1. 연구 개요

가. 연구 목적

본 연구는 macOS 기기 사용의 증가와 함께 증가하는 보안 위협에 대응하기 위해, Apple Silicon 기반 시스템의 복잡한 구조를 체계적으로 이해하고, 이를 기반으로 취약점 분석 자동화 방법론을 정립하고자 수행되었다. 특히, 기존 ARM 기반 분석 기법만으로는 Apple의 독자적 가상화 구조를 정확히 파악하기 어려우며, 최신 버전으로 갈수록 보안이 강화되어 자동화된 분석 체계의 필요성이 증대되고 있다.

- ARM 가상화 아키텍처와 Apple Silion 가상화 구조의 차이를 체계적으로 분석
- macOS 고유 보안 체계에 대한 공격 표면 식별
- 정적·동적 분석, 퍼징, 코드 서명 분석 등을 포함한 취약점 분석 자동화 프레임워크 설계
- 실제 제로데이 취약점 발굴을 통한 방법론의 유효성 검증 및 오픈소스 기여

나. 연구 필요성

macOS는 Darwin 기반의 XNU 커널 구조와 SIP, TCC, Gatekeeper, Sandbox profile과 같은 고유한 보안 체계를 갖추고 있으며, Apple Silicon은 ARM 아키텍처를 확장한 독자적 구조를 결합함으로써 다른 플랫폼과는 근본적으로 다른 공격 표면을 형성한다. 이로 인해 기존 ARM 기반 시스템에서 통용되던 분석 기법만으로는 macOS 내부 동작을 정확히 파악하기 어렵고, Windows나 Linux 대비 macOS 보안 연구 생태계가 상대적으로 미성숙한 상황에서 체계적인 분석 방법론 확립이 시급하다.

- macOS 기기의 기업 환경 도입 증가로 인한 표적 공격 위협 확대
- Apple Silicon 고유 구조로 인해 기존 ARM 분석 및 기법의 직접 적용 불가
- XNU 커널, IOKit, XPC 등 macOS 고유 컴포넌트에 대한 공개된 분석 자료 부족
- 수동 분석의 높은 시간 비용 및 반복 작업으로 인한 자동화 체계 필요성 증대

다. 연구 기여

본 연구는 Apple Silicon 기반 macOS 취약점 분석을 위한 체계적 방법론을 제시하고, 이를 실증적으로 검증하여 학술적·실용적 성과를 도출하였다. ARM 가상화 구조와 Apple Silicon 고유 아키텍처에 대한 심층 분석을 바탕으로 분석 프레임워크를 설계하였으며, 실제 제로데이 취약점 발굴과 오픈소스 도구 공개를 통해 macOS 보안 연구 생태계 발전에 기여하였다.

- 총 5건의 Apple 제로데이 취약점 발굴 (CVE-2025-24184, 43338, 43372, 43287, 43401)
- CoreMedia, ImageIO, CoreAnimation 등 macOS 핵심 컴포넌트 취약점 신규 발견
- 퍼저 소스코드 및 공격 표면 식별 도구 오픈소스 공개
- XPC/IOKit 공격 표면 자동 식별, ARM 가상화 취약점, macOS 커널 심볼 이식 기법 관련 논문 게재

2. 연구 방법론

○ 퍼징(Fuzzing) 기반 취약점 탐지

- 프로그램에 대량의 비정상적이고 예상치 못한 입력 데이터를 자동으로 주입하여 오동작, 비정상 종료, 메모리 손상 등의 잠재적 취약점을 발견하는 보안 분석 기법
- 바이너리 퍼징(Binary fuzzing)은 소스코드가 없는 환경에서도 실행 파일 수준에서 취약점을 탐지할 수 있어, 실제 공격 시나리오와 유사항 조건에서 보안 결함을 식별하는데 효과적임

○ 주요 수행 단계

- 입력 변형 (Mutation): 원본 입력 데이터를 기반으로 특정 바이트를 임의의 값으로 변경하거나 길이를 조절하여 프로그램이 예상치 못한 예외 상황을 유발하는 테스트 케이스 생성
- 실행 계측 (Instrumentation)
  - 퍼징 효율을 높이기 위해 프로그램의 실행 경로(Code coverage)를 관찰하고 추적하는 기술
  - 컴파일 시(Compile-time) 소스코드 수정, 바이너리 수정(Binary modification), 또는 실행 시점 동적 계측 방식(Runtime modification)을 통해 새로운 코드 경로 탐색 여부 판단
- 충돌 분석: 반복적인 입력 투입 중 발생하는 충돌(Crash)과 코드 커버리지 변화를 기록하여 실제 보안 위협 여부를 분석

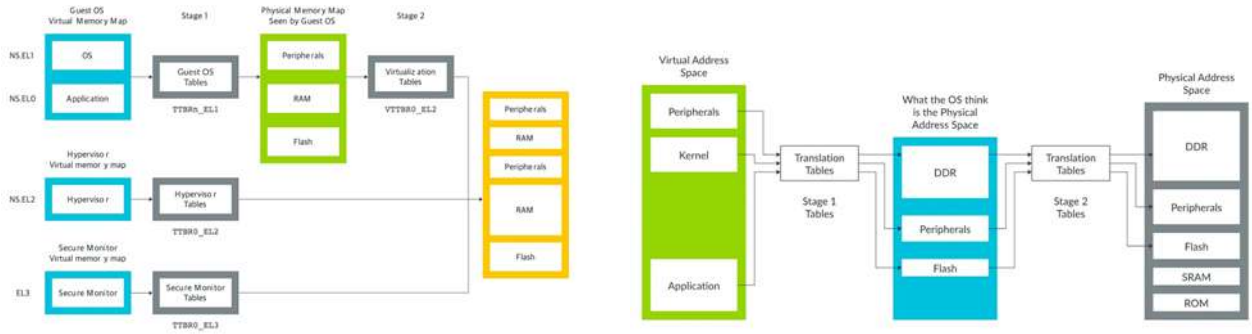
3. ARM과 Apple Silicon 차이

가. ARM 가상화

○ ARM 아키텍처의 메모리 가상화 (Memory Virtualization)

- 모든 VM은 논리적으로 완벽히 격리된 공간에서 구동되어야 함
- VM은 자신에게 할당된 영역 이외의 공간(Hypervisor 또는 다른 VM의 메모리)에 접근할 수 없도록 설계됨

○ 2단계 메모리 변환 (2-Stage Translation)[1][2][3]



