Side-Channel Notes

note: this document only contains the main points and eferences to good/usable sections for later use.

1 Contr-Channel Attacks (Paging15)

- virtually no noise, only 1 run required
- bypass 2 shielding systems (like Haven (uses SGX) and InkTag)
- Attacking Hunspell and libjpeg decompression

Challenges

- granularity of 4KB pages
- interrupts/handlers/switching in/out of enclave → large performance overhead
- does not work for off the shelf -O3 etc. optimized code

Requirements

- Memory management by OS
- Application code must be known, and not hardened

Idea

- offline analysis, outside the secure environment
- OS inserts page fault traps to observe input dependent or data dependent accesses
- we record page-fault sequences
- Input dependent accesses:

two addresses are associated with a page fault: The address which triggered the page fault and the instruction that was executed when it happened. When they are equal it's a code page fault, o/w it's a data page fault.

- Only track a small set of relevant pages
- To bypass ASLR we record the very early page faults of an executable, this allows finding the base address of the loader.

Control transfers

- get a couple of page fault traces P_i, and also convert it to page base address trace Q_i.
- For code page faults aka control transfers: identify the target address f. For each occurrence of f in any P_i, search the shortest sequence of preceding pages for which the corresponding sequence in Q_i leads only to f.

Data accesses

- Record all accesses outside the shielding system
- Identify all data page faults
- $\forall$ data page faults we search for a minimum sequence of code page faults right before, that can be used to uniquely infer the data access.

Implementation

- The custom page-fault handler is installed by overwriting the IDT interrupt descriptor table.
- For InkTag page-fault handler was inserted in the Linux page-handler

2 SGX Cache Attacks r practical (17)

- Nice short background chapters on cache, SGX
- $\exists$ extended version of this paper! https://arxiv.org/pdf/1702.07521.pdf
- SGX enclaves can disable PMCs by activating a feature called ASCI (Anti Side Channel Interference)

Goal

- Untrusted OS
- Prime & probe attacks on RSA decryption and genomic processing
- OS ensures victim enclave and attack program run on same core → uninerrupted → hard to detect by enclave, do interrupts and PMCs on other process.
- Generate cache access trace by precise L1 monitoring using CPU performance counters (PMC), learn access pattern step by step +1 data access at a time

Limitations

- Does not work against hardened code
- "We compiled enclave RSA decryption code with default optimization flags and compiler settings" =?
- does the genome sliding window really only move 1 chracter at a time? Doesn't make sense to me.

Thread model

- Attacker knows mapping of memory addresses to cache lines and can re-init resp replay the enclave and its inputs.
- Attacker controls allocation of resources & core of enclave
- Attacker can modify interrupt handler, frequency of timers etc.
- He can **not** see enclave memory content or registers

Noise Reduction Techniques For not having to run the attacks thousands of times

- Isolated core by modifying the linux scheduler
- Self-pollution exploit the fact that L1D and L1I are separate → less noise
- Uninterrupted execution: avoid AEX (async enclave exit) and the OS's ISR (interrupt service routine). That means configuring the interrupt controller to not deliver interrupts to the core the attack core. The only per-core interrupt is the timer interrupt. So they lowered its frequency.
- Measurement: apparently *reading the TSC* is itself in the order of L1-L2 access time difference → too noisy, use PMC
  $\exists$ RDTSC and RDPMC !!

RSA Attack

- Non-hardened version of RSA decryption found in Intel IIP crypto library
- fixed-size sliding window exponentiation using a precomputed multiplier table
- The algorithm iterates over all exponent windows and, depending on the window value, it may perform a multiplication with a value from the table
- monitor a single multiplier access at a time
- each multiplier in the table is 1024 bits, accessing the multiplier causes 16 repeated memory accesses
- repeat this process for a subset of all possible multipliers (10 out of 16 in our case), because extracting 70% of a key is enough. Additional advantage because some cache sets had more noise than others!

Genomic Attack STR (Short Tandem Repeats) analysis, looks at the lengths of some sequence at specific places. In its preprocessing the genome is divided into k-mer substrings, and then sliding window (one char at a time) over it. PRIMEX (open source analysis tool) inserts those k-mers into a hashtable.

- Goal is to find the length of a e.g. TAGA sequence
- Monitor the 4 cache lines of TAGA, AGAT, GATA, ATAG
- only when all 4 get utilized more than normal we're seeing a TAGA sequence
- can determine length ± 1. Doing that for 3 such sequences as STR does, it's enough to pinpoint individuals.

