Journal of The Korea Institute of Information Security & Cryptology
VOL.33, NO.3, Jun. 2023
ISSN 1598-3986(Print)
ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2023.33.3.437

ARM64 아키텍처 기반 하드웨어 보안기술 분석 및 보안성 진단*

심 명 규,[1†] 이 호 준[2‡]
[1,2]성균관대학교 (대학원생, 교수)

Security Analysis of ARM64 Hardware-Based Security*

Myung-Kyu Sim,[1†] Hojoon Lee[2‡]
[1,2]Sungkyunkwan University (Graduate student, Professor)

요 약

메모리 보호 관련 기술은 수십 년간 진행되어오고 있고 프로그램의 실행 보호에 있어 중요한 기술이다. 메모리 보호를 수행하기 위해 ARM에서는 새로운 하드웨어 보안기술을 개발하였고, 실제 하드웨어에 적용되기에 이르렀다. 하지만 하드웨어 메모리 보호 기술이 적용된 하드웨어는 수가 많지 않고 아직 연구가 활발히 진행되지 않고 있다. 우리는 새로운 하드웨어 메모리 보호 기술인 Pointer Authentication Code를 실제 하드웨어에서 동작하는 방식과 과정, 그리고 보안성에 대한 진단을 수행한다. 이를 통해 앞으로의 하드웨어 보안기술의 적용 방향과 사용, 보안성에 대해 알아내고 이를 응용할 수 있을 것이다.

ABSTRACT

Memory protection has been researched for decades for program execution protection. ARM recently developed a new hardware security feature to protect memory that was applied to real hardware. However, there are not many hardware with hardware memory protection feature and research has not been actively conducted yet. We perform diagnostics on how and how it works on real hardware, and on security, with a new hardware memory protection feature, named 'Pointer Authentication Code'. Through this research, it will be possible to find out the direction, use, and security of future hardware security technologies and apply to the program.

Keywords: ARM64, Pointer Authentication Code

I. 서 론

C/C++과 같이 메모리 보호가 이루어지지 않는 언어로 작성된 프로그램은 수십 년간 개발되어왔고, 다양하게 존재한다. 현대의 소프트웨어 공격은 실행 흐름 변조를 통한 공격을 기본으로 하며, 실행 흐름 변조는 메모리 변조로부터 시작된다. 따라서 메모리 보호가 이루어지지 않는 프로그램에 대한 메모리 보호를 하는 연구는 수십 년간 진행되어오고 있다. 메모리 보호를 소프트웨어적으로 해결하는 연구가 상당수 진행되고 있으나[2,13,14,15], 성능의 저하가 극심함으로 인해 실제 프로그램에 적용하기에는 한계가 존재한다. 소프트웨어 방식의 메모리 보호 기술이 가지고 있는 한계점인 성능의 문제를 해결하기 위해 최

Received(02. 27. 2023), Modified(03. 17. 2023), Accepted(04. 10. 2023)
* 이 연구는 2023년도 정부의 재원으로 한국연구재단(교육부) 기초연구사업 (NRF-2022R1C1C1010494), 정보통신기획평가원(융합보안핵심인재양성 (No. 2019-0-01343), 국제공동연구사업 (No. 2020-0-00666))의 지원을 받아 수행된 연구입니다.
† 주저자, myungkyu.sim@skku.edu
‡ 교신저자, hojoon.lee@skku.edu(Corresponding author)

근 ARM에서 하드웨어를 사용한 메모리 보호 기술을 선보였다[1]. 하지만 아직도 하드웨어 보안기술이 실제로 적용되어있는 하드웨어는 개발 초기 단계이기 때문에 그 수가 많지 않다. 앞으로 하드웨어 보안기술이 적용된 기기는 기존의 소프트웨어의 성능 한계를 이겨내고 연구가 진행됨에 따라서 점차 늘어날 것이고 다양해질 예정이다. 따라서 새로운 하드웨어 보안기술에 관한 연구가 필요하다. 아직은 실제 사용 환경에서의 보안에 관한 연구가 많이 이루어지지 않았고, 실제로 적용된 하드웨어에 대안 보안 연구도 다양하게 이루어지지 않았다. 따라서 우리는 ARM 하드웨어 보안기술이 적용되어있는 하드웨어의 적용 방식과 보안성에 대한 진단을 수행한다.

II. 배경 및 관련 연구

2.1 메모리 보호 기술

실행 흐름 보호를 위한 메모리 보호 기술은 다양하게 연구되어왔다. 할당된 메모리를 벗어나는 것을 확인하여 메모리가 다른 값으로 덮어씌워지는 것을 방지하는 AddressSanitizer[13]나 메모리에 태그를 주어 특정 메모리 영역이 아닌 부분을 사용하는 것을 방지하는 메모리 태깅[5] 등 메모리를 보호하는 연구가 진행되고 있다. 특히 메모리를 하나의 신뢰하지 않는 중간 저장 장치로 보고 메모리 내부 포인터의 변조를 해시 등으로 보호하는 연구인 CCFI[2]로부터 시작하는 포인터의 무결성 보호 방식은 캐시 내부에 메타 데이터를 넣어 메모리의 변조를 방지하는 ZeRØ[14]와 RISC-V를 사용하여 메모리 도메인을 만들고 그에 대한 태깅, 변위 및 암호화를 통해 포인터의 변조를 방지하는 Morpheus[15] 같은 연구를 통해 메모리의 포인터를 보호하는 방식이 연구되었으며, 최근의 연구는 하드웨어를 사용하여 메모리 보호 기술을 기존의 소프트웨어 방식보다 더 빠르게 하는 방향으로 진행되고 있다. 그중 현재 가장 활성화되어있는 PAC은 ARM에서 개발된 하드웨어를 사용한 포인터 인증 방식으로, 포인터를 사용하기 전에 인증하여 포인터의 무결성을 보장한다.

