Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.36, NO.3, Jun. 2026 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2026.36.3.857

문서화되지 않은 macOS 인터페이스 식별을 통한 퍼블릭 프레임워크-XPC 서비스 의존성 그래프 생성 및 공격 표면 분석*

이 동 하,[1†] 강 민 주,[1] 한 규 상,[1] 박 정 우,[1] 전 승 호[2‡]
[1,2]가천대학교 (학생, 교수)

Constructing a Public Framework-XPC Service Dependency Graph via Undocumented macOS Interface Identification for Attack Surface Analysis*

Dongha Lee,[1†] Minju Kang,[1] Kyusang Han,[1] JungWoo Park,[1] Seungho Jeon[2‡]
[1,2]Gachon University (Undergraduate student, Professor)

요 약

macOS는 퍼블릭·프라이빗 프레임워크, XPC 서비스 등 다층적 구성요소로 이루어지며, 프라이빗 프레임워크는 공식 문서와 헤더 없이 배포되어 실제 시스템에서 노출되는 인터페이스의 상당 부분이 SDK 범위 밖에 존재한다. 이처럼 문서화되지 않은 인터페이스는 공격자가 악용 가능한 현실적인 공격 대상이 되지만, 기존 공격 표면 분석 연구는 시스템 호출이나 IOKit와 같은 단일 계층에 집중되어 숨겨진 공격 표면을 충분히 식별하지 못하고 있다. 이를 해결하기 위해 본 연구는 XPC 서비스·엔드포인트 식별, 정적·동적 프레임워크 의존성 분석, ObjC 메타데이터 분석을 통합하여 퍼블릭 프레임워크에서 프라이빗 프레임워크를 경유해 XPC 서비스로 이어지는 의존성 그래프 생성 방식을 제안하고, 2,918개 바이너리를 대상으로 유효성을 정량적으로 검증한다.

ABSTRACT

macOS consists of multi-layered components including public and private frameworks and XPC services. Private frameworks are distributed without official documentation or headers, leaving a substantial portion of the interfaces exposed in the actual system outside the scope of the SDK (Software Development Kit). Such undocumented interfaces represent realistic attack targets that adversaries can exploit. However, existing attack surface analysis research has focused on single-layer interfaces such as system calls and IOKit, failing to sufficiently identify hidden attack surfaces. To address this, the present study proposes a dependency graph construction approach that integrates XPC service and endpoint identification, static and dynamic framework dependency analysis, and ObjC metadata analysis to trace paths from public frameworks through private frameworks to XPC services, and quantitatively validates the methodology against a dataset of 2,918 binaries.

Keywords: macOS, Attack Surface, Undocumented Interface, Private Framework, XPC Service

Received(03. 03. 2026), Modified(05. 11. 2026), Accepted(05. 13. 2026)
* 본 논문은 2025년도 한국정보보호학회 동계학술대회에 발표한 우수논문을 개선 및 확장한 것임.
† 주저자, microgreen2019@gachon.ac.kr
‡ 교신저자, shjeon90@gachon.ac.kr(Corresponding author)

I. 서 론

macOS는 퍼블릭·프라이빗 프레임워크(Public·Private framework)[1], XPC 서비스, 드라이버 인터페이스 등 다층적 구성요소로 이루어진 계층적 시스템 아키텍처를 갖는다. 이 중 프라이빗 프레임워크는 애플이 내부 구현 목적으로 사용하며 의도적으로 공식 문서와 헤더를 제공하지 않아, 실제 시스템에서 노출되는 인터페이스의 상당 부분이 공개된 SDK(Software Development Kit) 범위 밖에 존재한다[1]. 본 연구에서는 이처럼 실제 시스템에 존재하고 호출 가능하지만 공식 문서나 SDK에 공개되지 않은 프라이빗 프레임워크와 XPC 서비스의 엔드포인트를 문서화되지 않은 인터페이스로 정의한다. 최근 macOS가 악성코드의 주요 타깃이 되고 공격 기법이 체계화되고 있다는 연구 결과[2]를 고려할 때, 이러한 문서화되지 않은 인터페이스는 공격자가 악용 가능한 현실적인 공격 대상이 된다[3]. 그러나 기존 공격 표면 분석 연구는 시스템 호출, IOKit, IOUserClient 등 단일 계층 인터페이스에 집중[4, 5]되어 있으며, 문서화되지 않은 인터페이스를 경유한 숨겨진 공격 표면은 충분히 식별되지 못하고 있다.

