Towards Automated Vulnerability Analysis in ARM-based Virtualization

ABSTRACT

This study systematically analyzes the attack surface of ARM-based virtualization in comparison with x86 and proposes a methodology for identifying ARM-specific vulnerabilities. The methodology comprises three stages—extraction of address-translation, coverage-guided fuzzing, and multi-layered detection. In particular, provides reproducible instrumentation procedures and a cross-validation framework for low-level mechanisms such as NV and the TLB, enabling practitioners to rapidly detect and confirm issues in ARM environments. Applied to KVM/arm64, the methodology revealed and reproduced two concrete vulnerabilities: an ASID matching error and a TLB invalidation-range calculation error. We addressed both with small patches and validated the fixes using the proposed observation procedure, demonstrating a practical approach to vulnerability analysis in ARM virtualization.

Keywords : ARM Virtualization, Nested Virtualization, Vulnerability Analysis, Fuzzing

ARM 기반 가상화 취약점 분석 자동화에 관한 연구

요 약

본 연구는 x86 아키텍처와 비교하여 ARM 기반 가상화 환경의 공격 표면을 체계적으로 분석하고, ARM 고유의 취약점을 식별하는 방법론을 제시한다. 제안 방법론은 주소변환 코드 추출, coverage 기반 퍼징 그리고 다층적 탐지의 세 단계로 구성된다. 특히 NV, TLB 등 저수준 메커니즘에 대한 재현 가능한 계측 절차와 교차 검증 프레임워크를 마련하여 실무자가 ARM 환경에서 취약점을 빠르게 식별·확인할 수 있도록 한다. 이를 KVM/arm64에 적용한 결과, ASID 매칭 오류와 TLB 무효화 범위 계산 오류의 두 가지 실제 취약점를 발견·재현하였다. 소규모 패치로 이를 해결하고, 제안한 관측 절차를 통해 수정 효과를 입증함으로써, ARM 가상화 취약점 분석의 실용적 접근법을 제시하고자 한다.

키워드 : ARM 가상화, 중첩 가상화, 취약점 분석, 퍼징

1. 서 론

ARM 기반 프로세서는 클라우드, 엣지, 모바일 환경에서 빠르게 채택되고 있으며, 이에 따라 ARM 아키텍처에서의 가상화 활용도 증가하고 있다[1]. 대표적인 사례로 Amazon의 Graviton 계열, Apple의 M 시리즈 등 다양한 ARM 계열 SoC가 클라우드 인스턴스 및 엔드포인트에 보급된 것이 있다[2-3]. 이러한 플랫폼 전환은 단순한 CPU 아키텍처의 변화에 그치지 않고, 가상화 설계와 실행에서 본질적인 차이를 야기한다. 특히 ARM은 EL0-EL3와 같은 다중 실행 레벨, two-stage 주소 변환, TrustZone, PAC(Pointer Authentication), MTE(Memory Tagging Extension) 등 하드웨어 보안 확장 기능을 제공한다. 이러한 아키텍처적 특징은 가상화 계층의 동작 방식을 x86 환경과 다르게 만들며, 그 결과 기존 x86 기반에서 검증된 보안 모델과 분석 기법이 ARM 환경에 그대로 적용되기 어려운 경우가 발생한다. 실제로 최근 ARM 기반 가상화 환경에서 보고된 취약점 사례들이 증가하고 있어 이에 대한 체계적 분석의 필요성이 대두되고 있다[4-5].

ARM 기반 가상화 취약점은 클라우드의 다수 VM, 엣지 노드의 서비스 구성요소, 모바일 단말의 애플리케이션 등의 광범위한 서비스에 동시다발적 영향을 미칠 수 있다[6].

예를 들어 하나의 호스트에서 다수의 ARM VM을 운영하는 환경에서는 호스트 하이퍼바이저의 취약점이 여러 게스트의 보안에 연쇄적으로 영향을 줄 수 있다. 특히 컨테이너·

논문접수 :
수 정 일 :
심사완료 :
* Corresponding Author : Hong Gil Dong(xxx@xxx.ac.kr)

마이크로서비스 구조에서는 공격 영향 범위가 더욱 확산될 가능성이 크다.

기존 연구는 주로 x86 플랫폼을 중심으로 가상화 보안 문제를 다루어왔고, ARM 환경에 특화된 체계적 취약점 분석 방법론은 아직 부족한 상태이며 이들 취약점은 기존 x86용 도구로는 탐지·분석하기 어려운 특성을 보이기도 한다[7]. 또한 ARM 생태계에서는 VMI(virtual machine introspection) 및 모니터링 도구의 보편화가 x86만큼 이루어지지 않아 동적 분석과 취약점 탐지에 추가적인 어려움이 존재한다[8]. 이로 인해 ARM 환경에서의 보안 분석은 종종 벤더 문서·에뮬레이션 기반 우회에 의존하게 된다는 한계점이 있다.