2.2 Pointer Authentication

Pointer Authentication(PA)은 포인터의 값에 암호학 서명을 넣어 포인터의 무결성을 보장하는 방법이다. 포인터의 상위 특정 비트에 Pointer Authentication Code(PAC)를 넣고 서명된 포인터를 사용할 시 검증하여 포인터의 변조 여부를 확인하는 방식으로 포인터의 무결성을 확인할 수 있다.

ARMv8.3-A에서 새로 추가된 PAC은 포인터에 대한 무결성을 보장하기 위한 하드웨어 기술이다. PAC은 128비트 키, 64비트 포인터, 64비트 수정자를 통해 포인터에 서명한다. PAC에서는 포인터의 종류에 따라 키를 다르게 사용하고, 총 5개의 키를 사용한다. 수정자로는 일반적으로 스택 주소를 사용한다.

PAC에서 사용하는 5개의 키는 다음과 같다. 코드 포인터를 위한 APIAKey와 APIBKey, 데이터 포인터를 위한 APDAKey와 APDBKey, 일반적인 목적으로 사용되는 APGAKey가 존재한다. 해당 키들은 시스템 레지스터에 저장되어 유저 모드인 EL0에서는 읽을 수 없다. 따라서 유저가 PAC 서명에 사용되는 키를 읽거나 변경할 수 없다[4].

PAC을 사용하기 위해서 ARMv8.3-A에서부터 새로운 명령어가 추가되었다. "PAC"으로 시작되는 명령어와 "AUT"로 시작되는 명령어 두 가지의 경우로 나눌 수 있다. "PAC"으로 시작하는 명령어는 포인터에 PAC을 부여하는 명령어로 "PACIASP"와 같이 사용되는 키와 수정자에 따라 명령어를 다르게 사용한다. "AUT"로 시작되는 명령어는 PAC가 추가된 포인터를 검증하여 포인터를 사용할 수 있도록 해주는 명령어로 "PAC" 명령어와 마찬가지로 사용되는 키와 수정자에 따라 다른 명령어를 사용하게 된다. 함수의 리턴이나 점프 등 특정한 상황에서 사용할 수 있는 "RETAA"와 같은 명령어도 추가되었다.

macOS에서의 PAC은 포인터에 주소가 대입되는 시점에 계산하여 포인터를 보호하는 Forward edge, 함수의 종료 후 리턴 주소를 보호하는

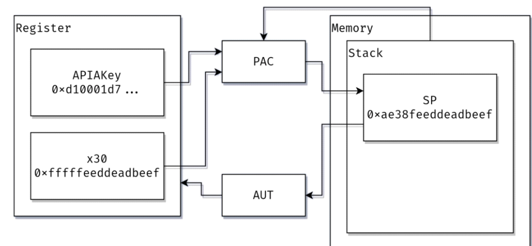

Fig. 1. Process of PAC and AUT

Backward edge 두 가지의 보호를 모두 지원한다〔8〕. 포인터가 PAC으로 서명이 되고 난 이후 포인터를 사용하기 직전 포인터에 대한 검증을 진행하고, 이에 대한 검증이 실패하였을 경우 오류 처리를 진행한다.

2.3 PAC 기반 메모리 보호 기술

P. Nasahl 외 2명은 fault attack을 이용하여 간접 점프를 수행하는 공격에서 PAC을 사용하여 간접 점프하는 주소를 보호하여 포인터의 실행을 방지하고 보호하는 방법을 제시하였다. 실행하려는 주소를 보호하기 위해 점프하려고 하는 주소와 간접 주소 분기점 두 가지를 보호하여 공격자가 해당 주소를 실행하는 것을 방지한다〔9〕.

PAL〔10〕은 커널 내부 실행 흐름 보호 방식으로 PAC을 사용하여 보호하는 방법을 제시하였다. 문맥 분석을 통해 함수 포인터와 리턴 주소를 생성하는 지점에 PAC 서명하고 사용하는 지점에서 PAC 인증하도록 하여 포인터의 무결성을 보장하고 이를 정적 검증을 통해 포인터 인증 이후 변경되거나 확인이 안된 실행 흐름이 있는지 등을 확인한다.

A. Fanti 외 4명은 자주 사용되지 않는 레지스터가 중복으로 사용될 때 메모리에 저장된다는 점을 이용한 공격을 방지하기 위하여 LLVM을 활용해 메모리에 레지스터가 저장될 때 PAC을 사용하여 레지스터의 값을 보호하는 방식을 제안하였다. 2개의 임시 레지스터를 사용하여 PAC 처리된 포인터와 메시지 인증 코드를 생성하고 메모리에 저장하여 기밀성과 무결성을 보장한다〔11〕.

PTAuth〔12〕 는 힙에서의 temporal memory corruption을 방지하기 위해 PAC을 사용하여 포인터의 무결성을 보장한다. 이를 위해서 포인터의 생성 시에 PAC 서명을 생성하고 load나 store와 같이 포인터를 사용하기 직전에 인증하도록 컴파일러를 수정하였다.

