DISCUSSION 4/22/20

SIDE CHANNELS

LOGISTICS

- PA2 due date extension to Tuesday 4/28/20
- PA3 will be released Thursday 4/23/20

WHAT DO YOU MEAN BY CHANNEL?

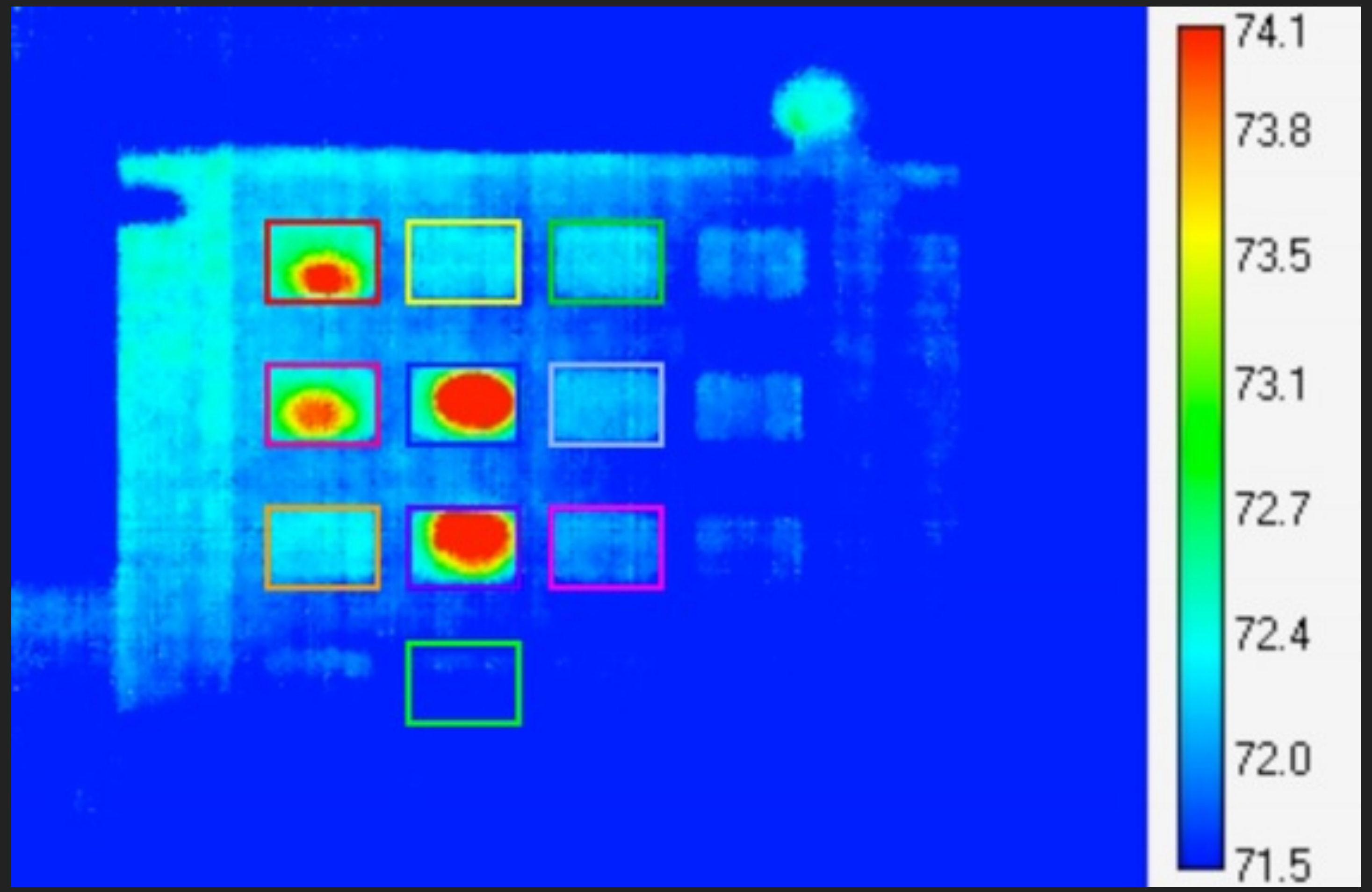
- in this context, a channel is a means of conveying information
- consider a password checking function $f(x) \quad x \mapsto \{0,1\}$
- the intended information channel of the function is the output: $\{0,1\}$
- in ideal circumstances, a user passes x to f, and may only observe $f(x)$