따라서 본 연구는 이러한 한계점을 해결하기 위해 ARM 기반 가상화 환경에서의 보안 취약점을 체계적으로 분석할 수 있는 자동화된 방법론을 제시한다. 제시된 방법론은 실제 ARM 가상화 취약점을 기반으로 검증되며 이를 통해 ARM 가상화 보안 분석의 실무·학술적 기반을 확장하고자 한다.

2. 배 경

2.1 x86과 ARM 가상화 아키텍처의 차이

x86과 ARM은 설계 철학과 가상화 구현에서 차이를 보인다. 전통적으로 x86은 CISC 기반의 VT-x, AMD-V와 같은 오랜 하드웨어 가상화 지원을 통해 하이퍼바이저·가상머신 모니터(VMM) 생태계가 성숙해 왔다. 반면 ARM은 RISC 설계 원칙을 중심으로 전력 효율성과 설계 단순화에 초점을 맞추며, ARMv8-A 계열부터 가상화 확장과 다중 실행 레벨(EL0–EL3)을 도입하였다[9]. ARM 가상화에서는 하이퍼바이저가 주로 EL2에서 동작하고, 게스트 커널은 EL1, 사용자 코드는 EL0에서 동작한다[10]. 또한 ARM의 two-stage 주소 변환은 가상화된 메모리 매핑을 처리하는 핵심 메커니즘으로, x86의 EPT와 유사한 역할을 수행한다.

이러한 아키텍처 차이는 가상화 계층의 동작·상태 공간을 다르게 형성하며, 결과적으로 하이퍼바이저와 게스트 간 상호작용에서 x86 기반과는 다른 취약점 유형 및 탐지 난이도를 낳는다. 특히 중첩 가상화나 반가상화 구조가 개입하는 경우 가능한 운영 상태 조합이 늘어나며, 비정상 상태의 추적·재현을 위해 더 많은 계측과 반복 실험이 필요해진다.

2.2 GIC와 가상 인터럽트 메커니즘

ARM 플랫폼의 인터럽트 제어는 GIC(Generic Interrupt Controller)에 의해 관리되며, 가상화 환경에서는 vGIC, virtual CPU interface 등 GIC의 가상화 기능을 통해 인터럽트를 게스트로 전달하거나 하이퍼바이저가 개입해 주입한다[11]. GIC의 구성 및 버전(GICv2/ v3/ v4)에 따라 인터럽트 라우팅·가상화 방식이 달라지며, 결과적으로 인터럽트 처리 경로는 가상화 환경에서 보안적으로 민감한 공격 벡터가 될 수 있다. 하이퍼바이저는 물리 디바이스 인터럽트와 게스트용 가상 인터럽트를 분리·매핑해야 하며, 이 과정에서의 미스매치나 취약한 구현이 공격 표면을 제공할 수 있다.

2.3 KVM과 ARM 가상화 구현의 특징

ARM 플랫폼에서 KVM은 하이퍼바이저 역할을 수행하면서 Stage-2 변환 및 MMU 관리, ASID/VMID 관리, 그리고 NV 시퀀스의 책임을 지는 중심 컴포넌트로 동작한다. 특히 VHE(Virtualization Host Extensions)/nVHE 모드의 존재는 KVM의 동작 방식에 중요한 영향을 미친다. nVHE 모드에서는 KVM 하이퍼바이저가 EL2에서 실행되고 커널이 EL1에서 실행되어 지속적인 트랩 오버헤드가 발생하는 반면, VHE는 하드웨어 지원으로 EL1용으로 작성된 커널이 실제로는 EL2에서 실행되도록 하여 소프트웨어적 해결책 없이도 오버헤드를 크게 줄인다.

