Miscellaneous Thingies

Berkeley EECS
ELECTRICAL ENGINEERING & COMPUTER SCIENCES

Three Miscellaneous Things:
Not on the final

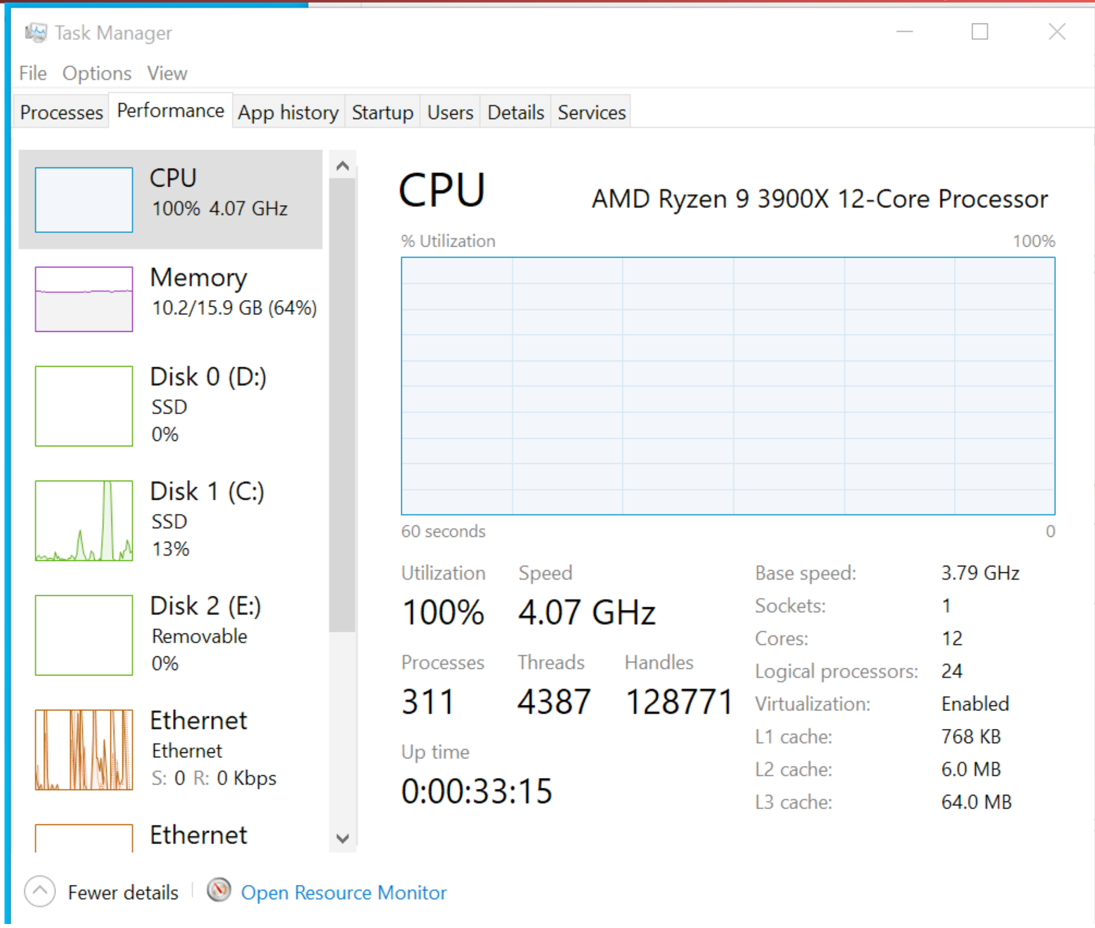
- GPUs (Guest from Apple)
- Building a software pipeline in a 61C style
  - Or "How Nick pegged all cores on The Beast"
- Two cool ISA items
  - ARM Pointer Authentication Codes
  - RISC-V 16b instruction encoding

Guest Lecture Here...
But Why?

- GPUs are huge SIMD parallel devices
  - "For every element do X" operations
- Really good for matrix multiplies
  - Which is what a huge fraction of "Machine Learning" really is
  - And how you render scenes graphically

Building A Data Pipeline...
61C Style on my Linux box

- Some bad actor stole >4 GB of data from UCOP
  - Basically everything that was on the "secure" file transfer server in December
- The bad actor released at least some of this data ***publicly***
  - As a 4GB compressed archive that anyone can download
- What information about ***me*** was in the archive?