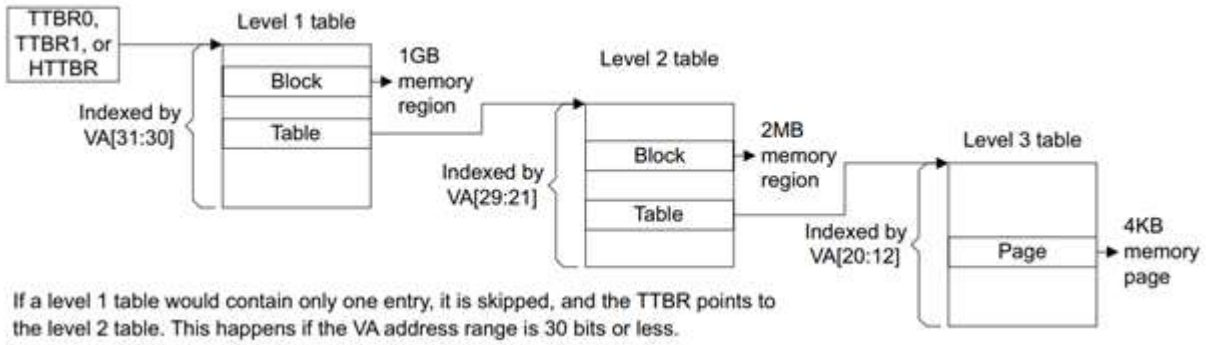
- Stage 1 테이블 (게스트OS 관리)
  - 게스트OS가 자체적으로 페이지 테이블을 관리하며 가상주소를 변환
  - 이 단계에서 변환된 주소 (IPA, Intermediate Physical Address)를 게스트는 실제 물리 메모리로 인식함

- Stage 2 테이블 (Hypervisor 관리)
  - Hypervisor가 관리하며, 게스트가 인식하는 주소(IPA)를 실제 시스템의 물리 주소(PA, Physical Address)로 최종 매핑
  - 각 VM마다 서로 다른 물리 메모리 영역을 매핑함으로써 하드웨어 수준의 메모리 격리를 강제

---

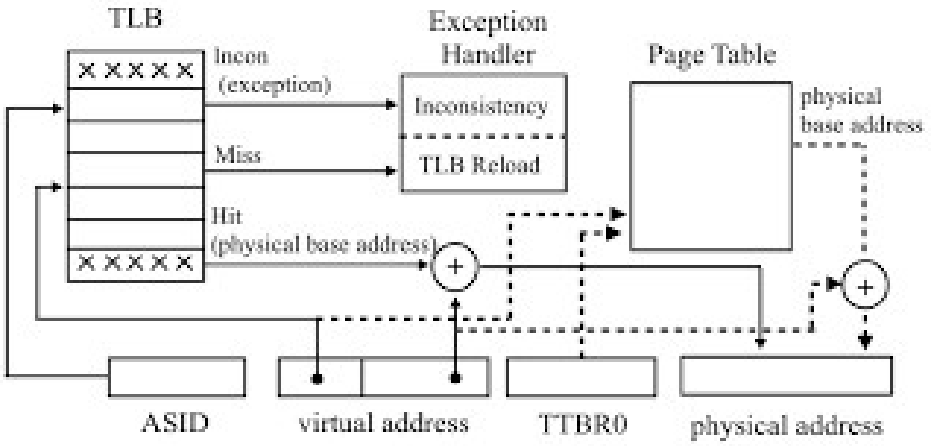
1) https://developer.arm.com/documentation/102142/0100/Stage-2-translation
2) https://developer.arm.com/documentation/101811/latest
3) https://developer.arm.com/documentation/ddi0406/c/System-Level-Architecture/System-Control-Registers-in-a-VMSA-implementation/VMSA-System-control-registers-descriptions—in-register-order/TTBR0--Translation-Table-Base-Register-0--VMSA

○ 페이지 테이블 및 TLB (Translation Lookup Buffer)



- TTBR (Translation Table Base Register)
  - x86 아키텍처의 CR3 레지스터와 동일한 역할을 수행
  - 현재 프로세스의 페이지 테이블 시작 주소를 저장
  - 가상 주소(VA)를 물리 주소(PA)로 변환할 때, 이 레지스터가 가리키는 페이지 테이블을 참조
  - 문맥 교화(Context Switch) 발생 시, 새로운 프로세스의 페이지 테이블을 가리키도록 TTBR 값도 함께 수정됨

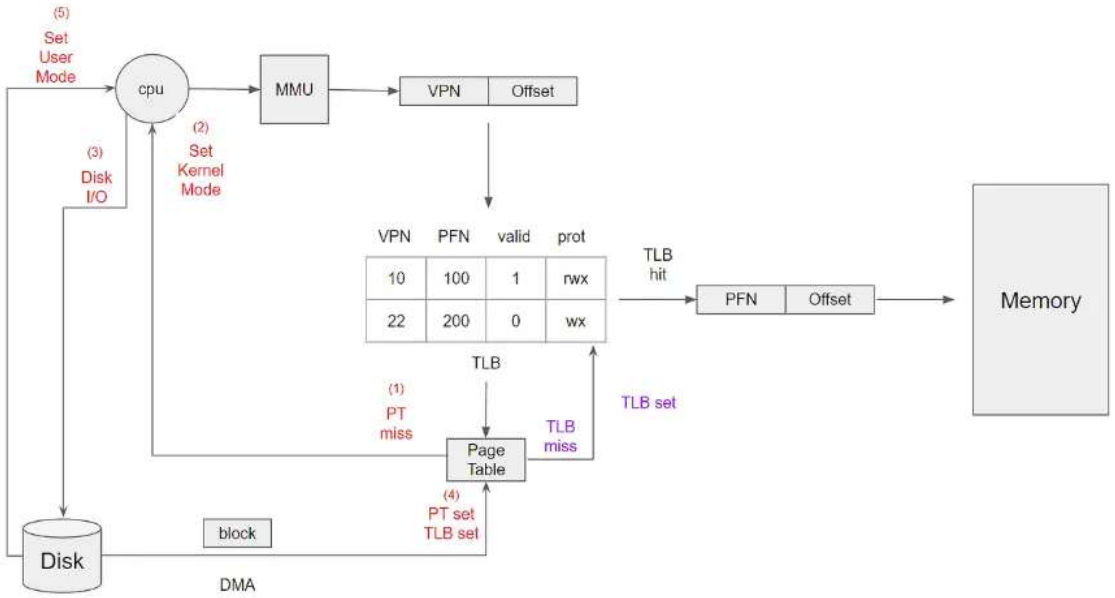
- 메모리 접근 오버헤드와 TLB의 필요성
  - 페이지 테이블을 통한 주소 변환은 '페이지 테이블 접근' + '실제 데이터 접근'의 2단계 과정을 거쳐야 하므로 메모리 접근 비용이 2배로 발생
  - x86과 마찬가지로 ARM64에서도 TLB라는 전용 캐시를 사용하여 자주 사용하는 변환 정보를 저장하여 오버헤드를 감소시킴