이러한 모드별 차이점은 KVM이 하드웨어가 제공하지 않는 일부 변환·검증 기능을 소프트웨어로 보강하게 만들며, 복잡한 상태 전파와 정합성 문제를 야기한다. 또한 KVM은 Stage-2 변환을 통해 게스트 메모리 접근을 관리하고, SMMU(IOMMU)와의 조율을 통해 DMA 장치의 메모리 격리를 보장해야 하므로, 가상화된 환경에서의 메모리 일관성 확보가 추가적 과제로 부각된다. 이러한 설계상의 책임 분리와 보정 로직은 작은 논리·연산 실수도 TLB 정합성 또는 주소 공간 격리의 붕괴로 이어질 수 있는 취약 지점을 만들어내며, 실제로 최근 커널 패치 사례에서 VNCR/TLB 무효화 및 ASID 처리와 관련된 결함이 보고된 바 있다. 결과적으로 ARM KVM의 내부 동작을 정확히 재현·관찰하기 위해서는 EL/VHE 모드별 행위, 중첩 가상화 지원, IOMMU 상호작용 등을 포함한 종합적인 계측·검증 프레임워크가 필수적이다.

3. 공격 표면 분석

3.1 메모리 격리·무효화·I/O 경계

(Fig. 1)은 각 아키텍처에서 가상화가 이루어지는 구조를 나타낸 그림이다. 가상화 계층에서의 메모리 격리는 근본적인 보안 경계이며, 아키텍처별로 주소 변환과 정책으로 구현된다. ARM 환경에서는 게스트 OS가 EL1에서 실행되고, 게스트 가상주소는 Stage-1 변환을 거쳐 IPA로, 이어 EL2의 하이퍼바이저가 관리하는 Stage-2 변환을 통해 최종 PA에 도달한다. 이들 번역 정보는 TLB에 캐시되고 ASID/VMID로 분리되는데, 무효화 범위나 순서의 누락, 또는 ASID/VMID의 부적절한 재사용은 캐시된 번역이 남아 있게 하여 stale mapping을 유발할 수 있다.

그 결과 VM 간 또는 VM과 호스트 간 메모리 침범, 잘못된 권한 판단, 전체 시스템 가용성 저해와 같은 심각한 보안 문제가 발생할 수 있다. 또한 디바이스가 DMA로 물리 메모리에 접근하는 경로를 관리하는 IOMMU(SMMU)의 설정·도메인 분리 오류는 가상화 경계를 우회하는 직접적 권한 상승 루트를 제공한다. MMIO 핸들러나 하이퍼콜을 통해

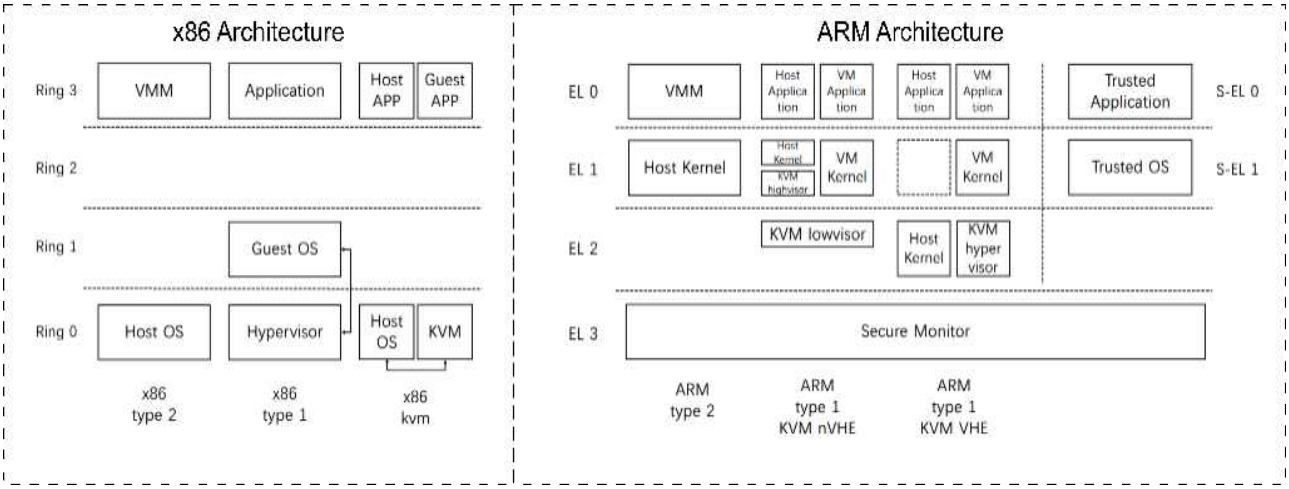

Fig. 1. Comparison of x86 Architecture and ARM Architecture in Virtualization

게스트의 액션이 하이퍼바이저로 전달되는 경로 역시 검증 누락 시 공격 표면으로 작용한다.