CPU AMD Ryzen 9 3900X 12-Core Processor

Utilization	Speed	
100%	4.07 GHz	
Processes	Threads	Handles
311	4387	128771
Up time		
0:00:33:15		

Base speed:	3.79 GHz
Sockets:	1
Cores:	12
Logical processors:	24
Virtualization:	Enabled
L1 cache:	768 KB
L2 cache:	6.0 MB
L3 cache:	64.0 MB

What I need to know...

- I already know my social security # got breached
  - They told us that...
  - But I've got fraud alerts & freezes in place already
- But what other information?
  - Address? Phone #? Things I don't know about?
  - Tax information?
  - ***Banking information?***
    - The numbers on the bottom of a check are all an attacker needs to make fake checks

The Nature of the Dump...

- A ***lot*** of pdf files
  - PDFs are a pain to search, need to convert to text
- A ***lot*** of data tables
  - Some as comma-delimited text, some as excel spreadsheets, some in stada format
- Need to convert it to something reasonable
- Google around...
  - Nice linux OCR pipeline cobbled together:
    PDF -> images -> OCR text
  - pandas can read both xlsx and stada files

Step 1:
File Conversion

- Want to convert everything into text files
- Obscenely parallel problem:
  - For every PDF do X
- But with some gotchas...
  - I can't just spawn 700 PDF->txt conversion programs
    - That would grind my machine to a halt by exceeding my available memory
  - And different invocations take a different amount of time
- Two approaches
  - Dynamically tune based on load...
    - When load < 95%, spawn another job and wait 1 second before checking again
    - When job ends and load < 99%, spawn another job
    - When job ends and load >= 99%, don't spawn another job
  - Or just say "F-it, and keep X jobs live"

Keeping X jobs live: Fork/join with a limiter

- Used a simple golang hack
  - Go == C with safety, garbage collection, and really good parallelism
  - Channels are like Queues in python:
    You can add elements, and trying to read blocks until an element is available

-
  ```
  capacity := make(chan bool, 10)
  done := make(chan bool)
  ```

-
  ```
  func run(txt string) {
     c <- True; // Will block until capacity is available
     ....
     <- c; // will release capacity
     done <- True; // join on fork/join
  }
  ```
- main just calls "**`go run()`**" on every line of stdin...
  and then an equal number of lines of "**`<- done`**"

A bit of tuning...

- ~10 jobs pegs all CPU cores on the OCR pipeline...
  - And it took a couple hours to PDF->txt the lot:
    Driver used multiple threads for single documents
- The .xlsx and stata conversion was a lot faster...
  - Set to ~25 jobs instead of 10 (since python doesn't thread, especially on this task)
  - Took ~10 minutes or so
- Not fully efficient...
  - At the end of the PDF run there was no longer pegged CPU
  - 100% CPU utilization means efficiency loss due to context switching:
    Optimum would be ~95%
  - Also, really stressing the Windows virtualization...
    - I do all my work in "linux" under WSL

And Now To Search

- Just use the same pipeline with grep...
- But...
  - Ends up ***not*** pegging the CPUs...
    Instead I'm pegging the "disk"!
- OK for just searching for me, but...
  - Want to be able to do a "for anyone who wants" service
- So to do this, parallelize on an alternate axis:
  - Don't check one person at a time, check ***all*** people using a single program
  - And then invoke that in parallel across all files
- Gotcha problem: Need to make sure to synchronize writes well
  - Again, golang FTW:
    A channel for each user's results, the search does an atomic write to the channel