- TLB 유지와 ASID
  - TLB는 캐시이므로 문맥 교환 시 이전 프로세스의 주소 변환 정보가 남아있으면 충돌이 발생함. 과거에는 이를 방지하기 위해 문맥 교환마다 TLB를 모두 비워야(Flush) 했으나, 이는 심각한 성능저하를 유발함

- ARM64는 TLB 엔트리에 ASID (Address Space ID) 태그를 추가하여 각 변환 정보가 어떤 프로세스의 것인지 식별
- 문맥 교환이 일어나도 TLB를 비울 필요없이 그대로 유지할 수 있어 시스템 성능이 크게 향상됨
- x86 아키텍처 또한 최근 PCID(Process-Context ID)라는 유사한 개념을 도입하여 동일한 최적화를 수행함

○ Address Space ID와 Virtual Machine ID



- Address Space ID[4)]
  - OS가 각 프로세스의 PID에 부여하는 고유 식별자 (PID의 범위가 크기 때문에, 하드웨어 처리에 적합하도록 압축된 ID를 부여)
  - 문맥 교환시 매번 TLB를 비우는 오버헤드를 줄이기 위해 도입되었음
  - 현재 실행 중인 프로세스의 ASID를 특정 시스템 리지스터에 저장
  - 메모리 접근 시, 하드웨어 레지스터(AA64MMFR0)의 ASID와 TLB 캐시의 ASID를 대조하여 일치하는 경우에만 해당 캐시를 사용

4) https://developer.arm.com/documentation/101811/0105/Address-spaces/Address-Space-Identifiers---Tagging-translations-with-the-owning-process

In practice, it is unlikely that software will use ASIDs differently across processors. It is more common for software to use ASIDs and VMIDs in the same way on all processors in a given system. Therefore, Armv8.2-A introduced the Common not Private (CnP) bit in the Translation Table Base Register (TTBR). When the CnP bit is set, the software promises to use the ASIDs and VMIDs in the same way on all processors, which allows the TLB entries that are created by one processor to be used by another.

- Virtual Machine ID[5)]
  - 가상화 환경에서는 여러 개의 VM이 구동됨. 각 VM 내부의 게스트OS는 독자적으로 ASID를 관리하므로, 서로 다른 VM간에 ASID가 중복될 수 있음.
  - ASID만으로는 구분할 수 없는 다중 VM 환경을 지원하기 위해 VMID를 추가로 사용
  - 최종적으로 TLB는 VMID + ASID 조합을 사용하여 변환 정보를 식별. 이를 통해 Hypervisor는 각 VM의 메모리 공간을 완벽하게 격리하고 관리할 수 있음

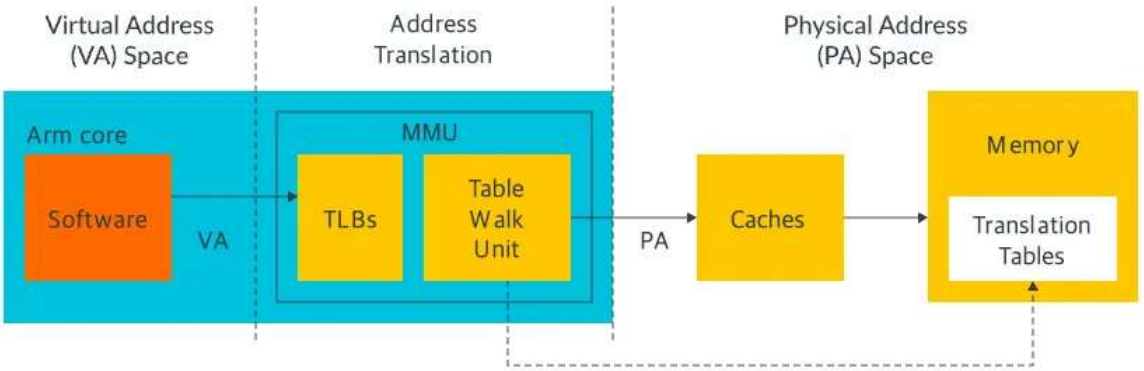
○ TLB 구현 검증 및 무효화 메커니즘

- ASID 비트 검증

```
// ID_AA64MMFR0_EL1.ASIDBits 확인
if (!kvm_has_feat_enum(vcpu->kvm, ID_AA64MMFR0_EL1, ASIDBITS, 16) ||
    !(tcr & TCR_ASID16))
    asid &= GENMASK(7, 0);   // 8비트로 제한
```

  - ARM64 스펙에 정의된 ASID 크기(8비트/16비트) 설정이 실제 Hypervisor 코드에서 정확히 처리되고 있음을 확인
  - Linux 커널의 KVM 소스코드(ID_AA64MMFR0_EL1 레지스터 확인 로직)를 분석한 결과, 하드웨어 기능 플래그(ASIDBITS)와 제어 레지스터(TCR_ASID16)를 체크하여 ASID 길이를 결정하는 로직이 존재함
  - 조건을 만족하지 못할 경우 GENMASK(7, 0)을 사용하여 강제로 8비트로 마스킹 처리함으로써 스펙을 준수하고 있음

5) https://developer.arm.com/documentation/101811/0105/Address-spaces/Common-not-Private



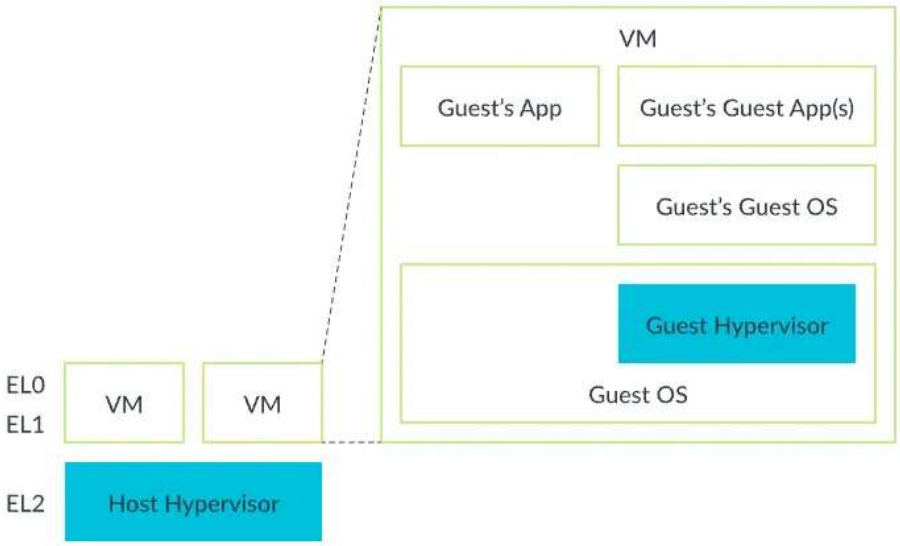
- TLB 엔트리 구조
  - Virtual Address (VA), Physical Address (PA)
  - ASID: 특정 프로세스(non-Global)를 식별하기 위한 ID
  - Memory Attributes: 캐시 가능 여부, 읽기/쓰기 권한 등 속성 정보
  - Translation Regime: 해당 엔트리가 어느 권한 레벨(EL0/1, EL2, EL3)에서 유효한지 명시
  - Global bit (nG): ASID 확인 여부를 결정하는 비트