3.2 ARM 특유의 권한·인터럽트·타이머 경계

(Fig. 1)에서 나타난 바와 같이, ARM 아키텍처는 EL0에서 EL3에 이르는 다중 실행 레벨로 이루어져 있다. 또한 HCR_EL2, SCTLR_EL2 등 광범위한 시스템 레지스터들을 통해 예외와 트랩의 처리 경로를 세밀히 제어한다. 이로 인해 트랩 분류 및 에뮬레이션 로직의 오류는 잘못된 권한 전이로 이어져 게스트가 의도치 않게 상위 권한 동작을 수행하거나 특권 경로를 남용할 수 있는 위험이 있다. 특히 GICv3/v4 기반의 가상 인터럽트 라우팅과 ITS(Interrupt Translation Service)는 APIC 기반의 x86보다 구조가 복잡하여 라우팅 테이블의 불일치나 이벤트 처리 누락이 발생하기 쉽고, 이는 인터럽트 관련 상태 불일치 및 권한 오용의 원인이 된다. 더불어 컨텍스트 스위칭 시 다양한 하드웨어 레지스터를 저장/복원하는 과정에서 상태 관리에 실패하거나, CNTV, CNTP와 같은 가상 타이머의 동기화 실패는 DoS나 호스트 데이터 유출, 또는 타이밍 기반 공격으로 이어질 수 있다.

3.3 중첩 환경에서의 상태 불일치

(Fig. 2)의 KVM 구조와 같이 중첩 가상화 환경에서는 guest-hypervisor와 host-hypervisor가 각기 Stage-2 번역, VMID/ASID 관리, 가상 GIC 상태를 동시에 유지하므로 상태 일관성 유지가 훨씬 더 복잡해진다. Guest가 관리하는 Nested Stage-2 테이블과 Host가 유지하는 쉐도우 상태 사이에 불일치가 발생하거나, TLB flush 문제로 다른 메모리에 접근할 수 있으며, Guest의 vGIC 상태와 Host의 shadow state 간의 충돌로 잘못된 인터럽트 라우팅이 발생할 수 있다. 또한 예외가 발생했을 때 어느 수준에서 이를 처리해야 하는지를 잘못 판단하면 예외가 의도하지 않은 하이퍼바이저 수준에서 처리되어 권한 체인이 뒤틀리거나 잘못된 자원 접근이 허용될 위험이 있다. 중첩 환경에서의 VMID/ASID 재사용은 특히 정보 누출을 초래할 수 있으므로, 두 수준의 이벤트를 동기화하여 기록하고 비교 검증하는 방식의 계측이 재현성에 필수적이다.

4. 분석 방법

본 절에서는 ARM64 KVM의 주소 변환 메커니즘을 대상으로 한 자동화된 취약점 분석 방법론을 제시한다. 해당 방법론은 실제 커널 코드를 사례로 하여 사용자 공간에서 실행 가능한 퍼징 하네스를 구축하고, coverage-guided fuzzing과 패턴 기반 버그 탐지를 결합하는 것이다.

4.1 커널 코드 추출 및 재구성

분석의 첫 단계는 ARM64 KVM의 2-Stage 주소 변환 코드를 커널에서 추출하여 독립적으로 실행 가능한 형태로 재구성하는 것이다. 본 연구에서는 linux 커널의 /arch/arm64 아래 kvm/hyp/pgtable.c에서 핵심 함수들을 추출하였으며, 특히 BBM(Break-Before-Make) 메커니즘을 포함한 PTE 조작 로직을 완전히 보존하였다. 추출한 핵심 함수는 다음과 같다. stage2_try_break_pte()는 ARM 아키텍처의 BBM 요구사항을 구현한 것으로, 유효한 PTE를 수정할 때 반드시 Invalid 상태로 전환한 후 TLB를 무효화하고, 이후 새로운 값을 설정하는 순서를 보장한다. 이 함수는 Lock 상태 확인, PTE LOCKED 설정, TLB 무효화, Refcount 감소의 4단계로 구성되며, 이 순서가 어긋나면 stale TLB entry로 인한 보안 문제가 발생한다. stage2_make_pte()는 새 PTE를 설정하는 함수로, 기존 PTE가 유효했다면 반드시 LOCKED 상태여야 한다는 BBM 검증을 수행한다. 이 검증 지점에서 위반이 발견되면 g_locked_pte_violations 카운터를 증가시켜 버그를 기록한다. 실제 매핑 로직을 담당하는 stage2_map_walker_try_leaf()는 물리 주소 계산, 새 PTE