Memory Hardening... ARM Pointer Authentication Codes

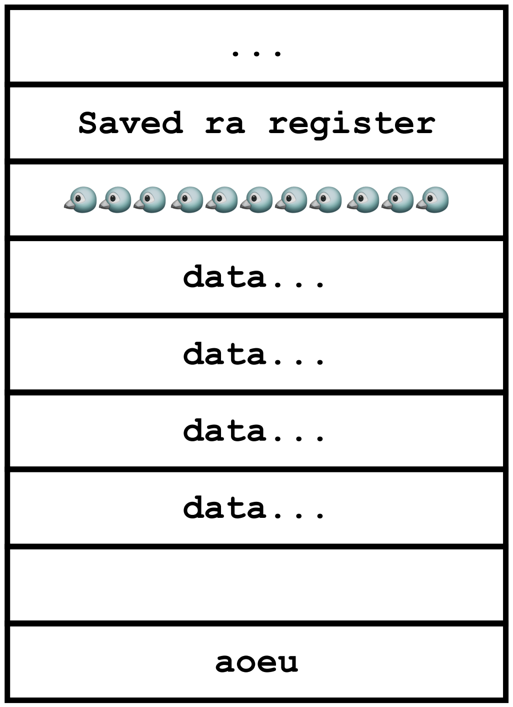
- Attackers want to overwrite memory...
  - When your C code fails to check a buffer
- The classic vulnerability

```
void foo(){
    char c[32];
    gets(c)
}
```

- Attacker gives you too long an input...
  - And c is stored on the stack
  - So the attacker overwrites not just c but the other stuff on the stack...
    - Such as the saved **`ra`**
  - The saved **`ra`** is overwritten to point to the attacker's code in memory

Stack Canaries…

- Goal is to protect the return pointer from being overwritten by a stack buffer…
- When the program starts up, create a ***random*** value
  - The "stack canary"
- When starting a function, write it just below the saved frame pointer or ra register
- When returning in a function
  - First check the canary against the stored value



Stack Canary Overhead...

- May require enabling an optional compiler flag...
  - So of course it is commonly not done!
- Requires a memory load & store on every function entrance
  - Highly cacheable so basically only 4 instructions on a typical RISC:
    Load address of canary (2 instructions)
    Load canary value into register
    Store canary value onto stack
- Requires 2 memory loads and a (probably) not taken branch on exit
  - So 5 instructions on a typical RISC:
    Load address
    Load canary value
    Load canary off stack
    BNE (mark as probably-not-taken if you can)

So example code...

-
  ```
  la t0 canary # Reminder, turns into two
               # instructions
  lw t0 0(t0)
  sw t0 x(sp)  # four below where ra got stored
               # if we don't bother saving the frame pointer
               # if we do save the frame pointer, 4 below fp
  ```

-
  ```
  la t0 canary
  lw t0 0(t0)
  lw t1 x(sp)
  bne t0 t1 dead_canary
               # Make sure this is a forward branch:
               # So CPU assumes it won't be taken
  ```
- Note also generally sequential:
  only parallelism present is in loading the canary from both the stack and storage

Brute Force...

- Brute force: just simply try every possibility
  - Or if its a different random # each time, just always try the same number
- Even the smallest timeout goes along way:
  - If you can try 10,000 per second, trying 2^{20} possibilities takes less than 2 minutes
  - If you can only try 10 per second, it takes a day and a half
  - And if 10 failures causes a 10 minute timeout...
    Forgettaboutit!
- Exponentials matter
  - If it take 1 minute to try 2^{20}, it will take 16 hours to try 2^{30}
  - And 2 years to try 2^{40}!
  - EG, Apple added a mitigation in the latest iOS:
    Crashing programs can (optionally) have an exponentially growing delay on restarting from crashes, which prevents attacks that need to repeatedly crash the service to extract information or get lucky

Pointer Protection: Modern 64b ARM 8.3 Pointer Authentication

- https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf
- ARM64 uses 64b pointers
- Idea: Since our pointers are 64b but we are only using say 42b of them...
  - Lets use that upper 22b to encrypt/protect pointers of various types!
- New instructions:
  - `PAC` -> Set Pointer Authentication Code
    - Sets the upper bits with a cryptographic checksum
  - `AUT` -> Check and Remove Pointer Authentication Code
    - If the check is invalid, it will instead put an error in the checksum space: If the pointer is dereferenced it causes an error
  - `XAUT` -> Strip PAC without checking
- Instructions are in NO-OP space if the processor doesn't support them

Plus some non-NOOP higher performance options

- When you know you will be running on a processor which supports it
  - check & return:
    Check the return address has a valid PAC and if so, return
  - check & load:
    Check the PAC and if so, load the pointer
  - check & branch:
    Check the PAC and if so, do a jump-and-link to that pointer
- Allows the complete elimination of the overhead for checking!
  - Well, cheat: You cause it to trigger an exception on instruction committing and just assume the pointer is valid to start with...

How To Use...