PACMem〔16〕 은 PAC 명령어를 활용하여 객체의 서명과 크기를 담은 메타 데이터를 생성하고, 이를 활용하여 객체의 temporal safety와 spatial safety를 보장한다. 메타 데이터에는 포인터가 생성되는 시점에 할당되는 임의의 값과 객체의 주소, 크기가 들어간다. 객체의 포인터를 사용할 때는 메타데이터에 있는 주소를 사용하여 PAC을 검증하고 사용하는 주소와 객체의 주소 간 오프셋과 객체 크기를 비교하여 해당 객체 내부의 포인터를 사용하는지 검증한다.

PAC을 공격하는 연구도 활발히 진행되었다.

PACMAN〔6〕 은 커널에서 PAC을 인증하고 실행하는 특정한 gadget을 이용하여 PAC에 대한 검증을 오류 없이 진행, 무차별 대입으로 실제 수행되는 PAC의 값을 알아낸 다음 일반적인 buffer overflow 공격을 수행한다. 이를 위해 PACMAN에서는 커널을 정적 분석하여 PAC을 검증하고 포인터를 실행하는 gadget을 발견, PAC이 실제로 맞는지 검증하기 위한 부채널 공격을 진행하였다. 해당 공격으로 2.94분 이내로 하나의 포인터에 대한 PAC을 알아낼 수 있으며, 이에 대한 정확도는 90%를 보여주었다.

에뮬레이션 된 PAC을 분석한 PAC it up에〔3〕 따르면 공격자는 포인터를 생성하기 이전에 포인터 값과 수정자를 이용하여 공격자가 원하는 PAC 서명된 포인터를 얻을 수 있다. 그리고 이미 서명된 포인터를 저장해두었다가 필요할 때 같은 수정자를 사용하는 함수에서 해당 포인터를 사용하여 서명의 인증을 회피할 수 있다.

III. PAC의 생성 및 관리 과정 분석

본 단원에서는 실제로 PAC이 적용된 하드웨어인 M1에서 PAC이 어떻게 생성되고 관리되는지를 서술한다. OS는 macOS를 사용하였고, 기준 커널은 xnu-7195-60.75 버전으로 분석하였다.

3.1 PAC의 생성 과정

3.1.1 XNU 커널

XNU 커널에서의 PAC은 부팅 시에 생성되며, 커널 초기화 과정에서는 특정 상수로 초기화되나 CPU의 권한 레벨과 PAC 키의 레지스터를 이용하여 PAC 키를 생성한다. 생성된 이후에는 시스템 레지스터에 저장된다.

3.1.2 유저

유저가 사용하는 PAC 키는 프로세스별로 분리되어 있으며, 같은 프로세스 내부의 스레드 간 같은 키를 공유한다. 프로세스를 'fork()'로 새로 생성하였

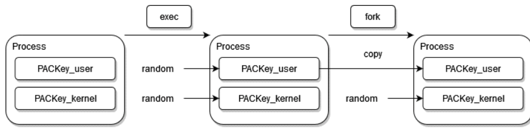

Fig. 2. Generation of PAC on 'exec' and 'fork'

을 시 부모의 프로세스가 가지고 있는 키와 같은 키를 상속받으며, 'exec()'를 통해 새로운 프로세스를 생성하면 새로운 PAC 키를 생성한다. 유저가 사용하는 PAC 키는 task 구조체에서 관리된다.

유저의 APIA, APDA 키는 주소 공간의 공유 영역이 할당될 때 생성된다[7]. 생성된 키는 프로세스가 활성화될 때 struct task의 jop_pid 에 복사되어 저장된다. APIB, APDB 키는 프로세스가 생성될 때 같이 생성되며, struct task의 rop_pid에 저장된다. 위 코드는 XNU 커널 내부의 코드 중 일부분이다. struct task 내부에 rop_pid, jop_pid가 존재하며 해당 변수에 프로세스마다 PAC 키가 할당되게 된다.

```
#if defined(HAS_APPLE_PAC)
#define TASK_ADDITIONS_PAC \
    uint64_t rop_pid; \
    uint64_t jop_pid; \
    uint8_t disable_user_jop;
#else
#define TASK_ADDITIONS_PAC
#endif
```

Fig. 3. PAC key management on XNU struct task

3.2 PAC에서의 키 관리 과정

3.2.1 XNU 커널

XNU 커널의 PAC 키는 기본적으로 시스템 레지스터에 보관된다. 하지만 코어가 절전이 되거나 인터럽트가 일어나는 등 코어의 상태를 저장하게 되는 경우, 커널 내부에 커널의 메모리 일부에 커널의 PAC 키가 저장된다. 커널의 PAC 키는 부팅 시에 확정되고 변경되지 않으므로, 임의의 커널 메모리를 읽을 수 있다면 저장되어있는 PAC의 키값을 읽을 수 있게 된다.

위 코드는 CPU 레지스터를 저장하는 구조체에서 존재하는 부분 중 일부분이다. CPU의 레지스터를

```
typedef struct cpu_data {
    ...
#if defined(HAS_APPLE_PAC)
    uint64_t    rop_key;
    uint64_t    jop_key;
#endif /* defined(HAS_APPLE_PAC) */
    ...
} cpu_data_t;
```

Fig. 4. Struct of CPU register on kernel

커널 메모리에 보관할 때 task 구조체와 같은 이름의 변수로 PAC의 키가 저장되게 된다.