이를 해결하기 위해서는 세 가지 핵심 과제를 해결해야 한다. 첫 번째(C1), 시스템 전역에서 동작하는 XPC 서비스와 각 서비스가 외부로 노출하는 XPC 엔드포인트를 식별해야 한다. 두 번째(C2), 각 XPC 서비스가 사용하는 퍼블릭·프라이빗 프레임워크와의 정적 의존 관계를 파악해야 한다. 세 번째(C3), Mach-O(Mach Object) 정적 링킹에 드러나지 않는 XPC 서비스와 프레임워크 간의 동적 의존 관계를 추가로 파악해야 한다.

본 연구는 각 과제에 대해 다음과 같은 접근을 취한다. C1에 대해서는, launchd 서비스의 plist를 분석하여 시스템 전역의 XPC 서비스 식별자와 각 XPC 서비스별 XPC 엔드포인트를 식별한다. C2에 대해서는, Mach-O 포맷의 로드 커맨드(Load command)를 분석하여 XPC 서비스의 프레임워크 정적 의존성을 파악한다. C3에 대해서는, XPC 서비스의 Mach-O 바이너리에서 문자열 패턴 검색과 심볼 로딩 함수 사용을 분석하여 동적 의존성을 추가로 식별하며, ObjC 메타데이터 분석을 통해 식별된 프레임워크가 제공(export)하는 클래스·프로토콜 집합을 파악한다.

이러한 접근을 바탕으로, 본 연구는 문서화되지 않은 인터페이스 분석을 통해 퍼블릭·프라이빗 프레임워크와 XPC 서비스를 연결하는 의존성 그래프 생성 알고리즘을 제안한다. 제안 알고리즘은 XPC 서비스의 XPC 엔드포인트 식별, 정적·동적 의존성 추출, ObjC 메타데이터 기반 클래스·프로토콜 분석을 통합하여 퍼블릭 프레임워크에서 프라이빗 프레임워크를 경유해 XPC 서비스로 이어지는 숨겨진 공격 표면 경로를 모델링한다.

이전 연구[17]에서는 plist 파싱 기반 KEXT (Kernel Extension) 식별자 매칭으로 커널 공간 내 드라이버 간 상호 관계를 모델링하였으나, 분석 대상이 커널 익스텐션 단일 계층에 국한되어 유저 공간에서 형성되는 공격 표면을 식별하지 못한다는 한계가 있었다. 본 연구는 이를 확장하여 분석 범위를 퍼블릭·프라이빗 프레임워크와 XPC 서비스를 포함하는 유저 공간 전체로 넓히고, Mach-O 로드 커맨드 기반 정적 의존성 추출, ObjC 메타데이터 분석, 동적 의존성 탐지를 통합한 다층 분석 파이프라인을 새롭게 제안한다.

본 연구의 주요 기여는 다음과 같다.

- 퍼블릭·프라이빗 프레임워크와 XPC 서비스를 연결하는 의존성 그래프 생성 방식을 제안한다.
- 제안 알고리즘을 바탕으로 대규모 프레임워크 의존성 분석을 지원하는 파이프라인을 구현한다.
- 최신 macOS에서 추출한 2,918개 바이너리를 대상으로 대규모 실험을 수행하여 제안 방법론의 유효성을 정량적으로 검증한다.

나머지 본 논문의 구성은 다음과 같다. 2장에서 제안 방법론을 서술하기 위한 macOS에 대한 배경지식과 관련 연구를 설명한다. 3장에서는 다단계 분석을 통한 의존성 그래프 생성 방법론을 서술한다. 4장에서는 대규모 바이너리 데이터셋을 이용해 제안 방법론을 평가한다. 5장에서는 제안 방법론에 대해 논의하고, 마지막 6장에서 결론으로 마무리한다.

1) macOS에서는 라이브러리보다 프레임워크라는 표현을 선호한다.

