hspace{1cm} $\triangleright$ loop from MSB to LSB of $D$
   3. $\Gamma_k \leftarrow \{(\delta \& 0, \delta \& 1) \mid \delta \in \Delta_{k-1}\}$ \hspace{1cm} $\triangleright$ Expand with two possible next bits
   4. $\Delta_k \leftarrow \{(\delta \& 0, \delta \& 1) \mid (P_0, ..., P_k)(\delta) = (P_0, ..., P_k)\}$ \hspace{1cm} $\triangleright$ Filter out non-plausible
   5. end for
6. return $\Delta_{w-1}$ \hspace{1cm} $\triangleright$ Final remaining plausible guesses for $D$

- Let $\Delta_k$ be the set of the remaining plausible guesses $\delta$ after iteration $k$
- Let $S \& b$ be the right-concatenation of bit $b$ to bit-string $S$
- Let $( )$ be the empty bit-string

Will the number of remaining guesses ($|\Delta_k|$) explode?
Probing attack on modular exponentiation

1: $\Delta_{w-1} \leftarrow \{()\}$  
2: for $k \leftarrow 0, w-1$ do  
3: $\Delta_k \leftarrow \{\delta \in \Delta_{k-1} \mid \delta \in \Delta_{k-1}\}$  
4: $\Delta_k \leftarrow \{\delta \in \Delta_k \mid \{P_0, \ldots, P_k\} = \{P_0, \ldots, P_k\}\}$  
5: end for  
6: return $\Delta_{w-1}$

Exercise #1: Calculate the probability $\epsilon = P(\delta \in \Delta_k \mid \delta$ is wrong) that a wrong guess $\delta$ survives the filtering stage (line #4) of the attack (assume bits $P_k$ and $\hat{P}_k$ are uncorrelated)

Exercise #2: Calculate $u_k = E(|\Delta_k|)$, the average number of remaining guesses after iteration $k$; is this attack practical?

Probing attack on DES

The probe records bit # $b$ of $L$

Select corresponding 6 input bits of $F$

Simulate with every possible 6 bits secret subkey

Eliminate guesses that do not match

In average, 6 different ciphertexts give 6 bits of $K_{16}$

Repeat on first round → 6 bits of $K_1$ (different)

Exhaustive search on 44 remaining bits

Algorithm 3 DES

1: $(L_0|R_0) \leftarrow \{IP(M)\}$  
2: for $i \leftarrow 1, 16$ do  
3:  
4: $L_i \leftarrow R_{i-1}$  
5: end for  
6: $C \leftarrow IP^{-1}(R_{16}|L_{16})$
Probing attack on DES

Exercise #7: How to optimize this attack?
Exercise #8: Is it practical on triple DES?

Homework on Probing Attack
- Have a look at at least one of these papers
  - Marcus Kuhn against DS5002FP
  - Cryptopage by Ronan Keryell
  - Or Google a bit on TrustZone by ARM
- Prepare questions

On-board probing

- Very convenient way to retrieve secrets
  - Easier than on-chip
  - Graduate student level, few $100
  - But not that easy...
    - High frequency busses
    - Metal tracks in inner PCB layers
    - Synchronization, storage, etc.
- Disk encryption keys from memory bus
- Game consoles reverse-engineering, hacking and piracy
- Active probing (injection) alter nominal behavior
  - Crypto-CPUs (DS5002FP)
  - Game consoles (XBox, ...)

The game consoles case

The Xbox 360 Security System and its Weaknesses

Housing removal

The first step: housing removal

Source: Playstation 3 Hacking – Linux Is Inevitable (http://www.pagetable.com/)

Source: Kingpin, @Stake
On board probing attacks

- Critical components may be encapsulated but IO remain accessible
- Even when deeply buried in multi-layer PCBs a track is much more vulnerable to probing than on-chip
- Wireless IO is even easier to monitor
- Test and other specific IO are at risk

On-board probing

- A lot of passive countermeasures
  - Make access to critical components more difficult
  - Remove identifiers from ICs
  - Use advanced packages (BGA)
  - Epoxy encapsulation
  - Remove test points
  - Obfuscate routing, route critical traces in inner layers, shorten traces
  - Increase frequency of busses