- TLB 무효화 명령어 (TLBI Instruction)
  - TLBI 명령어: TBLI {Type} {Level} {IS¦OS}
  - Type: 무엇을 삭제할지 결정 (All: 모든 엔트리 삭제, VA: 특정 가상 주소만 삭제, ASID: 특정 프로세스 관련 엔트리만 삭제)
  - Level: 어트 계층의 TLB를 삭제할 것인지 결정 (E1: 게스트OS, E2: Hypervisor 등)
  - Broadcasting (Shareability): IS 옵션이 있으면 해당 명령이 동일한 클러스터 내의 다른 코어들에게도 전파되어 다른 코어 TLB도 동기화됨. OS 옵션은 ARMv8.4부터 추가된 더 넓은 범위의 브로드캐스트

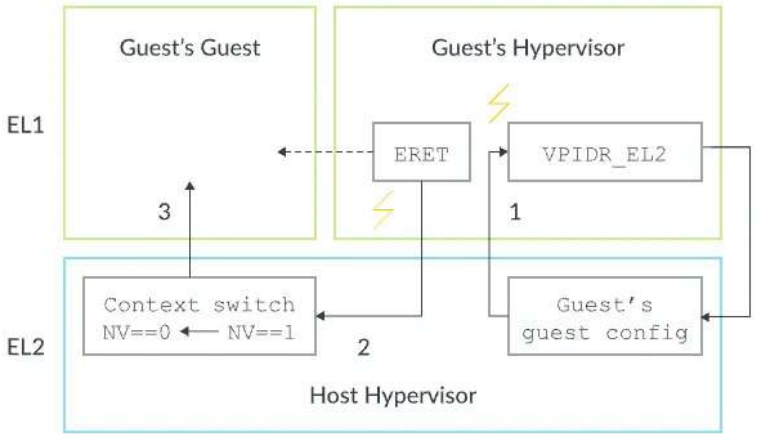
	Global 엔트리 (nG=0)	non-Global 엔트리 (nG=1)
매칭 조건	- VA가 일치하고 - Translation regime이 일치하면 → TLB 히트 (ASID 무관)	- VA가 일치하고 - ASID가 일치하고 - Translation regime이 일치하면 → TLB 히트
사용 사례	커널 매핑, 하이퍼바이저 매핑	사용자 공간 매핑

- Global Bit (nG)와 TLB 히트 조건
  - TLB 엔트리 내 nG 필드는 해당 페이지가 전역적인지, 특정 프로세스 전용인지 구분
  - Global (nG=0): 커널 메모리 영역과 같이 모든 프로세스에서 공통적으로 사용하는 주소. ASID를 검사하지 않고 매칭됨
  - non-Global (nG=1): 사용자 프로세스 영역. VA와 ASID가 모두 일치해야 TLB 히트가 발생

○ 중첩 가상화 (Nested Virtualization)

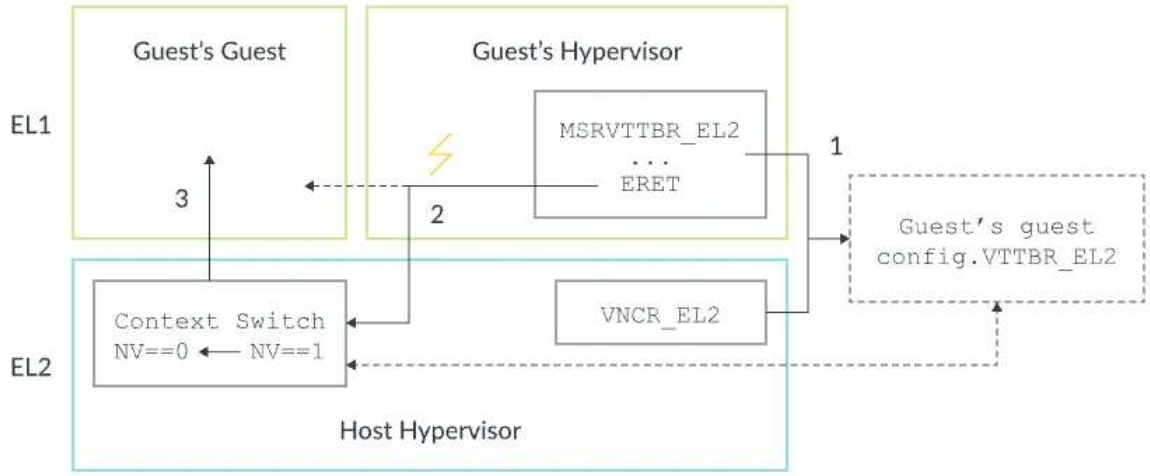


- 중첩 가상화의 격리 원칙
  - 가상머신 내부에서 다시 Hypervisor를 구동하여 또 다른 VM을 실행하는 기술
  - 게스트 Hypervisor가 호스트 Hypervisor에 직접 접근하면 보안 위협이 될 수 있음
  - 게스트 Hypervisor는 실제로 EL1(커널 모드) 권한으로 강등되어 구동
  - 호스트 Hypervisor는 HCR_EL2 레지스터를 사용하여 게스트가 호스트 영역을 침범하지 못하도록 보호



- ARMv8.3 소프트웨어 기반 에뮬레이션 (NV, NV1)
  - 게스트 Hypervisor(EL1)가 Hypervisor용 명령어(EL2 레지스터 접근 등)을 실행하면, 이를 즉시 실행하지 않고 호스트(EL2)로 트랩을 발생시켜 에뮬레이션 함
  - 게스트 Hypervisor가 _EL2 레지스터에 접근하면 EL2로 트랩됨. 호스트는 리 접근 시도록 기록함

- 게스트 Hypervisor가 내부 VM(L2 게스트)를 실행하기 위해 ERET 명령어를 내리면, 다시 호스트로 트랩됨
- 호스트 Hypervisor는 기록해둥 상태(condition)을 기반으로 실제 하드웨어 레지스터를 설정하고, NV 비트를 삭제한 후 L2 게스트를 실행
- 빈번한 트랩과 에뮬레이션으로 인해 심각한 성능 오버헤드가 발생함



