**Problem 1.** Simple questions

1. (a) First fit: 20KB, 10KB, 18KB.
   (b) Best fit: 12KB, 10KB, 9KB.
   (c) Quick fit: 12KB, 10KB, 9KB.
2. The effective instruction time is $\frac{10k}{10k+n}$.
3. The four page frames' value change as follows.

$$00000000 \to 01101110$$
$$10000000 \to 01001001$$
$$10000000 \to 00110111$$
$$10000000 \to 10001011$$

**Problem 2.** Page tables

1. Inverted page table.

   To solve the issue of large memory space, inverted page table is introduced. In this design, there is one entry per page frame in real memory, rather than one entry per page of virtual address space. For example, with 64-bit virtual addresses, a 4-KB page size, and 4 GB of RAM, an inverted page table requires only 1,048,576 entries. The entry keeps track of which (process, virtual page) is located in the page frame.

   Although inverted page tables save lots of space, at least when the virtual ad- dress space is much larger than the physical memory, they have a serious down- side: virtual-to-physical translation becomes much harder. When process n refer- ences virtual page p, the hardware can no longer find the physical page by using p as an index into the page table. Instead, it must search the entire inverted page table for an entry (n, p). Furthermore, this search must be done on every memory refer- ence, not just on page faults. Searching a 256K table on every memory reference is not the way to make your machine blindingly fast.

2. Multilevel page table.

   The multilevel page table is introduced to avoid keeping all the page tables in memory all the time. In particular, those that are not needed should not be kept around. Suppose, for example, that a process needs 12 megabytes: the bottom 4 megabytes of memory for program text, the next 4 megabytes for data, and the top 4 megabytes for the stack. In between the top of the data and the bottom of the stack is a gigantic hole that is not used.

   When a virtual address is presented to the MMU, it first extracts the PT1 field and uses this value as an index into the top-level page table. Each of these 1024 entries in the top-level page table represents 4M because the entire 4-gigabyte (i.e., 32-bit) virtual address space has been chopped into chunks of 4096 bytes. The two-level page table can be expanded to three, four, or more levels. Additional levels give more flexibility.

**Problem 3.** Research

1. CCS Injection Vulnerability

   In OpenSSL, a bypass vulnerability known as **CVE-2014-0224** exists due to the fragility in the keying materials method within the security process[1]. According to the bug reporter Masashi Kikuchi, OpenSSL accepts ChangeCipherSpec (CCS) incorrectly during a handshake. In a typical scenario, the client and the server should exchange messages.

   ChangeCipherSpec has specific locations during the handshake and OpenSSL has a timing difference when sending and receiving CCS. By using the man-in-the-middle attack, an attacker can affect the CCS transit. With an uniquely devised handshake, the attacker can force one to use a weak keying material from any location and at any authentication level. The vulnerability can be exploited, leading to a possible situation where the attacker is able to obtain any level of information. However, this attack can only occur if the client and server is vulnerable[2].

   To fix the bug as mentioned above, we have to make sure that the client sent a message and the server received it. The code below gives a simple model of correctly sent handshake messages:

```
enum {
    hello_request(0), client_hello(1), server_hello(2),
    certificate(11), server_key_exchange (12),
    certificate_request(13), server_hello_done(14),
    certificate_verify(15), client_key_exchange(16),
    finished(20), (255)
} HandshakeType;

struct {
    HandshakeType msg_type;    /* handshake type */
    uint24 length;             /* bytes in message */
    select (HandshakeType) {
        case hello_request:       HelloRequest;
        case client_hello:        ClientHello;
        case server_hello:        ServerHello;
        case certificate:         Certificate;
        case server_key_exchange: ServerKeyExchange;
        case certificate_request: CertificateRequest;
        case server_hello_done:   ServerHelloDone;
        case certificate_verify:  CertificateVerify;
        case client_key_exchange: ClientKeyExchange;
        case finished:            Finished;
    } body;
} Handshake;
```

2. Meltdown, corresponds to vulnerability CVE-2017-5754.

   Spectre, corresponds to vulnerability CVE-2017-5753/CVE-2017-5715.

   Meltdown affects almost all Intel CPUs (since 1995) and some ARM CPUs. Spectre has a much broader reach, affecting Intel, ARM, and AMD[1].