On-board probing: active countermeasures

- Passive countermeasures cannot fully protect IO
- Almost everything on board can be enciphered and integrity-checked in software... but the CPU memory bus
- Protect CPU memory bus against
  - On-board bus sniffing (confidentiality)
  - On-board bus injection (integrity)
- Enciphering and/or integrity checking of CPU memory bus
  - Bodyguard of the memory bus

The memory bus bodyguard
The on-chip memory bus bodyguard

On-board probing: active countermeasures

- Based on a hardware on-chip memory bus bodyguard
- Two different kinds of information to protect
  - Read-only (write once, read many), like code or read-only data
  - Read-write data
- Two different kinds of protection (four cases)
  - Confidentiality (yes / no)
  - Integrity (yes / no)
- Depends on the security objectives and targets
- Potential impact on cost / performances

On-board probing: active countermeasures

- Enciphering and/or integrity checking of CPU memory bus
  - Potential (huge) impact on cost / performances
  - Depends on the security objectives and targets
  - Some of this is used in XBox 360 (but not 100% efficiently)
- Autonomous bodyguard
  - Simple and efficient
  - Monolithic protection
  - Of whole or fixed parts of memory
  - Not flexible
  - Systematic performance degradation

On-board probing: active countermeasures

- Software-assisted bodyguard (Software Security Manager)
  - Flexible and efficient
    - Fine-grain (memory page) protection
    - Differentiated security policies
  - Minimize performance impact
  - Complex
  - Must avoid software exploits against SSM
    - Tiny and simple (fewer bugs)
    - Deeply verified, maybe formally proven (the must)
    - Integrated with virtualization hypervisor, micro-kernel, TrustZone® monitor, ...
**Code enciphering**

- Proposed by Robert M. Best in 1977 (patented)
- Frequently claimed security goals:
  - Protect embedded Industrial Property (IP): an attacker probing the bus between a CPU and its external memory cannot recover IP
  - Global system obfuscation: an attacker, whatever her goals are, cannot understand the system
  - Integrity: an attacker cannot induce her own code to gain privileged access to the system
- Principles:
  - Off-line enciphering (once), on the fly deciphering (many)
  - Simple on systems that do not update their embedded software
  - Software update still possible but a bit more complex
  - Note: read-only data can be handled the same way

**Programs are enciphered off line and stored in external memory**
- Bodyguard deciphers programs on-the-fly
- Instructions are cached in plain text form

**Code enciphering: performance**
- Deciphering cannot start before the read enciphered instruction is back from external memory
  - Overall latency = external memory latency plus deciphering latency
- How could we start the deciphering before the instruction is available?
  - Compute some seed from the secret key and the address issued by the CPU
  - Ultra-fast deciphering function that uses the seed ($\oplus$)
- If external memory latency \( \geq \) seed computation latency \( \Rightarrow \) no performance degradation even on cache misses
Code enciphering: performance

- Possible solution: (pseudo) one-time pad ciphers
  - Synchronous stream cipher
  - Block ciphers in counter mode, e.g. \( E_k(X, \theta) = DES_k(\theta) \oplus X \)
- Warning: one bit flip on \( E_k(X, \theta) \) flips the same bit on \( X \) (integrity!)
  ✓ Confidentiality and integrity are two completely different matters
  ✓ Oblfuscation does not help integrity

Code and data enciphering

- Same frequently claimed security goals:
  - Protect embedded software Industrial Property (IP): an attacker probing the bus between a CPU and its external memory cannot recover IP
  - Plus: an attacker cannot get information from the data observation
  - Global system obfuscation: an attacker, whatever her goals are, cannot understand the system
  - Integrity: an attacker cannot induce her own code to gain privileged access to the system
  - Plus: an attacker cannot induce her own data to gain privileged access to the system

Markus Kuhn against DS5002FP

- DS5002FP
  - 8 bits «secured» microcontroller
  - Enciphered external memory (8 bits data and 17 bits addresses)
  - One 64 bits key, cipher depends on memory address
  - Dummy accesses when memory bus unused
  - Instruction fetches randomly swapped with preceding dummy access
  - No integrity check
  - 10 years life lithium battery (for on-chip key memory and off-chip SRAM)
Markus Kuhn against DS5002FP