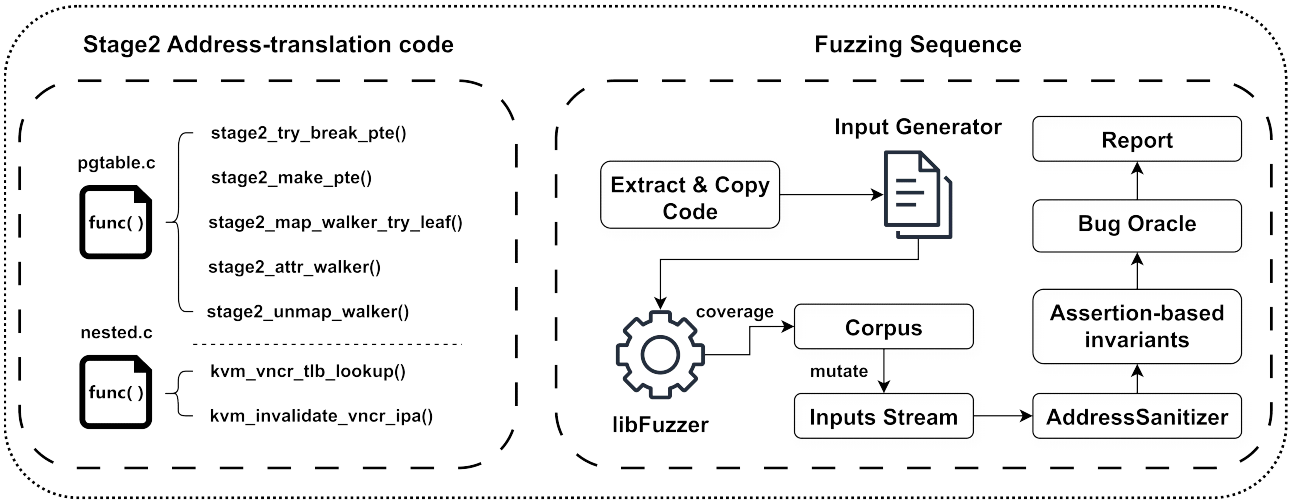

Fig. 2. Analysis sequence of fuzzer

생성, BBM 수행, 새 PTE 설치의 순서로 동작한다. Unmap을 담당하는 stage2_unmap_walker()는 Invalid PTE 처리, PTE 제거, TLB 무효화, Refcount 감소를 수행한다. 권한 변경을 담당하는 stage2_attr_walker()는 PTE의 속성 비트를 원자적으로 변경한다.

코드 추출 시 간소화한 부분은 다음과 같다. 전체 페이지 테이블 트리 순회는 제거하고 512개 PTE 배열에 대한 직접 인덱싱으로 대체하였으며, 커널 spinlock/rwlock 등 동기화 메커니즘을 제거하였다. 이러한 간소화는 퍼징 환경에서 단일 스레드 실행을 가정하기 때문에 안전하다. 그러나 BBM 로직, Refcount 관리, TLB invalidation 호출 순서, 그리고 WARN_ON 기반 assertion은 완전히 보존하여 커널의 핵심 불변 조건이 유지되도록 하였다.

이는 코드 수준의 fidelity를 보장하며, 퍼저가 탐지한 문제가 실제 커널에서도 의미를 갖도록 한다. 중첩 가상화 분석을 위해 linux 커널의 arch/arm64 아래의 kvm/nested.c에서도 핵심 함수를 추출하였다. kvm_vncr_tlb_lookup()는 ASID 비교 로직을 구현하며, 8-bit와 16-bit ASID 처리를 모두 지원한다. kvm_invalidate_vncr_ipa()는 IPA 범위 무효화를 담당하며, IPA 주소 정렬 연산이 핵심이다. 이들 함수는 중첩 가상화의 복잡한 상태 전이를 처리하며, 비트 연산과 조건 분기가 많아 잠재적 버그 발생 가능성이 높은 영역이다. 본 연구에서는 패치 전후 버전을 모두 테스트할 수 있도록 구성하여, 방법론의 유효성을 검증하였다.

4.2 구조화된 입력 생성 및 다층적 버그 탐지

본 연구에서는 LibFuzzer를 퍼징 엔진으로 채택하였다. LibFuzzer는 coverage-guided fuzzing을 지원하여 새로운 코드 경로를 발견한 입력을 자동으로 corpus에 저장하고 이를 변형하여 점진적으로 코드 커버리지를 확대한다. 그러나 ARM KVM 주소 변환 메커니즘을 효과적으로 테스트하기 위해서는 단순한 바이트 스트림이 아닌 구조화된 입력이 필요하다. 구조화된 입력 생성 방식은 단순 바이트 변형보다 복잡한 상태 전이와 연산 간 상호작용을 효과적으로 탐색할 수 있도록 한다. 특히 1-8개의 연속된 연산 시퀀스를 생성함으로써 단일 연산으로는 발견할 수 없는 복합적인 버그 패턴을 유도할 수 있다.