   With the Meltdown vulnerability, a low-privilege user can access the contents of the kernel and obtain information about the underlying local operating system; when a user accesses a website containing a malicious Spectre exploit via a browser, personal information such as account, password, email, etc. can be compromised; in a cloud service scenario, Spectre can be used to break through the isolation between users and steal data from other users.

   The Meltdown vulnerability affects almost all Intel CPUs and some ARM CPUs, while Spectre affects all Intel CPUs and AMD CPUs, as well as mainstream ARM CPUs. From PCs, servers[2], cloud computer servers to mobile smartphones, all are affected by these two sets of hardware vulnerabilities.

   This vulnerability exploits the handling of failures in CPU execution. Since CPUs are now designed to provide performance, out-of-order execution and predictive execution have been introduced.

   This means that the CPU does not execute the instructions in strict order, but groups them in parallel according to their relevance, and then summarizes the results of each group of instructions. Predictive execution is where the CPU predicts the result of a condition based on the information it has, and then selects the corresponding branch to execute in advance. When these two types of execution encounter an exception, the CPU will discard the result of the previous execution, restore the CPU to the correct state before the out-of-order execution or predicted execution, and then continue to execute the correct instructions to ensure that the program can be executed correctly and continuously. The problem, however, is that the CPU does not clear the CPU cache when state is restored, and these two sets of vulnerabilities exploit this design flaw in a side-channel attack. The corresponding exploit for out-of-order execution is Meltdown, while the corresponding exploit for predictive execution is Spectre.

   The result run of poc on this vulnerability is given as follows.

```
$ make
cc -O2 -msse2   -c -o meltdown.o meltdown.c
cc   meltdown.o   -o meltdown
$ ./run.sh
looking for linux_proc_banner in /proc/kallsyms
protected. requires root
+ find_linux_proc_banner /proc/kallsyms sudo
+ sudo awk
```

[1] https://support.microsoft.com/en-us/help/4073119/protect-against-speculative-execution-side-channel-vulnerabilities-in

[2] https://support.microsoft.com/en-us/help/4072698/windows-server-speculative-execution-side-channel-vulnerabilities

```
        /linux_proc_banner/ {
                if (strtonum("0x"$1))
                        print $1;
                exit 0;
        } /proc/kallsyms
+ linux_proc_banner=ffffffffa3e000a0
+ set +x
cached = 29, uncached = 271, threshold 88
read ffffffffa3e000a0 = 25 %
read ffffffffa3e000a1 = 73 s
read ffffffffa3e000a2 = 20
read ffffffffa3e000a3 = 76 v
read ffffffffa3e000a4 = 65 e
read ffffffffa3e000a5 = 72 r
read ffffffffa3e000a6 = 73 s
read ffffffffa3e000a7 = 69 i
read ffffffffa3e000a8 = 6f o
read ffffffffa3e000a9 = 6e n
read ffffffffa3e000aa = 20
read ffffffffa3e000ab = 25 %
read ffffffffa3e000ac = 73 s
read ffffffffa3e000ad = 20
read ffffffffa3e000ae = 28 (
read ffffffffa3e000af = 62 b
read ffffffffa3e000b0 = 75 u
read ffffffffa3e000b1 = 69 i
read ffffffffa3e000b2 = 6c l
read ffffffffa3e000b3 = 64 d
read ffffffffa3e000b4 = 64 d
read ffffffffa3e000b5 = 40 @
VULNERABLE
VULNERABLE ON
```

**Problem 4.** MINIX3

1. Use the following code `ls minix/servers/vm` and we have

```
minix/servers/vm/vm.h
minix/servers/vm/proto.h
minix/servers/vm/pt.h
minix/servers/vm/arch/i386/pagetable.h
minix/servers/vm/pagetable.c
```

2. Inside the `pagetable.h` we find the definition macro *`#define I386_PAGE_SIZE^^I^^I4096`*, thus the size is 4096 bit.

