Data collection and pre-processing for white-box side-channel analysis

Michaël ADJEDJ & Sylvain LEVEQUE

13/12/2018

Binary tracing

- Equivalent of an oscilloscope
  - Information collection
- White-box context
  - Perfect capture (no noise, leakage model = value)
  - Reproducible
- Debugger, emulator, instrumentation (DBIs), hardware (Intel PT)

An example: TracerGrind

- Publicly available, designed for WB analysis
  - Based on well-known Valgrind DBI
  - Focus on memory accesses
  - https://github.com/SideChannelMarvels/Tracer
- Example binary: CHES 2017 – Challenge 84
- Trace size: **1 AES ~27GB**

TracerGrind limitation

- As said, *focus on memory accesses*
- What about this situation?

Pseudo code	Memory	Registers
tmp = mask1_xor_val	⇦ mask1_xor_val	mask1_xor_val
tmp ^= mask1	⇦ mask1	val, mask1
tmp ^= mask2	⇦ mask2, ⇨ mask2_xor_val	mask2_xor_val, mask2

Working with registers

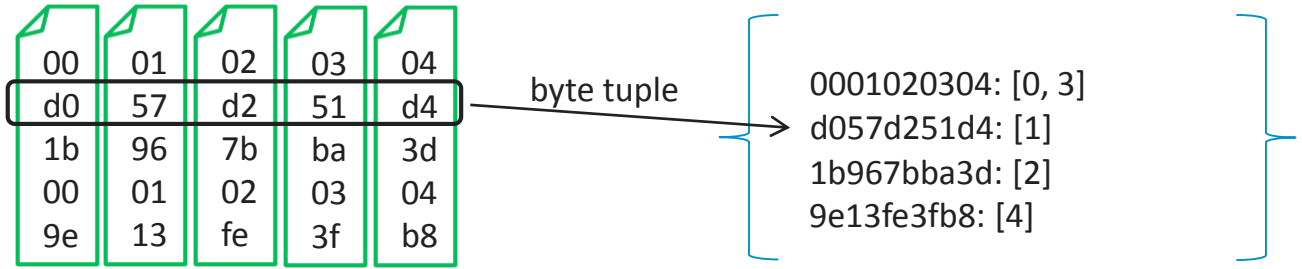
- Switch to *qemu*-based tracing to collect registers
  - `-seed 0`: stick PRNG sequence (ASLR …)
  - `-singlestep`: dump state at each instruction rather than basic block

- Many registers, many instructions ⇨ huge amount of data
  - Eg AES #84 generates > 120GB data ☹
  - And this is without obfuscation…

- Redundancy? Evaluate with compression
  - 120GB ⇨ 5GB, divided by 24! (27GB ⇨ 1.5GB for TracerGrind)

Reducing trace size: some tricks

- Deduplication
  - Noiseless data collection ⇨ remove redundant information
  - At the byte and bit levels



Reducing traces size: some more tricks

- Selective dump
  - At each instruction, only collect potential changes
  - Desynchronization is an issue

- ”*Toward Fully Automated Analysis of Whiteboxes – Perfect Dimensionality Reduction for Perfect Leakage*”, Breunesse et al. https://eprint.iacr.org/2018/095.pdf

Deduplicating efficiently

- Objectives
  - fast with lowest programming effort possible
  - Take advantage of HULK*
- C++11 with useful features:
  - Use `std::for_each`, `std::sort`
  - Lambda functions

- Thrust
  - Replace `std::vector` by `thrust::device_vector` ⇨ automatically run on GPUs!
  - Replace `std::for_each` by `thrust::for_each` ⇨ automatically parallelized!
    - Compile using Threading Building Blocks backend (Intel)
    - Combine with `thrust::device_vector` to run on GPUs

*HULK: 48 cores, 192GB RAM, 3TB SSD RAID, 2 GTX 1080Ti

Fun fact

- Function inlining + obfuscation: better perf than not inlined!
- Unexpected. Why?
  - Guess: obf impact smoothed when inlined, « hit-or-miss » when not
  - Security impact?

Thank you!

gemalto