Defenses n1 summary of defense papers

- disable cache
- architectural/hardware changes
- obfuscation (e.g. ORAM = continuous shuffling of data, add noise, scrubbing, flushing shared cache)
- hardening (safe libraries, scatter-gather, auto compiler)
- Randomization (SGX Shield, but only for code not for data)
- Attack detection (using performance counters, page faults, transactional memory, restart notification, time delay detection) but all hard with a powerful OS adversary

3 Cache Attacks on Intel SGX (17)

- AES-NI (hardware AES) is better against side channels
- This paper gives a good idea of various obsticles that one will encounter when performing a real side channel attack. (cores, threads, context switch, flushes, ECALL, evictions, noise, hyperthreading)

Idea

- Attack against old vulnerable AES algorithm in SGX
- Prime and probe in set-associative cache, using PMC

Limitations

- We cannot really run OpenSSL and pass ciphers/packets in and out of the enclave, as the context switching causes too much noise in L1. Therefore shared memory was set up between the enclave thread and the attack thread. Then they put text there as well as a flag that tells the enclave to start encrypting. In other words, "active waiting to avoid noisy syscalls". (ECALL triggers AEX). It's programmed to wait before the last round of AES starts etc.
- 2 processes won't work, because the context switch clears L1 as well (because L1 operates on virtual addresses, which need

to be flushed). Therefore they used 2 threads in one process. But this should be achievable by the strong attacker model.
- Two threads is still problematic. Though it's not a full ECALL AEX, the CPU still has to enter/leave the enclave which causes too many L1 evictions. Solution: Intel Hyperthreading (two logical CPUs per core → enclave exits are omitted)

4 Branch Shadowing (17)

- This aims to be better than the paging attack (which only leaks page address and not byte address)
- good background on enclave EENTER, EEXIT, ERESUME
- good background on branch prediction & BTB & LBR
- Works better than controlled channel (paging 15) on RSA, it's more fine-grained.
- has some code for SGX driver / APIC timer modification

Idea
- SGX does not clear branch history when switching from enclave mode to non-enclave mode
- use last branch record (LBR) based history inferring and an advalced programmable interrupt controller (APIC) based technique for fine grained execution history (resp branching history). Much more accurate than RDTSC.
- Enclave and OS share branch target buffer (BTB). The attacker can introduce set conflicts by positioning a branch that maps to the same spot in the BTB as the target branch inside the enclave. Then he can probe and measure the time.
- Program the APIC to interrupt as frequently as possible, and make it append the LBR content to a branch history.

Limitations
- know sourcecode
- needs to know virtual addresses, so attacker has to map enclave to specific memory region (ok, since compromised OS can easily manipulate the enclaves page table entries, and he can disable user-space ASLR, and modify the sgx driver to change the base address)
- attacker needs to know when exactly the moment is to start interfering. → needs some signs like page faults or syscalls.
- must prevent enclave from having a reliable time/other-source to avoid detection of slow down
- Enclave must run in debug mode, o/w it does not report to the LBR
- By manipulating the local APIC timer, the max interrupt frequency was about 50 cycles (too long for very short loops). But by selectively disabling L1 and L2 cache, they can get it down to ~5 cycles.
- it cannot distinguish a not-taken conditional branch from a not-executed conditional branch because, in both cases, the BTB stores no information
- it cannot distinguish an indirect branch to the next instruction from a not-executed indirect branch because their predicted branch targets are the same.
- it needs like 10+ repetitions, thus persistant storage / slowdown detection mechanisms could prevent it.

4.0.1 Conditional Branch Shadowing

Inferring through RDTSC
- create malicious program that has a BTB-aligned branch. Make it such that it is always (static) mispredicted.

```
if (c != c){
   // static will wrongly predict here
} else { }
```

- When the victim code is executed before the shadow code is executed, the branch (mis-)prediction of the shadow code depends on the execution of the victim code. So if the victim takes the branch, the shadow code will also do (so the always-mispredict does not happen, nice)
- This approach takes about 2x more runs compared to using LBR to get accurate results

Inferring through Intel PT
- Intel Processor Trace provides timestamp packets that contain precise elapsed cycles counts (CYC packets).
- Problem is that it aggregates multiple branches into one packet. But still more accurate than RDTSC.

Inferring through LBR
- Similar to RDTSC inferring, but instead of timing we do: run enclave brranch, interrupt, enable LBR, run shadow code, read out LBR, disable LBR, (if we're in a loop, "reset" the BPU by doing some interleaved takes/not takes).
- Use LBR's filtering to get rid of function calls, returns and other noise jumps.

4.0.2 Unconditional Branch Shadowing

- Unconditional branches are always taken (e.g. end of `if`-stmt always jumps over `else`-part)
- BTB still stores the unconditional branch though (Addr, target addr, take y/n). By having a branch at the same addr in the shadow code, but with a different target addr, we'll induce a misprediction.
- LBR falsely reports such mispredictions as good. Therefore the (mis)prediction is measured using the LBR cycle count. To time it out we need another small branch.