3.2.2 유저

유저 프로세스들의 PAC 키는 task 구조체에서 관리되며, APIA, APDA 키는 jop_pid, APIB, APDB키는 rop_pid에 각각 저장되어있다. A 키는 공유 영역 ID가 같은 경우 같은 키를 사용하며, B 키의 경우, fork()와 같이 메모리 주소 공간을 공유하는 경우 같은 키를 사용한다.

3.3 PAC 사용 과정

본 단원에서는 예시를 통해 PAC 명령어가 실제 어셈블리에서 어떻게 사용되는지 알아본다. 예시 코드는 LLVM 14.0 버전을 사용하였으며, 컴파일하기 위해서는 macOS에서 'arm64e_preview_abi'를 활성화한 이후 컴파일 아키텍처를 'arm64e'로 설정하여야 한다.

3.3.1 Forward Edge

PAC을 사용하여 함수 포인터 등 함수의 간접 호출을 수행하는 과정에서 실행 흐름을 보호할 수 있다. 그림 5를 통해 forward edge가 보호되는 부분

```
void foo();
void bar();
int main () {
    void (*ptr1)(void) = foo;
    void (*ptr2)(void) = bar;
    ptr1();
    ...
}
```

Fig. 5. Forward edge example

을 알아볼 수 있다.

PAC을 사용하여 포인터를 보호하는 경우 함수 포인터가 대입되는 시점에서 PAC이 계산되어 저장된다. 이때 PAC을 계산하기 위한 인자로 함수 포인터와 IA 키, 그리고 수정자로 0을 준다. 이후 함수 포인터를 사용할 시 포인터에 대한 인증을 수행하고 함수를 실행한다.

해당 코드는 예시 코드를 역어셈블하여 얻은 어셈블리 코드이다. 해당 코드를 보면 paciza로 함수 포인터를 PAC 처리하고 해당 포인터를 사용할 시에 blraaz를 사용하여 포인터의 무결성을 확인하는 것을 볼 수 있다.

```
[0x100003f64]: paciza x16
[0x100003f68]: mov    w8, #0x0
[0x100003f6c]: str    w8, [sp, #0x4]
[0x100003f70]: stur   wzr, [x29, #-0x4]
[0x100003f74]: str    x16, [sp, #0x10]
[0x100003f78]: ldr    x8, [sp, #0x10]
[0x100003f7c]: str    x8, [sp, #0x8]
[0x100003f80]: ldr    x8, [sp, #0x10]
[0x100003f84]: blraaz x8
```

Fig. 6. Forward edge assembly example

3.3.2 Backward Edge

함수 종료 후 리턴할 때 리턴 주소를 보호할 수 있다. ARM64에서 함수 호출규약에 의하면 리턴 주소는 x30(LR) 레지스터에 저장된다. 함수 내부에서 또 다른 함수가 실행되는 경우 리턴 주소를 스택에 저장한다. 그림 7을 통해 backward edge가 보호되는 부분을 알아볼 수 있다.

특정 함수가 실행되는 프롤로그 부분에 리턴 주소, IA 키, 함수 스택 포인터를 사용하여 생성된 주소를 스택에 저장한다. 함수가 종료될 시 x30 레지스터에 저장되어있던 리턴 주소를 넣고 이를 같은 수정자와 키를 사용하여 리턴 주소를 인증한다.

해당 코드는 예시 코드를 역어셈블하여 얻은 코드

```
void foo () {
    ...
}
void bar () {
    foo();
}
```

Fig. 7. Backward edge example

```
[0x100003e5c]: pacia  x30, sp
[0x100003e60]: stp    x29, x30, [sp, #-0x10]!
[0x100003e64]: mov   x29, sp
...
[0x100003e68]: bl     0x100003e3c
...
[0x100003e6c]: ldp    x29, x30, [sp], #0x10
[0x100003e70]: retaa
```

Fig. 8. Backward edge assembly example

이다. 예시 코드의 함수인 'bar()'의 시작 부분에서 'pacia x30, sp'를 통해 PAC 서명한 포인터를 생성한다. 그 후 스택에 서명된 포인터를 저장하여 다른 함수를 부를 수 있도록 한다. 마지막으로 함수 종료 시 저장되어있던 리턴 주소를 x30 레지스터에 불러오고 retaa 명령어를 통해 리턴과 동시에 해당 포인터에 대한 검증을 실행한다.

3.4 PAC에서의 오류 처리

PAC을 인증하는 데 있어 실패하였을 경우 커널과 유저 모두 유효하지 않은 주소를 가지는 포인터를 받는다. 인증 실패한 포인터의 결괏값은 인증을 성공한 포인터에서 오류 비트가 추가된 값이며, 실제로 오류를 일으키는 지점은 PAC을 인증하는 지점이 아닌 인증 실패한 포인터를 사용하는 시점에서 일어나게 된다[4]. 오류 비트는 상위 3비트에 1로 표시되며 사용할 때 페이지 오류로 프로그램의 오류가 생기게 된다[18]. 따라서 인증을 실패한 포인터에서 오류 비트의 변경만으로 PAC 처리된 포인터를 제약 없이 사용할 수 있게 된다.

3.5 객체의 함수 처리

C++과 같은 객체를 사용하는 언어에서는 객체마다 함수를 정의하고 사용할 수 있다. 객체에서는 가상함수를 통해 함수의 인터페이스의 역할을 하고 각 객체가 할당될 때 함수를 정의한다. 가상함수가 불리게 되면 우선 객체가 가지고 있는 가상함수 테이블을 참조하여 함수의 주소를 알아내고 함수를 실행하게 된다.