II. 배경 지식 및 관련 연구

2.1 Mach Object 실행 파일 구조

Fig. 1은 Mach-O 실행 파일 구조를 보여준다[6]. Mach-O는 macOS 및 iOS에서 사용되는 실행 파일의 바이너리 포맷이다. Mach-O 파일은 크게 세 영역으로 구성된다. 파일 전체의 메타데이터를 기술하는 Mach-O 헤더(Mach-O header), 파일의 구조와 로딩 방식을 지시하는 로드 커맨드(Load commands), 그리고 실제 코드와 데이터가 저장되는 데이터(Data) 영역이다. Mach-O 헤더에는 CPU 아키텍처, 파일 유형, 로드 커맨드의 수와 전체 크기와 같은 파일 해석에 필요한 기본 정보가 포함된다.

로드 커맨드는 dyld(Dynamic Linker/Loader)가 바이너리를 메모리에 적재하는 방식을 결정하는 지시자들의 집합이다. 각 로드 커맨드는 세그먼트 커맨드(Segment command)와 그 하위의 섹션 헤더(Section header)로 계층적으로 구성되며, 세그먼트 커맨드는 데이터 영역의 특정 범위를 메모리의 어느 주소에 어떤 접근 권한으로 매핑할지를 정의한다. 섹션 헤더는 세그먼트 내의 세부 섹션(Section)을 기술하며, 실제 코드, 전역 변수, ObjC 메타데이터[7]와 같은 용도별로 구분된 데이터가 데이터 영역의 각 섹션에 저장된다. ObjC 메타데이터는 바이너리에 어떤 클래스와 프로토콜이 정의되어 있는지를 기록한 구조체로, 정적 분석만으로도 프레임워크 간 참조 관계를 파악할 수 있게 한다.

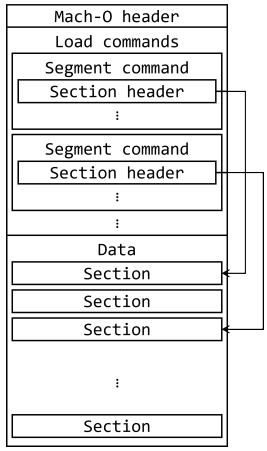


Fig. 1. Mach-O file format

2.2 macOS 아키텍처

Fig. 2는 macOS의 Mach IPC(Inter-Process Communication) 통신 흐름을 보여준다. Mach IPC는 그림과 같이 크게 세 영역으로 구성된다. 첫 번째, 클라이언트 측은 애플리케이션과 이를 지원하는 퍼블릭·프라이빗 프레임워크로 구성되며, 애플리케이션은 엔타이틀먼트(Entitlement)를 통해 특정 XPC 서비스에 대한 접근 권한을 부여받는다. 두 번째, launchd는 부트스트랩 서버(Bootstrap Server)로서 plist에 기술된 설정을 바탕으로 XPC 서비스를 등록·관리하며[8], 클라이언트의 서비스 조회 요청을 중개한다. 세 번째, XPC 서비스는 launchd에 의해 실행·관리되는 독립 프로세스로, 보안 정책 관리(securityd), 네트워크 설정(configd), 위치 정보(locationd)와 같은 특정 시스템 기능을 클라이언트에게 제공하는 서버 역할을

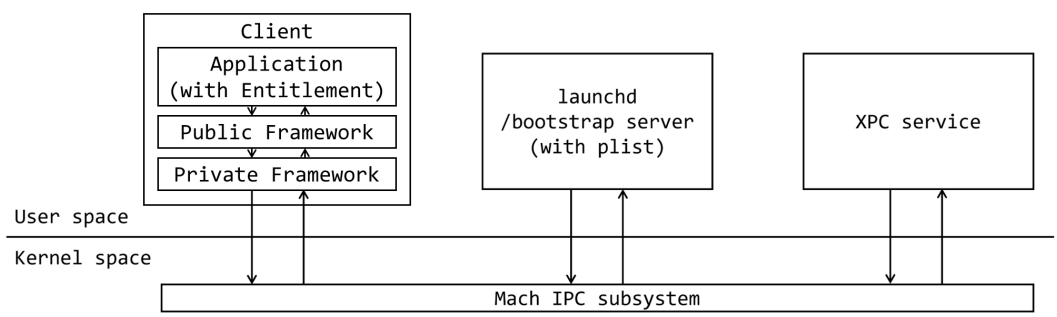


Fig. 2. Overview of Mach IPC mechanism in macOS

하며 하나 이상의 XPC 엔드포인트[2]를 외부에 노출한다.