Markus Kuhn
  • Brilliant mind, curious and motivated, master student

Memory bus passive and active probing
Injections with switch between SRAM and custom FIFO
Dictionary attack: even if key unknown, 8 bits block ciphers are just 256 ↔ 256 bijections
First step: exhaustive search of first three bytes (X, Y, Z) such that
\[ D_k(X, Y, Z) \equiv \text{MOV PP1 T} \] (write T to parallel port #1)
  • This will completely disclose \( E_0(T) = Z \), the enciphering function at address of Z (try all Z (256) and recover dictionary)

Markus tries all \((X, Y)\) such that \( Z \mapsto T \) is a bijection
  • About \( 2^{17} \) reset, 300 reset/s ⇒ 7 minutes
  • \( E_0 \) is fully known (but not the key)
  • Markus can now inject anything at Z’s address

Next steps:
  • Search \( X = \text{NOP} \)
  • Move ahead until enough \( E_i \) are known
  • Inject a complete «dump out» program

It took a few hours and a US$300 equipment to a master student
✓ Dictionary attacks are serious threats
✓ Integrity cannot be guaranteed by enciphered busses

Code and data enciphering

Actual security properties vs. claimed security goals
  • Protect embedded software IP: yes
    – As long as the secret key is protected
    – As long as a dictionary attack remains infeasible
  • Global system obfuscation: yes
    – As long as the secret key is protected
    – As long as a dictionary attack remains infeasible
  • Integrity: no
    – Any data or instruction can be replaced by anything, anytime

Performance
  • No penalty until cache miss
  • Cache misses have their cost increased by the deciphering latency
  • Embedded enciphering-deciphering function ⇒ silicon area and power

Performance enhancements

The same enhancement (one-time pad) holds for code and Read-Only (RO) data
But not for Read-Write (RW) data!
? Why?
  • On read-write locations «on-time» pad is many-times!
  • \( E_k(X, \emptyset) \oplus E_k(Y, \emptyset) = X \oplus DES_k(\emptyset) \oplus Y = X \oplus Y \)

RW-data confidentiality cannot be protected by one-time pad
Regular symmetric block ciphers are needed
RW-(en)deciphering latency cannot overlap memory latency
Performance degradation on data cache misses
Can be huge...
… or not: hundreds of clock cycles memory latencies are common
Some possible security enhancements

- Secret key cannot be accessed by software
- Multiple keys, maybe encrypted by a master key (public key cryptography?)
- Every CPU has a unique master key
- A more general master key per chip manufacturer could be needed
- Different keys for code and data
- Data keys change randomly at reset
- Illegal instructions trapped and used as intrusion detection
  - Beware the bogus software
- Anyway, integrity would really be a must!

Attacks against integrity

- Spoofing attack: space permutation
- Replay attack: temporal permutation
- Splicing attack: injection of forged data

Why check memory integrity?

- Because strong enciphering of data or instructions does not protect against spoofing, splicing or replay...
- An attacker could:
  - Inject forged data
  - Permute two memory locations
  - Replay an old data at same address
    - Prevent a memory write and induce an infinite loop:
      ```
      for(i = 0; i < size; i++)
        print(*p++);
      ```

- A lot of integrity attacks are possible and dangerous
- They aim at modifying the behaviour of the system (fault attacks)
- The system needs a way to detect any malevolent modification

Cryptographic hash functions

- Cryptographic hash functions are designed to check integrity
- Hash function $H$ produces constant length digest $h = H(M)$ from variable length message $M$
- Impossible to forge message with given digest (pre-image)
- Impossible to find two message with same digest (collision)
- Hash functions are also named one-way functions
- MD5, SHA-0, SHA-1, SHA-2, SHA-3
### How to check memory integrity?

- Group data in constant length chunks
- Store chunks' digests somewhere in memory
- Upon read access
  - Read whole chunk $C$
  - Compute chunk's digest $h = H(C)$
  - Read digest $\hat{h}$ from memory
  - Compare digests
    - $h = \hat{h} \Rightarrow OK$
    - $h \neq \hat{h} \Rightarrow not OK$
- Upon write access
  - Read whole chunk $C$
  - Compute chunk's digest before $h = H(C)$ and after write $h' = H(C')$
  - Read digest $\hat{h}$ from memory
  - Compare digests
    - $h = \hat{h} \Rightarrow OK$, perform write, update $\hat{h} \leftarrow h'$
    - $h \neq \hat{h} \Rightarrow not OK$
- But attacker can replace any chunk (spoofing) and its digest $\hat{h}$!
- MACs stored in external memory cannot detect splicing
  - Attacker can swap data chunks $C_1, C_2$ and their digests $h_1, h_2$ without being detected
- Use keyed hash functions or Message Authentication Codes (MACs)
  - A bit like hash functions
  - But computation requires a secret key
  - e.g. last block of block cipher in CBC-MAC
  - Attacker cannot compute digest $h$ of forged chunk
  - Spoofing detected
  - Does it detect splicing?
- MACs stored in external memory cannot detect replay
  - Attacker can swap data chunks $C_1, C_2$ and their digests $h_1, h_2$ without being detected
  - Blend chunk's address in digest computation
  - Use CBC-MACs with chunk's address as first block
  - Splicing detected
  - Does it detect replay?
How to check memory integrity?