SO WHAT'S A SIDE CHANNEL?

- in actuality, f must be implemented/processed to run on physical devices
- a side channel is any channel of information produced as a side-effect of conveying information along the primary/intended channel.
- break out rooms to brainstorm real world side channels

EXAMPLE SIDE CHANNELS

- timing
  - is the output produced in the same amount of time for each input?
- thermal
  - infrared pictures of pin pads can detect pressed keys
- memory
  - is memory accessed the same way in all cases?



Heat of the Moment: Characterizing the Efficacy of Thermal Camera-Based Attacks

PA3

- two part assignment on side channels
  - memhack (memory-based side channel)
  - timehack (timing-based side channel)

- in both the goal is to programmatically guess the password checked by check_pass in sysapp.c

sysapp.c

- check_pass
  - password to check (*pass) is passed by reference
  - check_pass loops over characters checking against true password sequentially
  - correct_pass is static in the given vm, but its value will change for grading so solution should generalize
  - delay is added to make time hack more feasible
- hack_system
  - solution should call this on the password when it is found

```c
void delay() {
    int j, q;
    for (j = 0; j < 100; j++) {
        q = q + j;
    }
}

int check_pass(char *pass) {
    int i;
    for (i = 0; i <= strlen(correct_pass); i++) {
        delay();  // artificial delay added for timehack
        if (pass[i] != correct_pass[i])
            return 0;
    }
    return 1;
};

void hack_system(char *correct_pass) {
    if (check_pass(correct_pass)) {
        printf("OK: You have found correct password: '%s'\n", correct_pass);
        printf("OK: Congratulations!\n");
        exit(0);
    } else {
        printf("FAIL: The password is not correct! You have failed\n");
        exit(3);
    };
};
```

MEMORY SIDE-CHANNELS

- memory is protected by OS (principle of least privilege / privilege separation)
  - processes have UID and are memory isolated
  - files have permissions by (User/Group/All)
- invalid memory access results in a hardware segfault signal
  - OS passes signal along to offending process
  - in c, SIGSEGV signal is raised and may be handled by the program

memhack/memhack.c

```c
int demonstrate_signals() {
    char *buf = page_start;

    // this call arranges that _if_ there is a SEGV fault in the future
    // (anywhere in the program) then control will transfer directly to this
    // point with sigsetjmp returning 1
    if (sigsetjmp(jumpout, 1) == 1) {
        // Code in this if block will execute whenever a
        // segfault signal is produced
        return 1; // we had a SEGV
    }
    signal(SIGSEGV, SIG_DFL);
    signal(SIGSEGV, &handle_SEGV);

    // We will now cause a fault to happen
    *buf = 0;
    return 0;
}
```

- creating signals
  - we can mark a section of memory as off limits to all with:
  - mprotect(page_start, page_size, PROT_NONE) == -1)
- intercepting signals
  - demonstrate_signals shows how segfault signal can be intercepted

memhack/memhack.c

- so, we can
  - set access rights to memory
  - intercept all segfault signals
- key features of the password checker we seek to crack:
  - takes arguments by reference
  - checks characters sequentially
  - short circuits on first invalid character
- how can we utilize the above factors to create a side channel and bypass check_pass?