가상함수의 실행 과정을 PAC으로 보호하기 위해서는 두 가지를 보호할 필요가 있다. 첫 번째로는 가상함수 테이블의 주소, 두 번째로는 실제로 불리는 함수의 주소가 PAC으로 보호된다. 첫 번째로 보호

```
[0x100002d98]: str    x0, [sp, #0x8]
[0x100002d9c]: ldr    x0, [sp, #0x8]
[0x100002da0]: ldr    x16, [x0]
[0x100002da4]: mov    x8, x0
[0x100002da8]: mov    x17, x8
[0x100002dac]: movk   x17, #0x19c7, lsl #48
[0x100002db0]: autda  x16, x17
[0x100002db4]: ldr    x8, [x16, #0x8]!
[0x100002db8]: mov    x9, x16
[0x100002dbc]: mov    x17, x9
[0x100002dc0]: movk   x17, #0x555d, lsl #48
[0x100002dc4]: blraa  x8, x17
```

Fig. 9. Example of C++ virtual table

되는 가상함수 테이블은 다음과 같은 방식으로 PAC 처리된다.

가상함수의 테이블 주소를 PAC 처리하기 위한 인자로 PAC의 키와 수정자, 해당 주소를 사용하는데, PAC의 키는 DA, 수정자로는 객체의 주소와 객체 고유의 고정된 값을 사용한다. 해당 값은 객체의 이름에 대한 해시값을 사용하며, 컴파일 시에 고정된다. PAC 처리된 포인터는 객체가 생성될 시에 저장된다. 가상함수의 주소는 PAC 키로 IA 키, 수정자로는 맹글링 된 함수 이름에 대한 해시값과 가상함수 테이블들의 주소를 사용하여 PAC의 처리를 수행한다[8].

PAC으로 보호되는 가상함수를 실행할 때는 우선 가상함수의 테이블에 적용되어있는 PAC을 인증하고 그 이후 인증된 테이블 주소를 사용하여 필요한 함수의 주소를 얻는다. 그 이후 얻은 주솟값도 마찬가지로 PAC을 인증한 이후에 실제 함수로 이동한다.

그림 9의 어셈블리는 실제 가상함수를 사용하기 위해 가상함수 테이블 포인터와 실제 함수 두 가지를 PAC 인증하고 사용하는 과정이다. x16 레지스터에 가상함수 테이블이 있다고 가정한다면 6번째 줄에서 가상함수 테이블의 주소와 객체 이름에 따른 해시값을 사용하여 테이블의 포인터를 인증한다. 그 이후 함수를 사용하기 위해 11번째 줄에서 가상함수의 주소와 가상함수 이름에 따른 해시값을 사용하여 함수 포인터를 인증한다.

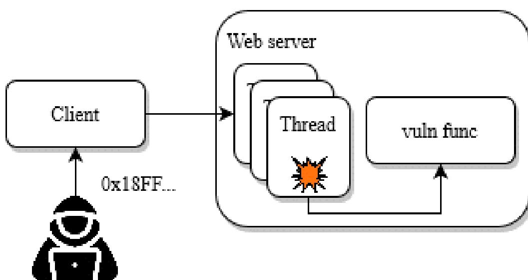

Fig. 10. Brute-force example

IV. PAC에 대한 공격

본 단원에서는 다양한 PAC의 공격 방법 및 공격에 대한 성공 확률에 대한 분석을 진행한다.

4.1 PAC 무차별 대입 공격

본 단원에서는 PAC의 값을 무차별 대입으로 알아내어 원하는 포인터를 사용하는 공격에 관해 서술한다.

4.1.1 PAC 무차별 대입 공격 개요

PAC은 가상주소의 크기와 PAC 이외의 하드웨어의 기능의 여부에 따라 크기가 결정된다. 일반적으로 설정된 PAC의 크기는 7비트이고, 이를 추측하는 데는 2^7 만큼의 개수로 전부 확인할 수 있다. 하지만 PAC을 생성하는 데에 있어 포인터의 주소, 스택의 주소, 상황에 맞는 키 총 3가지의 요소가 존재하며, 매번 PAC이 생성되므로 모든 PAC의 경우를 하나씩 대입하는 것만으로는 공격이 성공하지는 않는다. 하지만 유저 프로세스는 무한정으로 fork()가 가능하므로 같은 키를 가지고 있는 프로세스를 무한정으로 생성할 수 있다. 따라서 무차별 대입 공격을 수행하면 성공할 가능성이 존재한다.

웹서버와 같은 무한정으로 공격을 시도할 수 있는 상황에서는 무차별 대입 공격이 상당히 많이 일어난다. 그리고 이러한 공격은 정상적인 상황에서 생기는 버그 같은 경우와 분간하기 힘들다. 이러한 공격을 막기 위해서는 일반적으로 임계점을 설정하여 시도 횟수를 제한하는 방식으로 방어를 수행한다. 하지만 임계점은 값을 설정하는 데 있어 많은 어려움을 가지고 있다. 본 실험은 무차별 대입 공격이 성공하는 횟수와 시간을 통해 방어를 위한 임계점 설정에 도움을 주어 공격을 막고 정상적인 사용이 가능하도록 도움을 줄 수 있다.