- There are 5 secrets for pointer protection
  - These contain random 128b secrets that are used to authenticate the pointer: Provided by the OS
  - Two for data (DA/DB), two for instruction (IA/IB), and one general purpose (GA)
- These are contained in processor registers (ARM equivalent of the CSRs), and are ***not readable to the program itself!***
  - Key property: An information leakage vulnerability can't defeat this protection on a user-level program
  - But it could on a kernel level program:
    Solution would be to also have a secret random to the CPU that is included but non readable
- Other workaround: find a vulnerability that can trick the program into authenticating new pointers it shouldn't, or be able to reuse authenticated pointers in another context

So in practice

- The PAC is a function of the pointer, an additional register (or register 0) and the hidden secret
  - **`PACIA x30 sp`**
    `AUTIA x30 sp`
    Protect/Authenticate x30 as a function of x30, sp, and the secret data associated with the Instruction A context (x30 is the default link register for ARM == **`ra`** in RISC-V)
- Thanks to crypto-magic discussed in 161, the PAC's "look random"
  - Changing a single bit of anything should result in something looking totally different and random
- So to guess a 22 bit PAC would be 1 in 4 million odds.

So Cheaper Stack Canaries...

- On function entry: Create the PAC for the return address
  - Using the stack pointer as the context itself:
    This means the return address can't even be moved
- On function exit: Check & return as normal
- With backwards compatibility: only 2 instructions
  - **`PACIA`** on function start, **`AUTIA`** on function end
- Without backwards compatibility: only 1 instruction!
  - Just the **`PACIA`** on function start and a check & return on exit
  - Saves 8 instructions... Or >85%!
- Only 22 bits of entropy but...
  - If you get more than a few failures, just keep the program dead!

Or Protecting vtable pointers...

- When you allocate a new C++ object...
  - The first thing is a "vtable pointer", really a pointer to an array of pointers to functions
    - Attackers want to overwrite this with their own version
  - Protect the vtable pointer with a context and register 0:
    One additional instruction when calling **`new()`**
  - Then have the vtable itself live in read-only space so it can't be overwritten
- Now when calling a virtual function...
  - Check & Load the vtable pointer (RISC-V like pseudocode):
    eg, if the object pointer is in s0, the vtable pointer is at the start of s0...

```
LDRAA t0 0(s0) # Load 0 + s0, authenticated with data A
LW    t0 X(t0) # X == the specific function to call
JALR  t0       # Actually call it
```

- Now you ***can't*** overwrite a C++ object's vtable pointer to something else without either being very lucky, finding a separate vulnerability, or replacing with another valid pointer that you acquire...
  And the overhead is literally ***nothing***!
  - Apart from you need to recompile and using the latest ARM silicon, that is

Probably the biggest benefit for Apple going to ARM

- MacOSX ARM will be able to ***assume*** PAC support!
  - Since it is Apple A12 or newer processors only
  - Latest iOS also just started really aggressively turning on PAC support
- Can therefore use the more efficient primitives:
  - Check & Branch Register, Check & Load, Check & Return
    which all eliminate the instruction needed in a separate check
  - Usable in both the kernel and user space:
    Acts to harden both applications and the underlying OS
- x86 has nothing like this in the pipeline!
- If you have a choice of architecture for a product: ARM 8.3+
  - This gives you so much real-world security for crappy C-code

RISC-V 16b ISA

- Observation:
  Although we encode instructions with 32b, a lot of the instructions follow common patterns
  - Some registers used a lot more than others
  - Immediates are often small
  - Same source and destination for 3-operand operations
- So the optional "C" instruction set can be mixed in on a per-instruction basis
  - Look at the first two bits of an instruction can determine its type
    - 32b or 16b
  - For a C instruction, PC <- PC + 2 instead of PC + 4
    - And now normal instructions only need to be half-word aligned
- Results in ~30% smaller code
  - Roughly the same performance gain as ***doubling*** the icache!