다음으로는 세 가지 계층의 버그 탐지 메커니즘을 통합하여 다양한 유형의 취약점을 자동으로 식별한다. 먼저, AddressSanitizer(ASAN)를 통한 메모리 안전성 검증이다. 퍼징 하네스는 -fsanitize=fuzzer,address,undefined 플래그로 컴파일되어 Use-After-Free, heap buffer overflow, stack buffer overflow, memory leak 등을 실행 시점에 자동으로 탐지한다. ASAN은 메모리 접근을 계측하여 위반 발생 즉시 정확한 stack trace와 함께 crash 파일을 생성하므로, 버그 재현과 분석이 용이하다. 이는 특히 Refcount 관리 오류나 PTE 접근 범위 위반 등을 효과적으로 포착한다.

두번째로, assertion 기반 불변 조건 검증이다. 커널의 WARN_ON 매크로를 전역 카운터로 변환하여 실행 중 위반을 기록한다. 예를 들어 stage2_make_pte()에서는 기존 PTE가 유효했다면 반드시 LOCKED 상태여야 한다는 BBM 조건을 검증하며, 위반 시 g_locked_pte_violations 카운터를 증가시킨다. 마찬가지로 Refcount 관리에서는 mock_put_page()가 이미 0인 refcount를 감소시키려는 시도를 g_refcount_bugs로 기록한다. 이러한 assertion 기반 탐지는 메모리 corruption으로 즉시 드러나지 않지만 논리적으로 잘못된 상태를 포착한다.

세번째로, Bug Oracle을 통한 패턴 기반 탐지이다. bug_oracle.c는 최대 1024개의 연산 히스토리를 추적하며, 각 연산의 주소, 타입, 이전/이후 PTE 값, TLB flush 여부를 기록한다. oracle_check_patterns() 함수는 세 가지 주요 패턴을 탐지한다. 먼저 유효한 PTE를 5회 이상 연속으로 수정하면서 중간에 TLB flush가 없는 경우 stale TLB entry 위험이 있으므로 g_tlb_coherency_bugs를 증가시킨다. 다음으로 같은 페이지를 짧은 시간 내에 2번 이상 수정

```
KVM: arm64: nv: fix VNCR TLB ASID match logic for non-Global entries
--- a/arch/arm64/kvm/nested.c
+++ b/arch/arm64/kvm/nested.c
@@ -1276,7 +1276,7 @@ static bool kvm_vncr_tlb_lookup(struct kvm_vcpu *vcpu)
                    !(tcr & TCR_ASID16))
                        asid &= GENMASK(7, 0);

-               return asid != vt->wr.asid;
+               return asid == vt->wr.asid;
        }

        return true;
```

Fig. 3. nested.c code commit 1

하면서 TLB flush가 누락된 경우를 탐지한다. 마지막으로 PTE의 S2AP 비트가 Read-only에서 Read-Write로 변경되는데 TLB flush가 없으면, 일부 CPU 코어가 여전히 이전 권한으로 인식하는 race condition이 발생할 수 있다.

5. 취약점 분석

본 장에서는 제안한 방법론을 적용하여 ARM64 KVM의 중첩 가상화 경로에서 확인된 두 가지 취약점을 분석한다. 첫 번째는 VNCR TLB 조회 시 ASID 매칭 논리의 오류이며, 두 번째는 VNCR 엔트리 무효화 범위 계산의 오류이다. 두 취약점 모두 작은 연산자나 비트마스크 착오에서 비롯되었으나, 가상화 환경의 격리 보장과 캐시 정합성에 치명적 영향을 미칠 수 있다.

5.1 VNCR TLB ASID 매칭 오류

중첩 가상화 경로에서 동작하는 kvm_vncr_tlb_lookup() 함수는 캐시된 VNCR TLB 엔트리가 현재 실행 컨텍스트에 유효한지를 판별하는 역할을 수행한다. non-Global 엔트리의 경우 캐시 엔트리의 ASID가 현재 컨텍스트의 ASID와 일치할 때에만 재사용해야 함에도 불구하고, 기존 구현은 이 판정을 역전하여 ASID가 불일치할 때 캐시를 유효하다고 판단하였다. 이로 인해 유효한 캐시가 무시되어 불필요한 재번역 경로로 진입하는 성능 저하 현상이 발생했고, 반대로 다른 ASID의 엔트리가 잘못 재사용되어 주소 공간 격리가 붕괴될 가능성이 확인되었다. 정밀 관찰을 통해 ASID 전환 직후 동일 가상주소에 대한 접근에서 kvm_vncr_tlb_lookup의 반환값과 실제 물리 접근 경로가 일치하지 않는 현상이 반복적으로 포착되었고, (Fig. 3)과 같이 이러한 증거를 바탕으로 조건식의 역전이 문제의 중심임을 규명할 수 있었다.