memhack/memhack.c


```c
int check_pass(char *pass) {
    int i;
    for (i = 0; i <= strlen(correct_pass); i++) {
        delay();  // artificial delay added for timehack
        if (pass[i] != correct_pass[i])
            return 0;
    }
    return 1;
};
```

- pass the pointer to our guess to check_pass
  - M's match, i is incremented
  - Y's match, i is incremented
  - G != P, 0 is returned
  - what does the submitter learn from this trial?

memhack/memhack.c


```
int check_pass(char *pass) {
    int i;
    for (i = 0; i <= strlen(correct_pass); i++) {
        delay();  // artificial delay added for timehack
        if (pass[i] != correct_pass[i])
            return 0;
    }
    return 1;
};
```

- what if we use **mprotect** to segment our guess?
  - **A's** match, **i** is incremented
  - segfault triggered when **check_pass** attempts to check second character in protected memory!
  - what does the submitter learn from this experiment?

timehack/timehack.c

```
int check_pass(char *pass) {
    int i;
    for (i = 0; i <= strlen(correct_pass); i++) {
        delay();  // artificial delay added for timehack
        if (pass[i] != correct_pass[i])
            return 0;
    }
    return 1;
};
```

- key features of the password checker we seek to crack (same sysapp.c as in memhack):
  - checks characters sequentially
  - short circuits on first invalid character
  - performs same operations when checking each character
- using rdtsc macro we can get the current cycle counter value as type long
  - cycle counter increments by 1 for each instruction the system performs
    - BEWARE: this includes instructions performed by other programs on the system!
  - we can wrap a function call with calls to rdtsc and use the difference in the instruction counter before and after the function call as an estimate of the time the function call took to complete
- how can we utilize the above factors to create a side channel and bypass check_pass?

timehack/timehack.c

- we can run check_pass against all possible first characters and record how many cycles passed
  - the first character will be the only thing checked in all but one trial
  - only checking one character should take roughly the same number of cycles each time, while checking two should take more
  - we can then repeat the process fixing the first character and trying all possible second characters

timehack/timehack.c

- but wait, that seems too easy!
  - the cycle counter increments by 1 for each instruction the system performs, including instructions for other processes
  - each guess should be treated as a trial
  - performing multiple trials for each guess we can form statistics: e.g. mean, median, mode, etc.
  - we are interested in the expected runtime when our program hasn't been sidelined part way through execution due to multithreading

timehack/timehack.c

- mode:
  - sample measurements: [24, 21, 22, 11, 670, 22, 18]
  - mode is the most frequently observed value in a sample
  - issues:
    - given range of possible integer values may be rare to encounter repeat values
    - the mode is thus inconsistent and wouldn't be expected to approach our desired expected value

timehack/timehack.c

- mean:
  - sample measurements: x = [24, 21, 22, 11, 670, 22, 18]
  - $\bar{x} = \frac{1}{len(x)} \sum_i x[i] = 112.57$
  - issues:
    - sensitive to outliers.
      - e.g. removing the outlier 670 would result in a mean of 19.66
    - may be able to apply the central limit theorem (taking into account variance), but that's overkill when we have …

timehack/timehack.c

- median: the 'middle' number in a sample
  - sample measurements: x = [24, 21, 22, 11, 670, 22, 18]
    - x.sort() -> [11, 18, 21, 22, 22, 24, 670]
  - if len(x) % 2 == 0
    - y = [11, 18, 21, 22, 22, 24]
      - median(y) = $\frac{21 + 22}{2} = 21.5$
- the median is robust to outliers! removing 670 only shifted the median by .5

timehack/timehack.c

- final tips for timehack:
  - perform many trials to form robust statistics (the more the merrier)
  - calculate the median as a robust estimate of the runtime for a guess
  - consider implementing backtracking
    - if your runtime stays consistent as you add "correct" letters, the program likely isn't checking more characters.
    - backtrack until the last character for which a significant increase in runtime was observed across the board
      - valid solutions exist without backtracking, but it will improve robustness