하위 수준에서 모든 통신은 Mach IPC 서브시스템 상에서 이루어진다. 클라이언트가 XPC 서비스에 접근하려면 먼저 부트스트랩 서버에 XPC 엔드포인트 이름으로 조회를 요청하여 해당 서비스의 Mach 포트(Mach Port)를 획득하고, 이후 해당 포트를 통해 XPC 서비스와 직접 Mach 메시지를 교환한다. 즉, XPC는 Mach IPC 상에 구현된 고수준 추상화 계층이며[9], XPC 엔드포인트는 Mach 포트에 대응하는 식별자로 기능한다.

2.3 관련 연구

macOS 공격 표면 분석을 직접적으로 다루는 선행 연구는 현재까지 보고된 바 없다. 다만, 커널 익스텐션, 드라이버, Cross-XPU 메모리 등 단일 계층 인터페이스를 대상으로 취약점을 분석하는 연구들이 존재하며, 이들은 문서화되지 않은 인터페이스의 분석 필요성과 단일 계층 분석의 한계를 공유한다는 점에서 본 연구와 연결된다. 이번 장에서는 이러한 연구들에 대해 분석한다.

KextFuzz[10]은 애플 실리콘 macOS의 커널 익스텐션을 소스 코드·하이퍼바이저 없이 퍼징하는 시스템으로, PAC 명령어 교체를 통한 커버리지 수집, 엔타이틀먼트 검증 우회, 사용자 영역 인터페이스 추론의 세 기법을 제안한다. macOS의 보안 완화 기법을 역으로 활용하여 퍼징에 필요한 정보와 자원을 확보하는 것이 핵심 아이디어이며, 실제 기기 평가에서 48개의 고유 커널 버그와 CVE 5건을 발견하였다.

CrossFire[11]은 애플 실리콘의 통합 메모리 아키텍처(Unified Memory Architecture, UMA) 환경에서 CPU와 GPU/NPU가 공유하는 크로스-XPU 메모리를 대상으로 한 최초의 퍼저로, 유효한 퍼징 영역 식별 및 데이터 제약 추출을 자동화하였다. macOS Ventura에 대한 평가에서 15개의 취약점을 발견하였다.

SyzGen[12]는 소스 코드 없이 macOS 드라이버의 시스템 콜 명세를 자동으로 생성하며, 실행 추적 기반 의존성 추출과 점진적 보완을 통해 고품질 Syzkaller 템플릿을 생성한다. 실행 추적이 없는 인터페이스로 의존성을 일반화하는 서명 기반 추론이 핵심 기법이며, 특별한 하드웨어나 소스 코드 없이 커버리지를 수집하는 경량 커널 모듈도 함께 제안되었다. 25개 드라이버 평가에서 34개 버그와 2개 CVE를 발견하고 대조군 대비 평균 48% 커버리지 향상을 달성하였다.

III. 제안 방법론

3.1 위협 모델링

본 연구에서 공격자는 샌드박스 내의 일반 앱 프로세스 또는 사용자 권한의 비샌드박스 프로세스로 가정한다. 공격자의 목표는 퍼블릭 프레임워크를 통해 프라이빗 프레임워크를 경유하여 높은 권한을 가진 XPC 서비스가 노출하는 XPC 엔드포인트에 접근함으로써, 권한 상승, 정보 노출, 또는 시스템 상태 조작 가능성을 확대하는 것이다. 본 연구는 취약점 익스플로잇 자체를 목표로 하지 않으며, 의존성 그래프 생성을 통해 문서화되지 않은 인터페이스를 경유하는 경로의 도달 가능성과 노출 정도를 중심으로 공격 표면 확장 가능성을 측정한다.

3.2 개요

본 연구의 분석 범위는 퍼블릭 프레임워크에서 프라이빗 프레임워크를 거쳐 XPC 서비스로 이어지는 의존성 경로로 한정한다. 이를 위해 Fig. 3과 같이 다섯 단계로 구성된 의존성 그래프 구축 프로세스를 제안한다.