5.2 VNCR 무효화 범위 계산 오류

VNCR 무효화 경로의 함수들, 예컨대 kvm_invalidate_vncr_ipa()와 invalidate_vncr_va()에서는 지정된 주소 범위를 블록 크기에 맞추어 내림 정렬한 후 무효화를 수행해야 한다. 그러나 (Fig. 4)와 같이 소스코드에는 블록 시작 정렬을 위해 addr & ~(size - 1)를 사용해야 할 자리에 addr & (size - 1)가 적용되어, 주소의 하위 비

```
KVM: arm64: nv: Fix incorrect VNCR invalidation range calculation
--- a/arch/arm64/kvm/nested.c
+++ b/arch/arm64/kvm/nested.c
@@ -847,7 +847,7 @@ static void kvm_invalidate_vncr_ipa(struct kvm *kvm, u64 start, u64 end)

                ipa_size = ttl_to_size(pgshift_level_to_ttl(vt->wi.pgshift,
                                                            vt->wr.level));
-               ipa_start = vt->wr.pa & (ipa_size - 1);
+               ipa_start = vt->wr.pa & ~(ipa_size - 1);
                ipa_end = ipa_start + ipa_size;

                if (ipa_end <= start || ipa_start >= end)
@@ -887,7 +887,7 @@ static void invalidate_vncr_va(struct kvm *kvm,

                va_size = ttl_to_size(pgshift_level_to_ttl(vt->wi.pgshift,
                                                           vt->wr.level));
-               va_start = vt->gva & (va_size - 1);
+               va_start = vt->gva & ~(va_size - 1);
                va_end = va_start + va_size;
```

Fig. 4. nested.c code commit 2

트만 추출함으로써 시작 주소를 잘못 계산하고 무효화 범위를 축소시키는 오류가 존재했다. 이로 인해 TLBI나 관련 MMU 알림 이후에도 경계에 걸친 일부 엔트리가 stale 상태로 남아 접근 시 이전 번역을 참조하는 현상이 관찰되었으며, ftrace와 SMMU 로그의 정합 검사에서 무효화 호출 후에도 특정 VNCR 엔트리가 제거되지 않는 패턴이 반복적으로 드러났다.

5.3 근본 원인 분석

두 취약점은 표면적으로는 서로 다른 결함으로 보이지만 근본적으로는 ARM 가상화의 태깅 및 무효화 메커니즘이 지닌 민감성에서 기인한다. ARM의 two-stage 번역과 태그 기반 TLB 설계는 작은 논리·비트 연산 실수라도 캐시 정합성이나 주소 공간 격리의 붕괴로 이어질 수 있는 구조적 특성을 갖고 있다. VNCR TLB ASID 매칭 오류의 경우에는 단순한 조건식의 역전이 직접적인 원인으로 확인되었으며, 이는 ASID 처리의 경계 케이스에 대한 검증이 부족했음을 의미한다. VNCR 무효화 범위 오류는 비트마스크 연산의 부정확성에서 기인하였고, 특히 블록 경계 교차 상황을 포함하는 테스트와 무효화 직후 상태를 확인할 수 있는 관찰 체계가 부족했던 점이 문제의 조기 발견을 방해했다. 공통적으로 두 사례는 중첩 가상화에서의 복잡한 상태 전파와 무효화의 원자성, 그리고 하드웨어-소프트웨어 경계에서의 보정 로직(software PTW 등)에 대한 정밀 검증 부족이 결합되어 결함이 심각한 영향으로 증폭된 결과이다.