- ARMv8.4 하드웨어 가속 (NV2, VNCR)[6]
  - 오버헤드를 해결하기 위해 하드웨어적인 지원(NV2)가 도입됨
  - VNCR (Virtual Nested Control Register): EL2 시스템 레지스터들을 메모리 영역에 매핑하여, 게스트 Hypervisor가 트랩 없이 해당 메모리에 읽기/쓰기를 할 수 있도록 함
  - 게스트 Hypervisor(EL1)가 자신을 EL2로 인식하도록 하면서, 실제 TLB 관리는 VNCR을 통해 별도로 격리된 로직을 수행
  - 이를 통해 성능 저하 없이 안전한 중첩 가상화 환경을 제공할 수 있음

6) https://developer.arm.com/documentation/ddi0601/latest/AArch64-Registers/VNCR-EL2—Virtual-Nested-Control-Register

나. ARM 가상화 취약점 분석

○ KVM 중첩 가상화 취약점

```
static bool kvm_vncr_tlb_lookup(struct kvm_vcpu *vcpu)
{
    struct vncr_tlb *vt = vcpu->arch.vncr_tlb;

    // ... 검증 로직 ...

    if (vt->wr.nG) {  // non-Global 엔트리인 경우
        u64 tcr = vcpu_read_sys_reg(vcpu, TCR_EL2);
        u64 ttbr = ((tcr & TCR_A1) ?
                    vcpu_read_sys_reg(vcpu, TTBR1_EL2) :
                    vcpu_read_sys_reg(vcpu, TTBR0_EL2));
        u16 asid;

        asid = FIELD_GET(TTBR_ASID_MASK, ttbr);
        if (!kvm_has_feat_enum(vcpu->kvm, ID_AA64MMFR0_EL1, ASIDBITS, 16) ||
            !(tcr & TCR_ASID16))
            asid &= GENMASK(7, 0);

        return asid != vt->wr.asid;  // 버그: ASID가 일치하지 않을 때 true 반환!
    }

    return true;
}
```

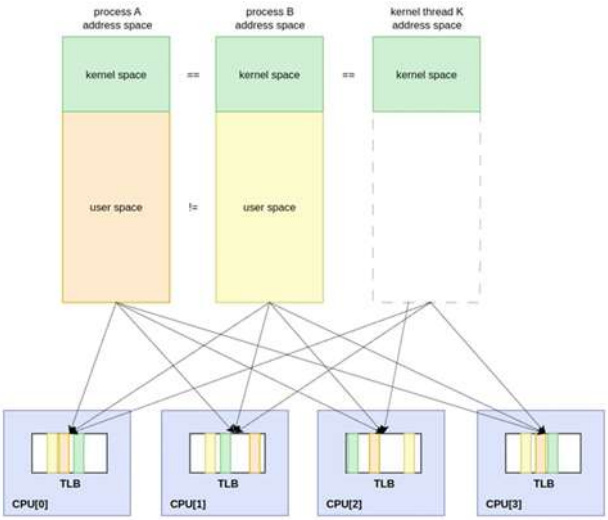
- KVM 취약점
  - VNCR 기반의 소프트웨어 TLB 조회 함수에서 ASID 비교 로직이 반대로 구현된 치명적인 버그를 확인
  - 정상 로직: non-Global(nG=1) 엔트리는 사용자 공간 매핑이므로, 현재 ASID와 캐시된 엔트리의 ASID가 일치해야만 TLB 히트로 판정해야 함
  - 버그: ASID가 일치하지 않는 엉뚱한 엔트리를 유효한 것으로 잘 못 판단하여 반환하는 논리 오류가 존재함

- 공격 시나리오
  - 초기 상태: L1 Hypervisor 상에서 두 개의 L2게스트 (게스트A: ASID 1, 게스트B: ASID 2)가 실행 중
  - 캐싱: 게스트A가 가장 주소 0x1000에 접근하여, VNCR TLB에 {VA:0x1000, PA: 0x5000, ASID:1} 정보가 생성됨
  - 스위칭: 컨텍스트 스위칭 후 게스트B(ASID 2)가 실행됨
  - 취약점: kvm_vncr_tlb_lookup 함수가 실행됨. 현재 ASID(2) != 캐시된 ASID(1)임에도 True 반환
  - 결과: 게스트B는 자신의 메모리가 아닌 게스트A의 물리 메모리 (0x5000)에 접근하게 됨

- 성능 저하 문제
  - 보안 문제뿐만 아니라, 정상적인 상황에서의 시스템 성능도 저하시키는 부작용이 확인됨
  - 현재 ASID와 캐시된 ASID가 일치하는 정상적인 상황에서는 asid != vt->wr.asid가 False가 되어야 함
  - 결과적으로 유효한 TLB 엔트리가 있음에도 불구하고 TLB 미스로 처리됨
  - 불필요한 kvm_translate_vncr() 함수가 호출되어 메모리 접근 지연이 증가할 수 있음

다. 중첩 가상화에서 TLB 제어 메커니즘

◯ 멀티코어 환경에서 TLB 일관성 및 Shootdown



- TLB 무효과와 ASID의 기본 이점 (싱글 코어)
  - TLBI: 변경된 주소 변환 정보를 반영하기 위해 기존 TLB 캐시를 삭제하는 명령어
  - non-Global (nG) 비트가 활성화된 로컬 엔트리에 ASID를 태깅하여 관리
  - 프로세스 A -> B -> A 로 문맥 교환이 발생하더라도, ASID 덕분에 프로세스 A의 캐시가 유지되어 성능이 향상됨

case 1	Process migration process 가 사용하는 메모리 영역의 주소가 바뀌면 virtual address 와 physical address 간의 memory mapping 관계가 변경되었으므로 TLB 를 invalidate 시키기 위해 TLB shootdown 이 발생한다.
case 2	Page table update page table 에 수정 사항이 생기면,(page table entry 를 추가 혹은 제거하거나 page permission 을 변경할 경우) 해당 page entry 에 대응하는 TLB entries 들은 invalidated 혹은 update 되어야 한다.
case 3	Inter-Processor Communication (IPI) 멀티 프로세서 시스템에서 다수의 코어에 데이터가 흩어지다 보니 TLB 동기화가 문제가 대두되었다. 이 시점에 TLB shootdown 이 발생한다. 즉 다수의 프로세서의 TLB 간에 latest memory mapping 및 consistency 를 유지하기 위해 TLB shootdown 이 trigger 된다.