이 과정은 크게 두 흐름이 병렬로 수행된다. 첫

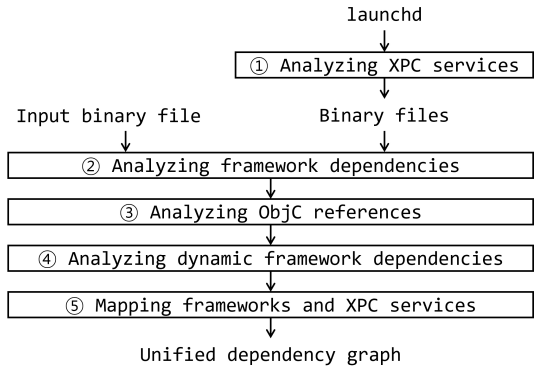


Fig. 3. Overview of constructing dependency graph

2) 혹은 Mach 서비스 이름(Mach Service Name)이라고도 한다.

번째 흐름은 입력 바이너리를 대상으로 Mach-O 로드 커맨드 분석을 통한 정적 프레임워크 의존성 추출, ObjC 메타데이터 분석을 통한 프레임워크 간 클래스·프로토콜 참조 관계 식별, 문자열 패턴 검색 및 심볼 적재 함수 분석을 통한 동적 의존성 식별을 순차적으로 수행한다. 두 번째 흐름은 launchd의 plist 분석을 통해 시스템 전역의 XPC 서비스와 XPC 엔드포인트를 식별하는 것으로, 첫 번째 흐름과 독립적으로 수행된다. 단, 식별된 XPC 서비스 데몬의 바이너리는 첫 번째 흐름과 동일한 분석 과정을 거친다. 최종적으로 두 흐름의 결과를 통합하여 퍼블릭·프라이빗 프레임워크와 XPC 서비스를 연결하는 의존성 그래프를 구축한다.

본 연구에서 구축하려는 의존성 그래프 $G=(V,E)$는 다음과 같이 정의된다. V는 의존성 그래프를 구성하는 노드의 집합으로, 여기에는 프레임워크 v_f, XPC 서비스v_s, XPC 엔드포인트v_e가 포함된다 ($v_f,v_s,v_e\in V$). $E(\in V\times V)$는 2개의 노드를 연결하는 엣지의 집합으로, 여기는 프레임워크간 연결 e_{ff}, XPC 서비스와 프레임워크간 연결 e_{sf}, XPC 서비스와 XPC 엔드포인트간 연결 e_{se}이 포함된다 ($e_{ff},e_{sf},e_{se}\in E$).

3.3 1단계: XPC 서비스 분석

1단계의 목표는 시스템 전역에서 동작하는 XPC 서비스와 XPC 서비스가 외부로 노출하는 XPC 엔드포인트를 식별하여 의존성 그래프 G의 초기 노드 및 엣지를 구성하는 것이다.

이 단계의 입력은 launchd이며, launchd가 관리하는 LaunchDaemons와 LaunchAgents의 plist를 분석하여 시스템 전역의 XPC 서비스와 각 XPC 서비스에 매핑된 XPC 엔드포인트를 식별한다[8]. 식별된 XPC 서비스는 노드 v_s로서 V에 추가되고, XPC 엔드포인트는 노드 v_e로서 V에 추가된다. 또한, 각 XPC 서비스와 XPC 엔드포인트 간의 관계는 엣지 e_{se}로서 E에 추가된다. 단, 부트스트랩 서버에 동적으로 등록되는 XPC 서비스는 plist에 기술되지 않으므로, plist 분석만으로는 시스템 전역의 모든 v_s와 v_e를 완전히 식별하는 것은 불가능하다.

3.4 2단계: 프레임워크 의존성 분석

2단계의 목표는 입력 바이너리가 컴파일 타임에 의존하는 프레임워크를 재귀적으로 탐색하여 퍼블릭 프레임워크와 프라이빗 프레임워크 간의 정적 의존 관계를 식별하고, 이를 의존성 그래프 G에 반영하는 것이다.