The Instruction Encoding

Format	Meaning	15 14 13	12	11 10	9 8 7	6 5	4 3 2	1 0
CR	Register	funct4		rd/rs1		rs2		op
CI	Immediate	funct3	imm	rd/rs1		imm		op
CSS	Stack-relative Store	funct3	imm			rs2		op
CIW	Wide Immediate	funct3	imm				rd′	op
CL	Load	funct3	imm		rs1′	imm	rd′	op
CS	Store	funct3	imm		rs1′	imm	rs2′	op
CB	Branch	funct3	offset		rs1′	offset		op
CJ	Jump	funct3	jump target					op

RVC Register Number	000	001	010	011	100	101	110	111
Integer Register Number	x8	x9	x10	x11	x12	x13	x14	x15
Integer Register ABI Name	s0	s1	a0	a1	a2	a3	a4	a5
Floating-Point Register Number	f8	f9	f10	f11	f12	f13	f14	f15
Floating-Point Register ABI Name	fs0	fs1	fa0	fa1	fa2	fa3	fa4	fa5

Stack Relative Load/Stores:
Shrink preamble/postamble code by nearly 50%!

- Immediates are 0-extended
  - Because we write up from the stack pointer
- Immediates assume basic alignment (lower two bits 0 for words, three for doubles...)

15 13	12	11 7	6 2	1 0
funct3	imm	rd	imm	op
3	1	5	5	2
C.LWSP	offset[5]	dest≠0	offset[4:2|7:6]	C2
C.LDSP	offset[5]	dest≠0	offset[4:3|8:6]	C2
C.LQSP	offset[5]	dest≠0	offset[4|9:6]	C2
C.FLWSP	offset[5]	dest	offset[4:2|7:6]	C2
C.FLDSP	offset[5]	dest	offset[4:3|8:6]	C2

15 13	12 7	6 2	1 0
funct3	imm	rs2	op
3	6	5	2
C.SWSP	offset[5:2|7:6]	src	C2
C.SDSP	offset[5:3|8:6]	src	C2
C.SQSP	offset[5:4|9:6]	src	C2
C.FSWSP	offset[5:2|7:6]	src	C2
C.FSDSP	offset[5:3|8:6]	src	C2

Register Relative Loads & Stores

- Same zero-extending and alignment tricks

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	imm	rd′	op
3	3	3	2	3	2
C.LW	offset[5:3]	base	offset[2|6]	dest	C0
C.LD	offset[5:3]	base	offset[7:6]	dest	C0
C.LQ	offset[5|4|8]	base	offset[7:6]	dest	C0
C.FLW	offset[5:3]	base	offset[2|6]	dest	C0
C.FLD	offset[5:3]	base	offset[7:6]	dest	C0

15 13	12 10	9 7	6 5	4 2	1 0
funct3	imm	rs1′	imm	rs2′	op
3	3	3	2	3	2
C.SW	offset[5:3]	base	offset[2|6]	src	C0
C.SD	offset[5:3]	base	offset[7:6]	src	C0
C.SQ	offset[5|4|8]	base	offset[7:6]	src	C0
C.FSW	offset[5:3]	base	offset[2|6]	src	C0
C.FSD	offset[5:3]	base	offset[7:6]	src	C0

Jumps & Branches

15 13	12 2	1 0
funct3	imm	op
3	11	2
C.J	offset[11|4|9:8|10|6|7|3:1|5]	C1
C.JAL	offset[11|4|9:8|10|6|7|3:1|5]	C1

15 12	11 7	6 2	1 0
funct4	rs1	rs2	op
4	5	5	2
C.JR	src≠0	0	C2
C.JALR	src≠0	0	C2

15 13	12 10	9 7	6 2	1 0
funct3	imm	rs1′	imm	op
3	3	3	5	2
C.BEQZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1
C.BNEZ	offset[8|4:3]	src	offset[7:6|2:1|5]	C1

And Then Assorted ALU instructions...

- Load 6 bit immediate values
  - Either to the lower 6 bits or bits 17-12
    - Useful for smaller immediates
- ADDI to self with small immediate
  - And a special form for a scaled by 16 immediate to the stack pointer
  - And another one to add a 0-extended immediate to the stack pointer to get addresses of stack-allocated variables
- Left shift more important than right shift:
  - Can left shift any register, but right shift only the encoded 8...
- Two register operations rather than 3 for the basic ALU ops:
  - Add, subtract, and, or, xor

Result is massive savings...

- Program preamble/postamble
  - Basically shrunk by 50%
- Common array offset access
  - How many times do you access **`x[0]`** or **`x[small positive integer]`**?
- Shorter jump offsets
- Branch = or != 0
  - So all **`beq x0`** and **`bne x0`**...
- Control logic complexity very low even for an aggressive processor
  - Easy decision tree to know length of instructions compared with something like x86