- 멀티코어 환경에서 TLB 일관성 문제[7)]
  - 멀티코더에서는 문맥 교환 시 TLB를 비우지 않으므로 (ASID 사용), 여러 코어에 특정 프로세스의 오래된 TLB 엔트리가 산재할 수 있음
  - 프로세스가 종료되거나 페이지 테이블 매핑(VA <-> PA)이 변경될 때, 해당 프로세스의 TLB 엔트리를 가진 모든 코어가 이를 즉시 무효화해야 함

- TLB Shootdown과 성능 오버헤드
  - 특정 코더가 다른 코어들에게 해당 TLB 엔트리를 비우라고 요청하는 과정
  - 시스템은 IPI (Inter-Processor Interrupt)를 사용하여 다른 코어들에게 인터럽트를 걸고 TLB Flush를 강제함
  - 이 과정에서 IPI 전송 및 처리 대기 시간으로 인해 상당한 오버헤드가 발생함
  - ASID (또는 PCID) 자체를 생성하고 관리하는 데에도 비용이 발생함

- TLB Shootdown과 발생 조건
  - 페이지 권한 변경
  - 메모리 페이지 회수 및 재할당
  - 프로세스 종료 및 주소 공간 해제

7) https://yohda.tistory.com/entry/%EB%A6%AC%EB%88%85%EC%8A%A4-%EC%BB%A4%EB%84%90-Scheduler-TLB

○ 중첩 가상화 TLB 관리 구현

```
static void kvm_invalidate_vncr_ipa(struct kvm *kvm, u64 start, u64 end)
{
        struct kvm_vcpu *vcpu;
        unsigned long i;
        lockdep_assert_held_write(&kvm->mmu_lock);
        if (!kvm_has_feat(kvm, ID_AA64MMFR4_EL1, NV_frac, NV2_ONLY))
                return;
        kvm_for_each_vcpu(i, vcpu, kvm) {
                struct vncr_tlb *vt = vcpu->arch.vncr_tlb;
                u64 ipa_start, ipa_end, ipa_size;
                if (!vt)
                        continue;
                if (!vt->valid)
                        continue;
                ipa_size = ttl_to_size(pgshift_level_to_ttl(vt->wi.pgshift,
                                                            vt->wr.level));
                ipa_start = vt->wr.pa & (ipa_size - 1);
                ipa_end = ipa_start + ipa_size;
                if (ipa_end <= start || ipa_start >= end)
                        continue;
                invalidate_vncr(vt);
        }
}
```

- 물리 주소 기반 무효화 (kvm_invalidate_vncr_ipa)
  - 게스트의 물리주소(IPA) 범위가 변경될 때 관련 TLB를 정리
  - 모든 vCPU의 vncr_tlb 구조체를 순회
  - 현재 캐시된 TLB 엔트리의 IPA 범위를 계산
  - 요청된 범위와 겹치는 엔트리가 존재하면 invalidation_vncr() 호출
  - 호출조건 1: kvm_nested_s2_wp: Nested Stage 2 테이블을 Write-protect 할 때
  - 호출조건 2: kvm_nested_s2_unmap: Nested Stage 2 매핑을 해제할 때

```
struct s1e2_tlbi_scope {
        enum {
                TLBI_ALL,       // 모든 TLB 엔트리
                TLBI_VA,        // VA 기반 + ASID 일치 (nG인 경우)
                TLBI_VAA,       // VA 기반 (ASID 무시)
                TLBI_ASID,      // ASID만으로 매칭
        } type;

        u16 asid;
        u64 va;
        u64 size;
};
```

- 가상 주소 기반 무효화 (invalidate_vncr_va)
  - TLBI 명령어 실행 시 특정 가상 주소 (VA)나 ASID에 해당하는 TLB를 정리
  - scope 인자를 통해 대상을 식별하고 필터링
  - TLBI_ALL: 모든 엔트리 삭제
  - TLBI_VA: 특정 VA 범위 삭제 (Global 페이지는 ASID 무시, nG 페이지는 ASID 일치 확인)
  - TLBI_ASID: 특정 ASID를 가진 모든 엔트리 삭제

○ 중첩 가상화 TLB 관리 취약점

```
// kvm_invalidate_vncr_ipa()
ipa_size = ttl_to_size(pgshift_level_to_ttl(vt->wi.pgshift, vt->wr.level));
ipa_start = vt->wr.pa & (ipa_size - 1);        // 버그: 마스킹 연산
ipa_end = ipa_start + ipa_size;

// invalidate_vncr_va()
va_size = ttl_to_size(pgshift_level_to_ttl(vt->wi.pgshift, vt->wr.level));
va_start = vt->gva & (va_size - 1);             // 버그: 마스킹 연산
va_end = va_start + va_size;
```

- 비트 마스킹 연산 오류
  - TLB 엔트리의 유효 범위를 계산하여 정렬(alignment)하는 과정에서 비트 연산 실수가 발견됨
  - 예를 들어, 크기가 4KB (0x1000)인 페이지에서 물리주소 PA가 0x12345678일 때, 현재 구현에서는 0x12345678 & 0xFFF = 0x00000678이 계산됨
  - 페이지의 시작 주소가 아니라 페이지 내의 오프센이 계산되는 것으로 확인됨
  - 결과적으로 엉뚱한 주소 범위를 계산하게 됨

1. VNCR TLB 엔트리:
   - pa = 0x40123000
   - level = 2 (2MB block)

2. TLBI 명령으로 0x40000000~0x40200000 무효화 요청

3. 버그 동작:
   ipa_start = 0x40123000 & (0x200000 - 1)
             = 0x40123000 & 0x1FFFFF
             = 0x123000 // 잘못된 값!
   ipa_end = 0x123000 + 0x200000 = 0x323000

   검증: if (ipa_end <= 0x40000000 || ipa_start >= 0x40200000)
         if (0x323000 <= 0x40000000) // TRUE!
         → continue; (무효화 건너뜀!)