4.1.2 무차별 대입 공격의 확률

유저 프로세스에서의 PAC은 프로세스마다 키가 결정되고, 스레드끼리 같은 키를 공유하기 때문에 PAC의 생성에는 생성하려고 하는 포인터의 주소와 스택 주소에 의해 결정된다. 부모와 자식 프로세스 간 같은 PAuth 키를 사용하기 때문에 PAC의 생성에 대한 엔트로피는 줄어들게 된다. 따라서 특정 포인터의 PAC을 생성하기 위한 확률은 다음과 같다.

$$H_{PAC} = \frac{1-p}{1-2^{-b}} \tag{1}$$

여기서 p는 PAC의 공격이 성공할 수 있는 확률, b는 PAC의 비트 수다. 그리고 한 번의 시도에 PAC을 맞출 수 있는 확률은 다음과 같다.

$$P_{PAC} = \frac{1}{2^b} \tag{2}$$

일반적으로 PAC의 비트 수는 7비트이다. 따라서 PAC의 값이 변경되지 않는 경우, PAC을 공격하는데 총 128회로 공격에 성공할 수 있다.

유저 프로그램에서의 포인터 보호는 PAC의 값을 결정하는 3가지의 요소(포인터의 주소, PAC 키, 수정자) 중 포인터의 주소와 PAC의 키는 고정되어있고 수정자가 변경되게 된다. 이 경우에 PAC 생성 엔트로피는 다음과 같다.

$$H(PAC, modifier) = \Sigma(p(modifier)\log_2(p(modifier))) \tag{3}$$

여기서 p(x)는 PAC 생성에 대한 알고리즘이다. 여기에서 PAC의 크기를 b라고 하였을 시 이를 맞출 수 있는 확률은 위 (2) 식과 같다.

커널에서의 포인터 보호는 유저 환경의 프로그램과는 달리 PAC의 키와 수정자 두 가지 모두 변경되므로 유저의 PAC을 알아내는 확률과는 다르다. 이를 계산하면 다음과 같다.

$$H(PAC, mod, key) = \Sigma(p(mod, key)\log_2(p(mod, key))) \tag{4}$$

4.2 C++ 가상함수 공격

단락 3.5에 따르면, PAC은 C++의 가상함수에 대한 보호를 적용하고 있다. 가상함수의 포인터는 가상함수 테이블 및 가상함수 포인터가 PAC에 의해 보호되고 있으며, 각각 APDA, APIA 키로 보호되어있다. 각자의 수정자는 클래스와 함수의 고유한 해시값으로 고정되어있다[8]. 이 값은 프로그램 내부에 상수로 저장된다. 따라서 공격자가 PAC 서명 명령어를 사용할 수 있다고 가정한다면 공격자는 원하는 함수를 PAC 서명을 통해 실행할 수 있을 것이다.

자세한 공격 방법은 다음과 같다. 공격자는 공격자가 원하는 가상함수 테이블을 생성하고 목표로 하는 객체 가상함수의 고유한 해시값을 알아낸다. 해당 해시값은 객체의 이름 및 가상함수의 이름에서 얻을 수 있으며, 컴파일 시기에 고정된 값을 가진다. 그 후 공격자는 알아낸 해시값을 수정자로 이용하여 원하는 함수 포인터와 테이블을 정상적인 PAC 명령어를 통해 서명한다. 가상함수 테이블과 가상함수는 고정된 해시값을 통해 보호하기 때문에 실행 시기에는 항상 같은 키와 같은 수정자로 PAC 서명을 한다. 따라서 서명된 악의적인 포인터는 정상적인 인증과정을 거쳐 사용할 수 있다. 마지막으로 서명한 가상함수 테이블의 주소 포인터로 덮어씌우면 공격자는 목표 객체의 가상함수를 부르는 것으로 공격자가 원하는 함수를 수행할 수 있다.

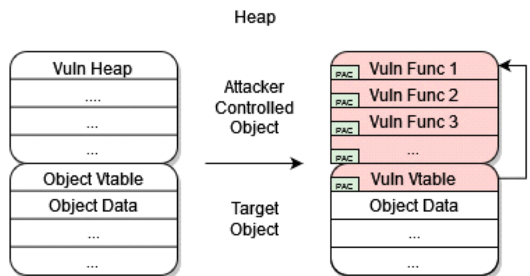

Fig. 11. Attacks on C++ Virtual Table

V. 실험환경

5.1 실험 환경

분석을 진행한 macOS 커널은 xnu-7195-60.75 버전이다. 하드웨어로는 ARMv8.5-A 기반 Apple M1 칩셋을 가지는 Mac mini를 사용하였다.

5.2 무차별 대입 공격 환경

본 실험은 공격자가 PAC이 적용되어있는 유저 프로그램에 무차별 대입으로 공격자가 원하는 주소의 코드를 실행하는 것을 목적으로 한다. 프로그램은 리턴 주소에 임의의 값을 쓸 수 있는 buffer overflow와 같은 메모리 보안 취약점이 있지만 PAC을 사용하여 리턴 주소의 값의 무결성을 보장하고 있다고 가정한다. buffer overflow와 같은 메모리 보안 취약점은 상당히 자주 일어나는 형태의 취약점이며, PAC은 이러한 포인터의 변조에 대한 무결성을 보장해주는 방어 기법으로 해당 취약점으로 공격이 일어나더라도 포인터의 변조를 막을 수 있다. 웹서버와 같이 무한정으로 스레드를 생성할 수 있는 환경에서 해당 메모리 보안 취약점이 존재한다면 공격자는 무한정으로 메모리 보안 취약점을 통해 공격자가 원하는 값의 주소를 리턴 주소에 넣을 수 있다. 하지만 공격자가 임의로 넣은 리턴 주소는 PAC 검증 과정에서 포인터의 인증에 실패하여 실질적으로 임의의 주소를 실행하지는 못한다. 따라서 정상적인 PAC 서명된 값이 존재하는 포인터만 실행할 수 있다.