- MACs stored in external memory cannot detect replay, even with address involved
  ✓ Upon every memory access CPU computes digest of entire memory, compares with internal reference and updates reference (if write access)
    • Computationally infeasible
    • Computing digest implies reading entire memory
    • Huge, unacceptable, performance degradation
  ✓ Use Merkle hash trees to reduce overhead

Merkle hash trees

- Used as integrity checkers in secured file systems or peer-to-peer networks

Verified read and write

1: \( a_{i,j} \leftarrow \text{Read}(i,j) \)
2: \( a \leftarrow a_{i,j} \)
3: while \( i > 0 \) do
4: \( f \leftarrow a_{i,0} \)
5: \( p \leftarrow \lfloor f/2 \rfloor \)
6: \( a_{i,f} \leftarrow \text{Read}(i,f) \)
7: \( a_{i-1,p} \leftarrow \text{Read}(i-1,p) \)
8: \( h \leftarrow H(a_{i,min(j)}, a_{i,max(j)}) \)
9: if \( a_{i-1,p} \neq h \) then
10: error
11: end if
12: \( j \leftarrow j - 1 \)
13: end while
14: return \( a \leftarrow \text{VerifiedRead}(i,j) \)

Cost of replay detection

- Hypothesis: Merkle nodes are 128 bits long, Merkle trees are binary
- Cost of integrity
  - Memory to store Merkle trees: double the size of secured memory
  - Read latency: \( \log_2(n) \) times the standard read latency
  - Write latency: \( \log_2(n)^2 \) times the standard write latency (can be reduced to \( \log_2(n) \) if recursion is not used)
- Example:
  - 1 Mbytes of secured memory, 16 bytes (128 bits) nodes
  - \( 2^{16} \) leaves \( \Rightarrow n = 16 \)
  - Total memory size: 2 Mbytes \( (2^{16} + 2^{15} + 2^{14} + \ldots + 2^1 + 1) \)
  - One verified read = 17 reads
  - One verified write = 17 writes
Cost of replay detection

- Caches storing upper part of Merkle trees can speed up reads and writes
- Example:
  - 1 Mbytes of secured memory, 16 bytes (128 bits) nodes
  - \(2^{16}\) leaves \(\Rightarrow n = 16\)
  - 8 first stages of Merkle trees stored in internal cache
  - Cache size: 28 nodes \(= 256 \times 16 = 4\) k-bytes
  - Total memory size: \(= 2\) Mbytes
  - One verified read \(= 9\) external reads, 8 internal reads
  - One verified write \(= 9\) external writes, 8 internal writes
- More on memory bodyguard in appendix

Science fiction: a true TCP

- What follows is pure fantasy
- The future’s not ours to see
- Technical innovations could completely cancel current threats
  - 3D integration
  - On-chip extra-large memories
  - Increasing gap between sampling rates and busses frequencies
- Technical innovations could open new threats
- Anyway, these are my own thoughts, feel free to disagree

Must have of a true TCP

- Assume on-chip is secure
  - SoC internals are “naturally” protected (too expensive)
  - If your adversary is that powerful,…
  - Take however appropriate countermeasures against side channel and fault attacks
- Hardware protections
  - Communication with external world
  - On-board bus sniffing (confidentiality)
  - On-board bus injection (integrity)
  - Most can be protected in software but… only if memory is trusted…
  - ✓ On-chip hardware bodyguard between CPU and memory controller?
- Bodyguard does not protect against software exploits
  ✓ Strong software kernel
    - ✓ Tiny and simple (fewer bugs)
    - ✓ Deeply verified, maybe formally proven (the must)
    - ✓ In charge of critical tasks
  ✓ Virtualization hypervisor, micro-kernel, TrustZone® monitor,…
  ✓ If software involved in bodyguard management \(\Rightarrow\) Software Security Manager (SSM)
  ✓ Both protected (integrity and confidentiality of sensitive data) by bodyguard
Must have of a true TCP