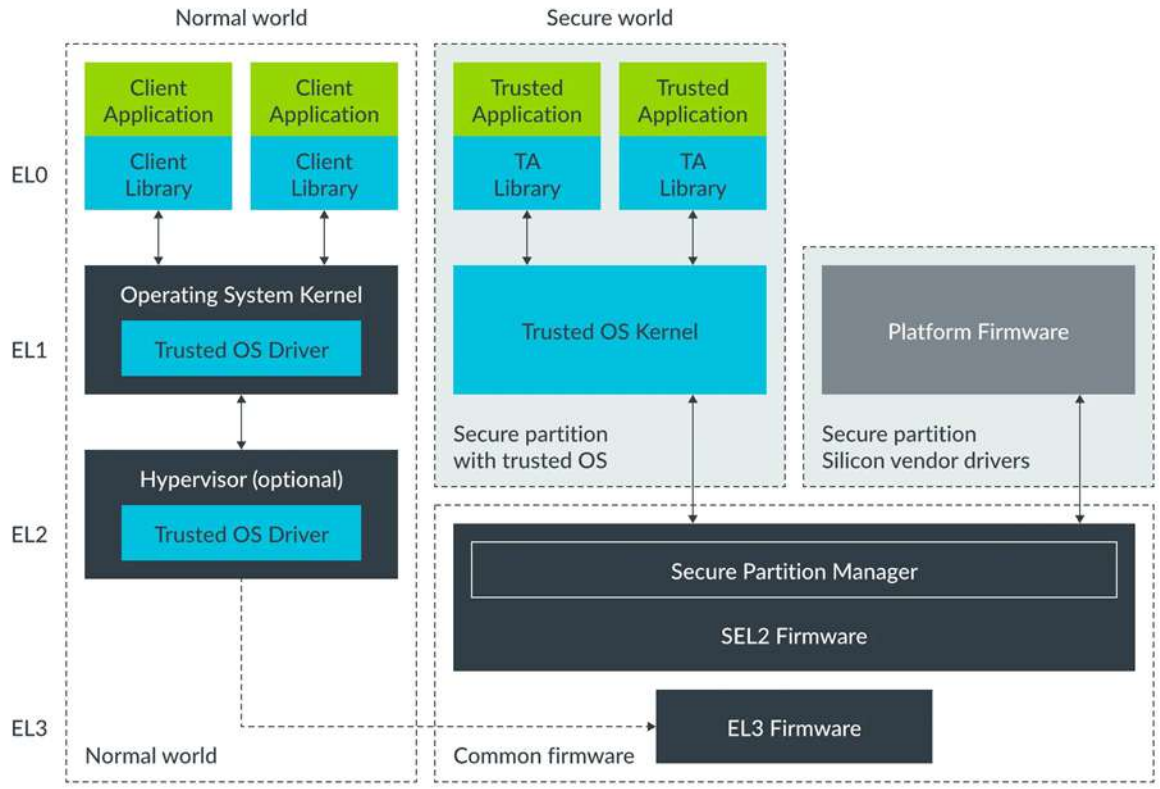
*4. 올바른 동작:
   ipa_start = 0x40123000 & ~0x1FFFFF
             = 0x40123000 & 0xFFE00000
             = 0x40000000 // 올바른 블록 시작!
   ipa_end = 0x40000000 + 0x200000 = 0x40200000

   검증: if (0x40200000 <= 0x40000000 || 0x40000000 >= 0x40200000)
         → FALSE (무효화 실행!)*

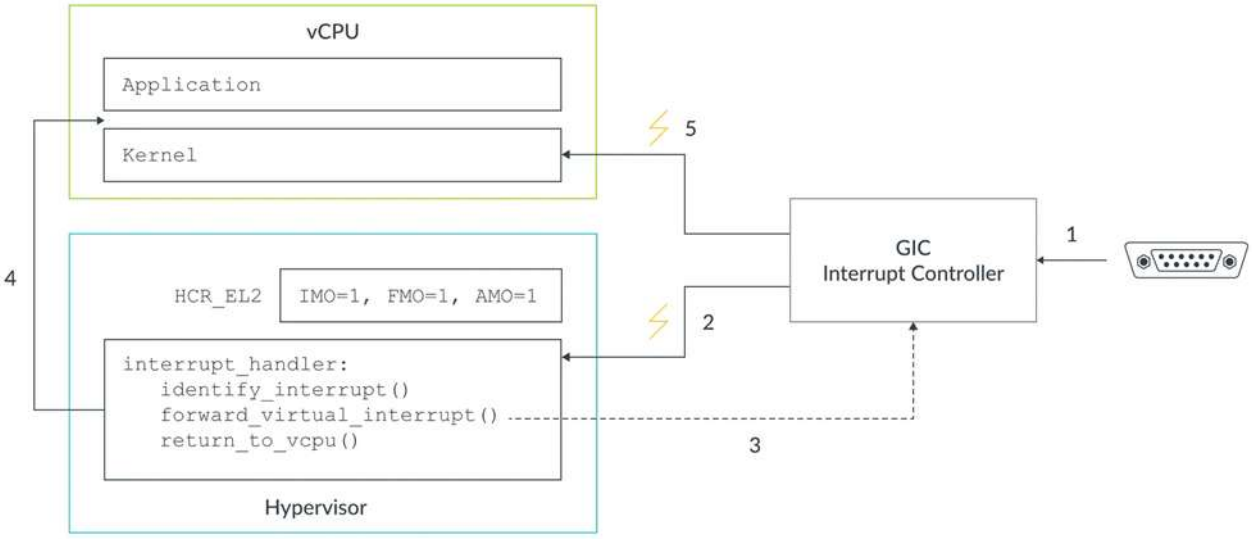
- TLB Stale Entry를 이용한 UAF 공격 시나리오
  - 초기상태: L2 게스트가 가상주소 VA_X를 물리 주소 PA_OLD에 매핑하고 접근 (VNCR TLB 캐싱됨)
  - 매핑 변경: 게스트OS가 페이지 테이블을 수정하여 VA_X를 새로운 주소 PA_NEW로 변경
  - TLBI 요청: 변경 사항을 반영하기 위해 TLBI 명령을 실행하여 기존 캐시 제거를 요청
  - 취약점: KVM 함수 (invalidate_vncr_va) 내부의 마스킹 오류로 인해, 계산된 주소 범위가 실제와 달리 무효화 로직이 실행도지 않고 건너뛰게 됨
  - 잔존 엔트리: VNCR TLB에는 여전히 VA_X -> PA_OLD 정보가 남아있음.
  - 불법 접근: L2 게스트가 VA_X에 다시 접근하면, PA_NEW가 아닌 이미 해제되었거나 다른 용도로 사용 중인 PA_OLD에 접근하게 됨