유저 프로그램에서 PAC으로 서명하는 키는 프로세스 내부의 스레드끼리 공유되어 같은 키를 가지고 서명하며 공격자가 실행을 원하는 함수는 고정되어있다. 따라서 PAC의 값은 수정자에 의해서만 조정된다.

본 실험에서는 공격자가 실행을 원하는 함수의 주소를 알고 있고, 임의로 취약점이 있는 함수를 실행할 수 있다고 가정한다. 그리고 공격자는 PAC 키와 스레드의 스택 생성 위치, PAC의 생성 알고리즘을 알지 못한다고 가정한다. 대상 프로그램은 스레드에 취약점이 있는 함수가 있고, 스레드는 무한정으로 생성 가능하다고 가정한다.

5.3 가상함수 공격 환경

본 실험은 공격자가 PAC이 적용되어있는 프로그램에서 가상함수 테이블을 수정하여 공격자가 원하는 코드를 실행하는 것을 목적으로 한다. 공격자는 힙 배치를 조절할 수 있다고 가정한다. 힙 메모리에 대한 공격에서 이러한 가정은 보편적이다[17]. 그리고 공격자는 코드 재사용과 같은 공격을 활용하여 PAC 서명 명령어를 사용할 수 있다고 가정한다. 이러한 가정은 마지막으로 공격자는 프로그램의 소스 코드를 통해 공격자가 원하는 객체에 대한 정보를 확인할 수 있다고 가정한다.

Table 1. Time and probability of brute-force attack

	Time(sec)	Tries
Average	217.17	73663
Min	0.030032	784
Max	2840.74	359542
Stddev	434.45	66533.92

VI. 실험 결과

6.1 무차별 대입 공격

본 실험에서는 PAC의 값을 결정하는 3가지의 요소(포인터의 주소, PAC 키, 수정자) 중 포인터의 주소와 PAC의 키는 고정되어있고 수정자가 변경되는 상황에서 무차별 대입 공격을 통해 공격자가 원하는 함수를 실행 가능한지, 그리고 공격이 성공되기까지의 횟수와 시간을 알아본다.

무차별 대입 공격은 상황마다 편차가 크게 나오기는 하였으나 평균적으로 217초 이내로 아주 짧은 시간 내로 공격이 성공하는 것을 확인할 수 있었다. 시도 횟수의 평균은 73,663회로 약 287회만큼 전체 PAC을 시도하여 공격에 성공하였다. 최소 횟수는 784회로 상당히 짧은 시도로 성공하였으며, 최대 횟수도 약 1,500회 미만의 시도로 공격에 성공하였다. PAC의 결과가 매번 변경되는 상황에서 이러한 수치는 상당히 적은 횟수로, PAC의 크기가 상당히 작은 부분이 가장 큰 원인이다. 무차별 대입 공격은 어떠한 상황에서도 공격이 가능한 방법인 만큼, 공격을 시도하는 방법이 굉장히 자유로운 방식이다. 그리고 무차별 대입 공격은 상당히 많은 시도를 할 수 있고,

그에 대한 공격이 짧은 시간에 일어나게 된다. 따라서 PAC의 목적인 포인터 보호를 우회할 수 있는 해당 공격은 실질적으로 유용하다. 특히 루트 권한 탈취와 같이 시간과 횟수 상관없이 성공하는 것 자체를 목표로 하는 공격에 대해서 PAC은 상당한 취약점을 가지고 있다고 할 수 있다.

6.2 가상함수 공격

본 실험에서는 PAC으로 보호되는 가상함수의 수정자를 알아내어 공격자가 원하는 함수를 실행하는 공격을 진행하였다. 다음 코드는 가상함수 테이블 생성 과정을 보여준다.

본 예시에서는 x8 레지스터에 '0x9009'라고 하는 객체의 고유한 해시값을 사용하여 PAC 서명의 수정자로 활용하고 있다. 따라서 공격자는 위에서 얻은 해시의 값을 수정자로 활용하여 정상적인 PAC 서명이 된 포인터를 획득할 수 있다. 그 이후 일반적인 가상함수의 공격과 비슷하게 진행할 수 있다. 공격자는 목표 객체의 가상함수 테이블에 서명하였던 포인터를 덮어씌운다. 그 이후 목표 객체의 가상함수를 실행하면 공격자는 원하는 함수를 실행할 수 있다.

```
[0x100002d70]: sub    sp, sp, #0x10
[0x100002d74]: str    x0, [sp, #0x8]
[0x100002d78]: ldr    x0, [sp, #0x8]
[0x100002d7c]: mov    x8, x0
[0x100002d80]: adrp   x16, 2 ; 0x100004000
[0x100002d84]: ldr    x16, [x16, #0xe8]
[0x100002d88]: add    x16, x16, #0x10
[0x100002d8c]: mov    x17, x8
[0x100002d90]: movk   x17, #0x9009, lsl #48
[0x100002d94]: pacda  x16, x17
[0x100002d98]: str    x16, [x0]
[0x100002d9c]: add    sp, sp, #0x10
[0x100002da0]: ret
```