4.0.3 Indirect Branch Shadowing

- Indirect means the target comes from e.g. a register. If not taken it just goes to the next instruction
- To detect this our shadow code has a 1 instruction jump. If the enclave jumps we'll see a mispredict, o/w a predict. Measure by reading LBR.

Countermeasures
- Flushing Branch State when EEXIT/EENTER etc. (negligible overhead under reasonable circumstances) However, it's kind of a hardware update and for existing CPU's it would need to be added using micro code (unsure if possible). However hyperthreading would be a way around this, because there no real exits happen when switching.
- Obfuscating Branch (remove branches, ORAM Raccoon, using `cmov`). But these are algorithm specific to some degree. Raccoon has like 20x overhead.
- They propose:

ZigZagger uses `cmov` and introduces a sequence of non-conditional jumps instead of each branch. It's just a few jumps in a row that are executed almost simultaneously such that recognizing the current instruction pointer is difficult.

5 SGX-Step (17)

- enclaved execution can be precisely single-stepped using a novel APIC timer manipulation
- We implement SGX-Step as an open-source 1 Linux kernel driver and runtime library

Idea
- Somehow more fine graded programmability of the APIC. And better timer interval prediction
- When an enclave receives an interrupt, it performs an Asynchronous Enclave Exit (AEX) and then jumps to the handler defined in the interrupt descriptor table (IDT) to take care of the interrupt. After the interrupt has been handled, it jumps to the address set in the asynchronous enclave pointer (AEP). The function in the AEP eventually executes the ERESUME instruction to resume the enclave.
- SGX-Step installs a custom interrupt handler in user-space to gain control as soon as possible after the interrupt. It also replaces the AEP to execute custom instructions right before ERESUME. SGX- Step uses these modified routines to store the current cycle count just before entering the enclave and right after an AEX.

Limitations
- CPU specific

6 Frontal Attack (20)

- we observe that in modern Intel CPUs, some instruction's execution times will depend on which oper- ations precede and succeed them, and on their virtual addresses.
- Frontal attack allows the adversary to distinguish between two execution sequences even if they contain identical instructions (and even identical data).

- n1 Introduction cloud → TEEs → SGX → paging → Intel: "devs in charge" → BPU → Steping

Limitations

- This is the paper Kari said: They discovered the side channel but cannot explain it!
- Need writes that have a certain alignment
- Works better on some processors than others
- Extensive offline preparation and measurements beforehand

Idea

- When resuming after an interrupt, the CPU fetches the code block that includes the next instruction. Depending on where it is in the codeblock, the previous instructions need to be discarded by the frontend
- especially for `mov`'s there seem to be two groups that are about 100 cycles apart.
- it happens with memory (write) instructions
- also the density of surrounding memory instructions seems to be a factor

Defense

- Branches should be aligned
- Or hardware fix

7 Raccoon (15)

- method of defending against a broad class of side-channel attacks, which we refer to as *digital side-channel* (side channels that carry information over discrete bits) attacks. Instead of point defense
- input is program (source code level), output obfuscated binary that provides the illusion that many extraneous program paths are executed.

Ideas

- All digital side channels emerge from variations in program execution
- On the real paths, each store operation first reads the old value of a memory location before writing the new value, while the decoy paths read the old value and write the same old value.
- Decoy and real paths will both write (different) values, but unless an adversary can break the data encryption, she cannot distinguish decoy from real paths by monitoring digital side-channels.
- Upside: Source code can be public, only data needs to be secret. Works on existing machines

Limitations

- Non-digital side channels such as e.g. power are not defended. Because decoy path might do a (encrypted) increment while real path does something else.
- Assume OS is trusted. Input program must be free of errors. Raccoon does not obfuscate I/O yet. Cannot obfuscate libraries whose source code we don't have. Raccoon does not prevent leaks via loop iteration count. Does not obfuscate terminal branches of recursive calls.

Design

1. taint analysis to identify statements that need to be obfuscated.
   user annotates secret variables.
2. runtime transaction-like memory mechanism for buffering intermediate results along decoy paths.
   Raccoon buffers loads and stores along each path. and do them execute batch(branch) wise? or all together?decoys read an existing value and write it back.
3. program transformation that obfuscates control-flow statements.
   First, perturbing the branch outcome by executing all branches sequentially (`obfuscate()`). To go back to the start from an e.g. if-clause they implemented an `epilog()` function. Finally need to ensure decoy does not change anything relevant, done with oblivious store operation (which leverages `cmov`).
4. code transformation that uses "Path ORAM" to hide array accesses that depend on secrets.
   the algorithm conceals data accesses by continuous shuffling and re-encrypting data when accessed.