이 단계의 입력은 사용자가 제공한 Mach-O 바이너리 또는 1단계에서 식별된 XPC 서비스의 바이너리이다. 각 바이너리의 Mach-O 로드 커맨드 (LC_LOAD_DYLIB, LC_LOAD_WEAK_DYLIB, LC_REEXPORT_DYLIB)를 분석하여 정적으로 등록된 프레임워크 의존성을 식별한다 [6]. 식별된 프레임워크는 노드 v_f로서 V에 추가되고, 바이너리와 프레임워크 간, 혹은 프레임워크 간 의존 관계는 엣지 e_{ff}나 e_{sf}로서 E에 추가된다. 퍼블릭 프레임워크와 프라이빗 프레임워크의 구분은 프레임워크 경로를 기준으로 하며, /System/Library/Frameworks에 위치한 프레임워크는 퍼블릭 프레임워크로, /System/Library/PrivateFrameworks에 위치한 프레임워크는 프라이빗 프레임워크로 분류한다. 이후 식별된 프레임워크가 임포트하는 다른 프레임워크들을 재귀적으로 탐색하여 퍼블릭 프레임워크에서 프라이빗 프레임워크로 이어지는 의존 관계를 완전히 식별한다.

3.5 3단계: ObjC 참조 분석

3단계의 목표는 2단계에서 식별된 프레임워크 간의 ObjC 클래스·프로토콜 참조 관계를 분석하여, Mach-O 정적 링킹만으로 포착되지 않는 세밀한 수준의 프레임워크 간 의존 관계를 추가로 식별하고 의존성 그래프 G를 보강하는 것이다.

이 단계의 입력은 2단계에서 식별된 프레임워크 바이너리의 ObjC 메타데이터이다. 각 프레임워크 바이너리에서 OBJC_CLASS_$ 형태로 외부에 공개되는(export) 심볼[7]을 수집하여 해당 프레임워크가 제공하는 클래스·프로토콜 집합을 구성한다. 이후 프레임워크 v_f가 참조하는 외부 클래스·프로토콜 이름을 식별하고, 이를 외부에 제공하는(export) 프레임워크 v'_f를 전체 프레임워크 집합에서 탐색하여, 이들을 연결하는 엣지 e_{ff}를 E에 추가한다.

3.6 4단계: 프레임워크 동적 의존성 분석

4단계의 목표는 Mach-O 정적 링킹에 드러나지 않는 프레임워크 동적 의존성을 추가로 식별하여 의존성 그래프 G를 보완하는 것이다.

이 단계의 입력은 2단계에서 식별된 프레임워크 바이너리 및 1단계에서 식별된 XPC 서비스의 바이너리이다. 각 바이너리에서 dlopen·dlsym 관련 심볼의 존재를 탐지하여 동적 프레임워크[13] 적재 가능성을 기록하고, 바이너리 내 문자열 패턴 검색을 통해 *.framework 및 *.dylib 형태의 경로 문자열을 수집하여 동적으로 적재될 수 있는 프레임워크 후보를 식별한다. 식별된 동적 의존성은 노드 v_f로서 V에 추가되고, 해당 바이너리와의 의존 관계는 엣지 e_{ff} 또는 e_{sf}로서 E에 추가된다. 단, 암호화·압축된 문자열, 실행 중 동적으로 생성되는 경로, 외부 입력에 의해 구성되는 로딩 경로는 탐지하기 어려우므로, 본 단계의 결과는 실제 동적 의존성의 하한으로 해석되어야 한다.

3.7 5단계: 프레임워크와 XPC 서비스 연결

5단계의 목표는 1단계에서 식별된 XPC 서비스·XPC 엔드포인트와 2-4단계에서 식별된 프레임워크 의존성을 통합하여, 퍼블릭 프레임워크에서 프라이빗 프레임워크를 경유해 XPC 서비스로 이어지는 완전한 의존성 그래프 G를 완성하는 것이다.

이 단계의 입력은 1단계에서 구성된 XPC 서비스 집합 v_s와 2-4 단계에서 구성된 프레임워크 의존성 그래프이다. 1단계에서 식별된 XPC 서비스의 바이너리에 대해 2-4단계와 동일한 분석을 수행하여 해당 XPC 서비스가 의존하는 프레임워크를 식별하고, XPC 서비스 v_s와 프레임워크 v_f간의 의존 관계를 엣지 e_{sf}로서 E에 추가한다. 이를 통해 퍼블릭 프레임워크에서 프라이빗 프레임워크를 경유하여 XPC 서비스 및 XPC 엔드포인트로 이어지는 전체 의존성 경로가 표현된 그래프 G가 완성된다.