- Boot ROM, internal RAM
- eFuses
- Random Number Generator
- Boot to boot integrity (including downgrade replay attack) is an issue
  - ✓ On-chip non-volatile memory
  - ✓ Larger boot ROM and a trusted third party (smart card, network)

Conclusion

- A true TCP would have hundreds of applications
  - Digital Rights Management (but is this just feasible?)
  - Trustable cloud computing (not only storage)
  - Trustable remote third party (smart card-like PC)
- Technical solutions are not really mature
  - Software flaws
  - Bus enciphering is not sufficient
  - Limited bus integrity is not scalable and/or risky
  - Full bus integrity is expensive

Conclusion

- The performance and cost impact are critical
- Security, like speed, is a matter of tradeoffs
  - Inside ⇒ fast and secure
  - Outside ⇒ slow and insecure
  - Inside security is cheap, outside security is expensive
- TCP will probably rely on several layers, from small and strongly secured to large insecure
- There are also many philosophical, legal and social issues

Thank you for your attention

- And remember please Auguste Kerckhoff's principles (1883)
  - The system must be substantially, if not mathematically, undecipherable;
  - The system must not require secrecy and can be stolen by the enemy without causing trouble;
  - It must be easy to communicate and remember the keys without requiring written notes, it must also be easy to change or modify the keys with different participants;
  - The system must be portable, and its use must not require more than one person;
  - Finally, regarding the circumstances in which such system is applied, it must be easy to use and must neither require stress of mind nor the knowledge of a long series of rules.
Summary

- HWSec: the course. Questions?
- Probing attacks. Questions?

Further Reading

- Stefan Mangard and Elisabeth Oswald and Thomas Popp
  Power Analysis Attacks: Revealing the Secrets of Smart Cards
- Wolfgang Rankl and Wolfgang Effing
  Smart Card Handbook
- Nadia Nedjah and Luiza de Macedo Mourelle
  Embedded Cryptographic Hardware: Design & Security
  And a lot more in the library...

Bodyguard: role

- Filter memory accesses
- Enforce security policies
- Encipher and decipher
- Check integrity (spoofing, splicing and replay)
- Protect memory bus and only that (inter-process isolation, software exploits are totally different issues)
- Details depend on target market constraints
  - Store digests in internal memory ⇒ limited amount of integrity-protected external memory (XBox 360)
  - Merkle hash trees ⇒ scalable
  - Software controlled («a la MMU») ⇒ flexible
  - Hard-wired security policies ⇒ simpler (XBox 360)
  - Embedded caches in bodyguard improve performance

Bodyguard: constraints

- No CPU modifications (acceptability very low in many cases)
  - Sees only physical addresses
  - Has no knowledge of process IDs, privilege levels,...
  - If such knowledge needed, must be provided by software
**Bodyguard: performance**

- **Confidentiality**
  - Encipher and decipher on write and read
  - Increases memory latency
  - Stream ciphers as one-time pad on read-only data
    - Block cipher in counter (address) mode
    - Simple
    - One-time pad computation hidden by read latency
  - Block cipher in CBC mode on RW data
  - ? But how does bodyguard distinguish RO and RW?

- **Integrity**
  - Complex
  - Slow
  - Memory hungry
  - Spoofing: Message Authentication Codes (MAC)
  - Splicing: MACs with addresses
  - Replay: the most difficult
    - Internal digests memory: non scalable
    - Merkle hash trees: expensive
    - RO data not sensitive to replay
    - ? But how does bodyguard distinguish RO and RW?

**Bodyguard: important features**

- Offer security-performance trade-offs
- Run existing applications unmodified
- Support security-aware applications
  - Software managed bodyguard (SSM)
  - Unaware applications
    - Default security policy applied by OS (e.g. on a linker sections basis)
    - Security policy specified at launch time
  - Security-aware applications
    - Come with pre-defined security policies (e.g. on a linker sections basis)
    - Use the SSM Application Programming Interface (API) to dynamically specify security policy of newly allocated memory pages
    - Can dynamically adapt their security policies