Fig. 12. Creation of C++ Object with Virtual Table

VII. 결 론

우리는 ARM의 새로운 하드웨어 기능인 PAC에 대한 보안성 진단하였다. PAC이 생성되는 과정부터 사용하는 과정 및 보관 및 관리 과정까지 확인하였고, 이를 공격하기 위한 확률 및 횟수를 알아보았다. PAC은 아직은 하드웨어의 한계로 무차별 대입 공격 및 수정자의 유출로 인한 취약점이 있는 PAC 서명된 포인터 생성 등에 취약한 모습을 보인다. 앞으로의 연구를 통해 PAC 시도 횟수 제한 등을 사용하여 더욱 안전한 하드웨어 기반 메모리 보호 기법이 개발되기를 기대한다.

References

〔1〕 Qualcomm, "Pointer Authentication on ARMv8.3 Design and Analysis of the New Software Security Instructions," https://www.qualcomm.com/content/dam/qcomm-martech/dm-assets/documents/pointer-auth-v7.pdf, Feb, 10, 2023.

〔2〕 A. J. Mashtizadeh, A. Bittau, D. Boneh, and D. Mazières, "CCFI," Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 941-951, Oct. 2015

〔3〕 Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan., "PAC it up: towards pointer integrity using ARM pointer authentication.", In Proceedings of the 28th USENIX Conference on Security Symposium (SEC'19), pp. 177-194, Aug. 2019

〔4〕 Project Zero, "Project Zero: Examining Pointer Authentication on the iPhone XS,", https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html, Feb. 01, 2019.

〔5〕 ARM, "Armv8.5-A Memory Tagging Extension.", https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf, Feb, 12, 2023

〔6〕 Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan, "PACMAN: attacking ARM pointer authentication with speculative execution.", In Proceedings of the

49th Annual International Symposium on Computer Architecture (ISCA '22), pp. 685-698, Jun. 2022

〔7〕 GitHub, "ARMv8.3 Pointer Authentication in xnu", https://github.com/apple-oss-distributions/xnu/blob/xnu-7195.60.75/doc/pac.md, Feb. 15, 2023.

〔8〕 GitHub, "Pointer Authentication." https://github.com/apple/llvm-project/blob/next/clang/docs/PointerAuthentication.rst, Feb. 14, 2023.

〔9〕 P. Nasahl, R. Schilling and S. Mangard, "Protecting Indirect Branches Against Fault Attacks Using ARM Pointer Authentication.", 2021 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 68-79, 2021

〔10〕 S. Yoo, J. Park, S. Kim, Y. Kim, and T. Kim, 'In-Kernel Control-Flow Integrity on Commodity OSes using ARM Pointer Authentication', in 31st USENIX Security Symposium, USENIX Security 2022, pp. 89–106, Aug. 2022

〔11〕 A. Fanti, C. Chinea Perez, R. Denis-Courmont, G. Roascio and J.-E. Ekberg, "Toward Register Spilling Security Using LLVM and ARM Pointer Authentication," in IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 41, no. 11, pp. 3757-3766, Nov. 2022

〔12〕 R. M. Farkhani, M. Ahmadi, and L. Lu, 'PTAuth: Temporal Memory Safety via Robust Points-to Authentication', in 30th USENIX Security Symposium, USENIX Security 2021, pp. 1037–1054, Aug. 2021

〔13〕 K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, 'AddressSanitizer: A Fast Address Sanity Checker', in 2012 USENIX Annual Technical Conference, pp. 309–318, Jun. 2012

〔14〕 M. T. Ibn Ziad, M. A. Arroyo, E. Manzhosov and S. Sethumadhavan, "ZeRØ: Zero-Overhead Resilient Operation Under Pointer Integrity Attacks," 2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA), pp. 999-1012, 2021

〔15〕 M. Gallagher et al., 'Morpheus: A Vulnerability-Tolerant Secure Architecture Based on Ensembles of Moving Target Defenses with Churn', in Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, pp. 469–484, 2019

〔16〕 Y. Li et al., 'PACMem: Enforcing Spatial and Temporal Memory Safety via ARM Pointer Authentication', in Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pp. 1901–1915, 2022

〔17〕 Blackhat, "Heap Feng Shui in JavaScript. https://www.blackhat.com/presentations/bh-usa-07/Sotirov/Whitepaper/bh-usa-07-sotirov-WP.pdf, Mar. 16, 2023.

〔18〕 developer.arm.com, "Documentation – Arm Developer", https://developer.arm.com/documentation/ddi0602/2020-12/Shared-Pseudocode/AArch64-Functions?lang=en, Feb, 16, 2023

〈저자소개〉

심 명 규 (Myung-Kyu Sim) 학생회원
2020년 2월: 성균관대학교 컴퓨터공학과 졸업
2021년 3월~현재: 성균관대학교 소프트웨어학과 석사과정
〈관심분야〉 정보보호

이 호 준 (Hojoon Lee) 정회원
2010년 12월: The University of Texas at Austin 졸업
2013년 8월: KAIST 석사
2018년 2월: KAIST 박사
2019년 11월~현재: 성균관대학교 소프트웨어대학 교수
〈관심분야〉 정보보호, 프로그램 분석, 소프트웨어보안, 시스템보안, TEE, 클라우드 AI보안