3.8 의존성 그래프 구축 알고리즘

Fig. 4는 전체 의존성 그래프 구축 프로세스를 나타낸다. 알고리즘은 빈 그래프 $G=(V,E)$를 초기화한 후, AnalyzeXPCservices를 통해 plist 디렉토리 P를 분석하여 XPC 서비스와 XPC 엔드포인트를 식별하고, 초기 노드 및 엣지를 구성한다 (1단계). 이후 1단계에서 식별된 서비스 바이너리 집합과 입력 바이너리 b_0를 대상으로 AnalyzeFrameworkDeps를 호출하여 Mach-O 로드 커맨드 기반의 정적 분석 프레임워크 의존성을 재귀적으로 호출한다 (2단계). 이어서 AnalyzeObjCref는 식별된 프레임워크 간의 ObjC 클래스·프로토콜 참조 관계를 분석하여 Mach-O 정적 링킹만으로 확인되지 않는 세밀한 수준의 의존 관계를 그래프에 추가한다 (3단계). AnalyzeDynamicDeps는 동일한 바이너리 집합을 대상으로 문자열 패턴 검색 및 dlopen 심볼 탐지를 수행하여 동적 의존성을 보완한다 (4단계). 마지막으로, IntegrateXPCframework는 1단계에서 구성된 XPC 서비스 집합과 2-4단계에서 구성된 프레임워크 의존성을 통합하여, 퍼블릭 프레임워크에서 프라이빗 프레임워크를 거쳐 XPC 서비스로 이어지는 완전한 의존성 그래프 G를 완성한다 (5단계).

Algorithm 1. Multi-Layer Dependency Graph Construction

Input: binary b_0, plist directory P
Output: $G = (V, E)$
1: $V \leftarrow \emptyset, \quad E \leftarrow \emptyset$
2: $(V, E) \leftarrow$ AnalyzeXPCService(P, V, E)
3: $D \leftarrow \{d \in V \mid d \text{ is XPC binary}\} \cup \{b_0\}$
4: $(V, E) \leftarrow$ AnalyzeFrameworkDeps$(D, V, E$
5: $(V, E) \leftarrow$ AnalyzeObjcRefs(V, E)
6: $(V, E) \leftarrow$ AnalyzeDynamicDeps(D, V, E)
7: $(V, E) \leftarrow$ IntegrateXPCFramework(V, E)
8: **return** $G = (V, E)$

Fig. 4. Algorithm for Multi-Layer Dependency Graph Construction

IV. 평 가

본 장에서는 제안한 의존성 그래프 구축 방법론의 유효성을 정량적으로 평가한다. 먼저, 구현 환경과 데이터셋 구성을 서술하고, 다음의 네 가지 연구 질문(Research Question; RQ)에 대한 실험 결과를 보고한다.

- RQ1. 다단계 분석을 통해 단일 단계 분석 대비 얼마나 많은 의존성을 추가로 식별할 수 있는가?
- RQ2. 퍼블릭 프레임워크에서 프라이빗 프레임워

크로 도달 가능한 경로는 얼마나 존재하는가?
- RQ3. 다수의 XPC 엔드포인트를 노출하는 XPC 서비스에 대해 공격 표면이 얼마나 집중되는가?
- RQ4. 동적 의존성 분석을 통해 정적 분석만으로는 식별되지 않는 의존성이 얼마나 추가로 식별되는가?

4.1 구현

본 연구 방법론은 Python으로 구현하였으며, 애플 M-시리즈 프로세서 기반의 macOS Tahoe 26.2 환경에서 수행하였다. Mach-O 로드 커맨드 분석, ObjC 메타데이터 분석, 문자열 패턴 기반 동적 의존성 탐지, launchd plist 파싱을 각각 독립적인 모듈로 구현하였으며, 최종적으로 각 모듈의 결과를 통합하여 의존성 그래프 G를 구성한다. 구현 소스코드 및 데이터셋은 GitHub에 오픈소스로 공개하였다[3].

macOS Tahoe 환경에서는 대부분의 시스템 프레임워크가 dyld shared cache에 통합되어 개별 바이너리 파일로 존재하지 않는다. 이를 처리하기 위해 dyld 계열 메타데이터를 활용하는 비추출 방식을 채택하였으며[14], 캐시로부터 심볼, 헤더, 의존성 정보를 직접 조회하여 개별 바이너리 추출 없이 의존성 분석을 수행한다.