3. We find the structure as defined in the header file.

```
typedef struct {
 /* Directory entries in VM addr space - root of page table. */
 u32_t *pt_dir; /* page aligned (ARCH_VM_DIR_ENTRIES) */
 u32_t pt_dir_phys; /* physical address of pt_dir */

 /* Pointers to page tables in VM address space. */
 u32_t *pt_pt[ARCH_VM_DIR_ENTRIES];

 /* When looking for a hole in virtual address space, start
  * looking here. This is in linear addresses, i.e.,
  * not as the process sees it but the position in the page
  * page table. This is just a hint.
  */
 u32_t pt_virtop;
} pt_t;
```

4. The functions are defined as follows.

```
void pt_init(void);
void vm_freepages(vir_bytes vir, int pages);
void pt_init_mem(void);
void pt_check(struct vmproc *vmp);
int pt_new(pt_t *pt);
void pt_free(pt_t *pt);
int pt_map_in_range(struct vmproc *src_vmp, struct vmproc *dst_vmp,
 ↪  vir_bytes start, vir_bytes end);
int pt_ptmap(struct vmproc *src_vmp, struct vmproc *dst_vmp);
int pt_ptalloc_in_range(pt_t *pt, vir_bytes start, vir_bytes end, u32_t
 ↪  flags, int verify);
void pt_clearmapcache(void);
int pt_writemap(struct vmproc * vmp, pt_t *pt, vir_bytes v, phys_bytes
 ↪  physaddr, size_t bytes, u32_t flags, u32_t writemapflags);
int pt_checkrange(pt_t *pt, vir_bytes v, size_t bytes, int write);
int pt_bind(pt_t *pt, struct vmproc *who);
void *vm_mappages(phys_bytes p, int pages);
void *vm_allocpage(phys_bytes *p, int cat);
void *vm_allocpages(phys_bytes *p, int cat, int pages);
void *vm_allocpagedir(phys_bytes *p);
int pt_mapkernel(pt_t *pt);
void vm_pagelock(void *vir, int lockflag);
int vm_addrok(void *vir, int write);
int get_vm_self_pages(void);
int pt_writable(struct vmproc *vmp, vir_bytes v);
```

**Problem 5.** Linux

When replacing a page, if the replacement page is a page that will soon be accessed again, another missing page break will be followed by a new missing page break soon after, which will result in a dramatic decrease in overall system efficiency.

The most important line here is line 15, which is significantly slower than `array[i][j] = i*j;`, this is because the differences in memory access pattern. First example writes to memory serially, while the second one thrashes L1 cache by loading 0-15 to line 0, write value to cell 0, flush it, load address 32768-32783 to line 0, write value to address 32768, flush cache line, load the next RAM address to line 0 write and flush it and so on. These access patterns show 10 times difference in application performance.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <time.h>
4
5  #define L1_CACHE_CAPACITY   (3331457 / sizeof(int))
6
7  int main(void) {
8    int i, j;
9    int **array;
10
11   array = (int **)malloc(L1_CACHE_CAPACITY*L1_CACHE_CAPACITY);
12
13   for (i=0; i<L1_CACHE_CAPACITY; i++)
14     for (j=0; j<L1_CACHE_CAPACITY; j++)
15       array[j][i] = i*j;
16
17   return 0;
18 }
```

**Problem 6.** Dirty COW

Dirty COW (CVE-2016-5195) is a privilege escalation vulnerability in the Linux Kernel. Every version of Linux in this decade, including Android, desktop and server versions, has been affected by it. The vulnerability allows malicious attackers to easily bypass common vulnerability defenses and target millions of users. Although a patch has been applied to the vulnerability, an in-depth study of the patch and its contents by foreign security firm Bindecy found that the patch for the Dirty Cow vulnerability is still flawed, giving rise to the "Big Dirty Cow" vulnerability.

The vulnerability is in the `get_user_pages` function[3], which gets the physical address after the virtual address called by the user process, and the caller needs to declare the specific operation it wants to perform (e.g. write/lock operations), so memory management can prepare the corresponding memory page. Specifically, that is, when a write to a privately mapped memory page is performed, it undergoes a COW (copy on write) process, i.e. the

[3]https://github.com/dirtycow/dirtycow.github.io

read-only page is copied to generate a new page with write access, the original page may be privately protected as unwritable, but it can be mapped and used by other processes. The user can also re-write to disk after modifying the contents of the new page after COW. The purpose of the entire while loop is to fetch each page in the request page queue, repeating operations until the need to build all memory mappings is satisfied, which is where the retry tag comes into play.

# References

[1] OpenSSL continues to bleed out more flaws – more critical vulnerabilities found". Cyberoam Threat Research Labs. 2014. Archived from the original on 2014-06-19. Retrieved 2020-07-21.

[2] https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0224