5.4 패치 및 검증

본 연구에서 제시한 정밀 관찰 파이프라인을 통해 패치 전후의 동작을 교차검증하였다. VNCR TLB ASID 매칭 오류에 대해서는 kvm_vncr_tlb_lookup() 내의 ASID 비교 조건을 등가 판정으로 변경하여 ASID가 일치할 때만 캐시 히트를 허용하도록 수정하였고, 이 변경 이후 동일 재현 시나리오에서 ASID 불일치 시 캐시 미스가 일관되게 발생하며 즉시 올바른 재번역 경로로 진입하는 것을 확인하였다. 성능 지표 측정에서는 패치로 인한 유의미한 성능 저하가 관찰되지 않았고, 불필요한 재번역 경로로의 진입이 감소함으로써 일부 TLB miss 관련 지표가 개선되었다. VNCR 무효

화 범위 계산 오류에 대해서는 주소 정렬식을 올바른 비트 마스크로 변경하여 무효화 시작 주소를 정상화하였고, 그 결과 경계 교차 접근에서 관찰되던 stale 히트가 소거되었으며 무효화 직후의 접근이 항상 재번역 경로로 진입함을 반복 실험으로 확인했다. 두 패치 모두 메일링 리스트에 보고되어 리뷰를 거쳐 병합되었으며, 본 연구의 자동화된 관찰 파이프라인은 패치의 효과를 정량적으로 입증하는 데 기여하였다.

6. 결 론

본 논문은 ARM 기반 가상화 환경에서의 공격 표면을 x86과 체계적으로 비교·분석하고, ARM 특유의 Exception Layer, 주소변환, 태깅, 인터럽트 경계를 중심으로한 구조 재매핑과 주소 변환 위험 분석 및 정밀 관찰·검증으로 구성된 실용적 분석 방법론을 제시하였다. 제안 방법론을 KVM/arm64에 적용한 결과, 중첩 가상화 경로에서 VNCR TLB ASID 매칭 논리 오류와 VNCR 정렬 계산 오류라는 두 가지 결함을 발견·재현하고, 단일 라인 수준의 수정으로 정합성과 격리를 회복함을 실험적으로 확인하였다. 이는 ARM 가상화에서 태깅 정합성과 무효화 시퀀스의 정확성이 보안과 신뢰성의 핵심 축임을 실증적으로 보여준다.

향후 연구에서는 Apple Silicon HVF와 같은 폐쇄적 생태계의 가시성 제약, 다양한 SoC·커널 조합에 대한 광범위한 커버리지라는 제약을 극복하고 PAC/BTI 등 아키텍처 보안 확장을 고려한 관찰 포인트 확장, SMMUv3·가상 GIC의 중첩 경로에 대한 자동 정합성 검사, HVF 등 블랙박스 플랫폼을 대상으로 한 사용자 공간 기반의 경량 추적 기법을 고도화하여, 탐지·재현·패치 검증의 전주기를 더 자동화·일반화한다.

References

[1] M. Paolino et al., "Building Trusted and Real Time ARM Guests Execution Environments for Mixed Criticality," International Journal on Advances in Security, Vol.8, No.3&4, 2015.

[2] Amazon Web Services, "AWS Graviton Processors" [Internet], https://aws.amazon.com/ko/ec2/graviton/

[3] Apple Security Research, "Private Cloud Compute: A New Frontier for AI Privacy in the Cloud" [Internet], June 10, 2024, https://security.apple.com/blog/private-cloud-compute/

[4] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: Attacking ARM Pointer Authentication with Speculative Execution," In Proceedings of the 49th Annual International Symposium on Computer Architecture, pp.685-698, 2022.

[5] H. Jang, T. Kim, and Y. Shin, "Sysbumps: Exploiting Speculative Execution in System Calls for Breaking KASLR in macOS for Apple Silicon," In Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, pp. 64-78, 2024.

[6] Q. Zhou, X. Jia, and N. Jiang, "Protecting Virtual Machines Against Untrusted Hypervisor on ARM64 Cloud Platform," In ICC 2022 - IEEE International Conference on Communications, pp. 5451-5456, IEEE, 2022.

[7] M. S. Haq, A. Ş. Tosun, and T. Korkmaz, "Security Analysis of Docker Containers for ARM Architecture," In 2022 IEEE/ACM 7th Symposium on Edge Computing, pp.224-236, IEEE, 2022.

[8] J. Zhu et al., "CrossFire: Fuzzing macOS Cross-XPU Memory on Apple Silicon," In Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security, pp.3749-3762, 2024.

[9] S. Fan, Z. Hua, Y. Xia, H. Chen, and B. Zang, "Isa-grid: Architecture of Fine-Grained Privilege Control for Instructions and Registers," In Proceedings of the 50th Annual International Symposium on Computer Architecture, pp.1-15, 2023.

[10] Arm Ltd., "Virtualization Host Extensions" [Internet], https://developer.arm.com/documentation/102142/0100/Virtualization-host-extensions

[11] Arm Ltd., "IHI0069: ARM Architecture Reference Manual“ [Internet], https://developer.arm.com/documentation/ihi0069/latest