라. Apple Silicon 가상화

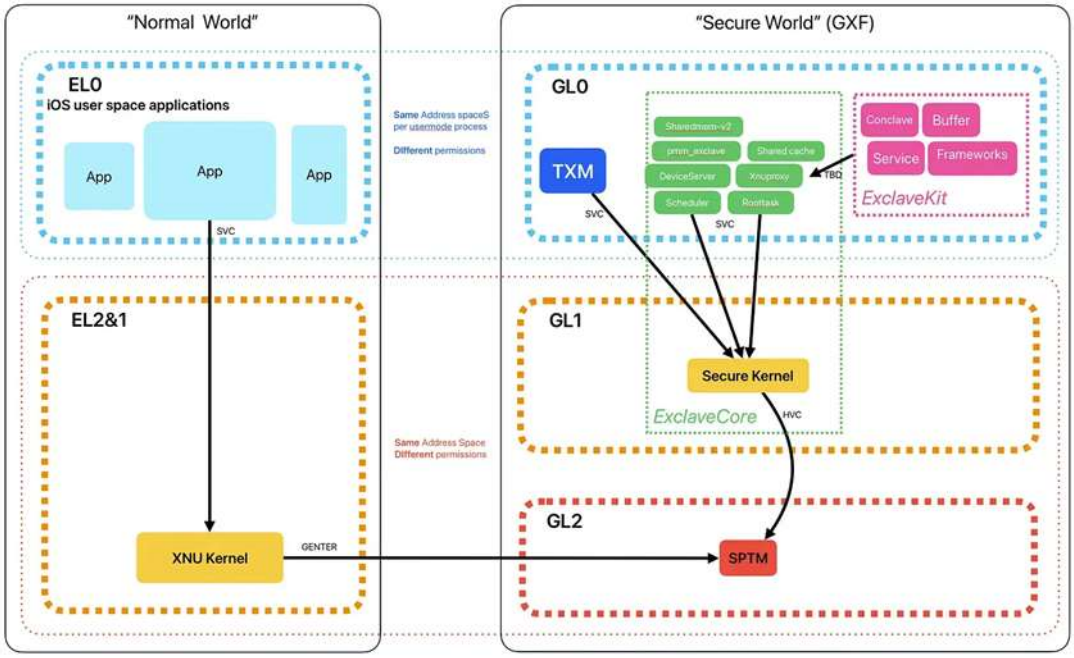
◯ Apple Silicon 보안 아키텍처



- Guarded Level (GL) 기반 가상화
  - 표준 ARM의 EL 구조를 Apple 고유의 가상화 레이어로 감싸, 커널조차 Hypervisor의 완벽한 통제하에 두는 구조
  - 표준 ARM의 EL1(커널)과 EL0(사용자)를 직접 제공하지 않고, 이를 GL1과 GL0라는 가상 환경으로 치환하여 제공
  - 게스트OS(macOS 커널)은 자신이 EL1에서 동작한다고 익식하지만, 실제로는 Hypervisor의 제어 아래 GL1 가상 레벨에서 실행됨
  - 모든 메모리 접근, 인터럽트, 예외 처리가 Hypervisor를 거쳐 가상화된 형태로 공급되므로, 커널이 침해당해도 하위 시스템 (EL2)는 보호됨



- 인터럽트 가상화
  - 물리적인 인터럽트를 Hypervisor가 먼저 가로채고, 게스트에게는 가상 인터럽트만 주입하는 방식으로 처리
  - HCR_EL2 레지스터의 IMO/FMO/AMO 비트를 활성화하여, 물리 인터럽트가 발생하면 EL2(Hypervisor)로 즉시 전달
  - Hypervisor는 ARM GIC (Generic Interrupt Controller)의 Virtual CPU Interface를 활용하여, 게스트에세 가상 인터럽트 (vIRQ, vFIQ 등)을 생성해 전달
  - Apple Silicon은 이 과정을 하드웨어적으로 최적화하여 가상화로 인한 인터럽트 처리 비용을 최소화 함



- Exclave
  - TrustZone을 넘어 Apple만의 격리된 보안 OS 환경으로, 마이크로커널 아키텍처를 기반으로 설계되었음
  - Secure Kernel (SK): L4 계열의 마이크로커널을 기반으로 추정되며, GL1 레벨에서 동작하는 200KB 크기의 초소형 커널
  - SPTM Secure Physical Trusted Monitor(): Exclave 전체를 총괄하는 최상위 제어 레이어. SK는 HVC 호출을 통해 SPTM과 통신하며, CPUTRCE, TXM 등의 서브시스템을 관리
  - 일반 사용자 프로세스는 GL0에서 동작하며, SVC #0 ~#5 시스템콜을 사용하여 커널과 통신
  - roottask: SVC #4, #5를 통해 메시지를 송수신하며, 이벤트 루프를 담당하는 핵심 프로세스

Conclave	역할
com.apple.audiomxd.conclave	마이크/오디오 DSP/Siri 음성 처리
com.apple.cameracaptured.conclave	ISP(Camera pipeline)
com.apple.corespeechd.conclave	Siri trigger
com.apple.kernel	공통 자원 관리

- Conclave 도메인 격리 및 ExclaveOS
  - Exclave 내부는 기능별로 철저히 격리된 Conclave라는 도메인으로 구분되어 운영됨
  - Conclave(기능별 격리 도메인)은 오디오, 카메라, Siri 음성 트리거 등 민감한 기능들이 서로다른 Conclave로 분리되어 있음. 서비스별로 메모리와 IPC가 분리되어 있어, 메인 커널(iOS)이 해킹당하더라도 카메라나 마이크 같은 하드웨어 제어권은 탈취할 수 없음
  - ExclaveOS는 ExclaveCore와 ExclaveKit로 구성됨
  - ExclaveCore는 Secure Kernel과 기본 실행 바이너리를 포함한 핵심 이미지.
  - ExclaveKit는 Exclave 서비스 동작에 필요한 대규모 라이브러리 및 프레임워크 집합.

4. 결론

본 연구는 Apple Silicon 기반 macOS의 복잡한 보안 구조로 인해 기존 분석 기법이 한계에 봉착함에 따라, 이를 극복하기 위한 체계적인 아키텍처 분석과 자동화된 취약점 분석 방법론을 정립하였다. 특히 ARM 가상화 기술과 APple 고유의 하드웨어 확장성을 심층 분석하여 정적·동적 분석 프레임워크를 구축하였으며, 이를 통해 실직적인 공격 표면을 식별하고 효율적인 취약점 발굴 체계를 마련하는 성과를 얻었다.

- 표준 ARM의 가상화 및 메모리 관리 구조와 Apple Silicon의 독자적 확장을 비교 분석하여 취약점 연구의 기술적 기반을 확보함
- macOS 공격 표면 조사 및 퍼징 방법론 설계를 바탕으로 정적 분석 모듈, 코드 서명/권한 추출기, 전용 퍼저, 분석 하네스 자동화 기능을 갖춘 프로토타입 도구를 개발하여 분석 효율성을 극대화함
- 제안된 방법론을 적용하여 CoreMedia, ImageIO, CoreAnimation 등 macOS 핵심 구성요소에서 총 5건의 제로데이 취약점을 발굴함으로써 연구의 실효성을 입증함
- 개발된 퍼저 소스코드와 분석 도구를 오픈소스로 공개하여 생태계에 기여하였으며, XPC/IOKit 공격 표면 식별 및 커널 디버깅 기법 등을 다룬 다수의 학술 논문을 발표하여 관련 연구 분야를 확장함.