4.2 데이터셋 서술

본 연구의 분석 대상은 macOS Tahoe 26.2 시스템에서 추출한 총 2,918개 바이너리이며, 구성은 Table 1과 같다. 퍼블릭 프레임워크는 /System/Library/Frameworks 경로에 위치한 268개 프레임워크이며, 프라이빗 프레임워크는 /System/Library/PrivateFrameworks 경로에 위치한 1,989개 프레임워크이다. XPC 서비스는 /System/Library/LaunchDaemons과 /System/Library/LaunchAgents의 plist에서 식별된 661개 서비스이다.

Table 1. Data distribution

Type	Count	Ratio (%)
Public Framework	268	9.2
Private Framework	1,989	68.2
XPC Service	661	22.7
Total	2,918	100

3) https://github.com/Apple-Reversing-Tools/MacOS-UserlandAttackSurface-visualization

4.3 RQ1: 단계별 의존성 식별 비교

다단계 분석의 효과를 정량적으로 평가하기 위해 다음 세 가지 단일 대조군(Baseline, BL)을 설정한다.
- BL1: Mach-O 로드 커맨드 기반의 정적 의존성만을 포함하는 의존성 그래프
- BL2: ObjC 클래스·프로토콜 참조 관계만을 포함하는 의존성 그래프
- BL3: launchd의 plist에서 식별된 XPC 서비스와 XPC 엔드포인트 간의 매핑만을 포함하는 의존성 관계

RQ1은 다단계 분석을 통해 단일 단계 분석 대비 얼마나 많은 의존성을 추가로 식별할 수 있는지를 측정한다. Table 2는 각 대조군과 통합 분석에서 식별된 노드 수와 엣지 수를 비교한다.

BL1 대비 BL2는 약 3.5배 많은 엣지를 식별하였다. 이는 ObjC 메타데이터 분석이 Mach-O 로드 커맨드만으로는 포착되지 않는 프레임워크간 클래스·프로토콜 참조 관계를 추가로 식별할 수 있게 하기 때문이다. 4개 단계를 모두 통합한 의존성 그래프는 BL1 대비 양 4.8배의 엣지 증가를 보였으며, 단일 단계 분석만으로는 상당량의 의존성 경로가 식별되지 않음을 확인하였다. 이러한 결과는 다단계 분석이 퍼블릭 프레임워크에서 프라이빗 프레임워크를 경유하여 XPC 서비스로 이어지는 숨겨진 공격 표면 경로를 보다 완전하게 식별할 수 있음을 시사한다.

또한, 제안한 그래프 생성 알고리즘의 확장성을

Table 2. Graph size comparison across baselines

Baseline	#Nodes	#Edges
BL1	2,428	8,247
BL2	2,428	28,865
BL3	1,422	2,133
Ours	**2,918**	**39,263**

평가하기 위해 그래프 구축에 걸리는 시간을 측정했다. 전체 2,918개 프레임워크를 대상으로 수행한 실험에서, 의존성 그래프 생성 시간은 평균 61.9초였으며, 8개 스레드를 사용한 병렬 환경에서는 14.6초로 감소하여 약 4.2배의 성능 향상을 보였다. 또한 입력 규모가 증가하는 환경에서도 안정적인 병렬 처리 효율을 유지하며, 선형적인 시간 증가를 확인하여 대규모 분석 대상에 대해 실용적인 수준의 확장성을 제공함을 보여준다.

4.4 RQ2: 퍼블릭 프레임워크에서 프라이빗 프레임워크로의 도달 가능성

RQ2는 퍼블릭 프레임워크에서 프라이빗 프레임워크로 도달 가능한 경로가 어느 정도 존재하는지를 측정한다. Table 3은 퍼블릭 프레임워크에서 프라이빗 프레임워크로의 도달 가능성 분석 결과를 측정한 것이다.

268개 퍼블릭 프레임워크 중 183개(68.4%)가 최소 1개의 프라이빗 프레임워크에 직접 의존하는 것으로 식별되었다. 직접 의존뿐만 아니라, 다른 프레임워크를 경유하여 도달 가능한 경우까지 포함하면 전체 프라이빗 프레임워크 1,989개 중 394개(19.8%)가 퍼블릭 프레임워크 경로를 통해 접근 가능한 것으로 파악되었다. 퍼블릭 프레임워크에서 프라이빗 프레임워크까지의 평균 최단 경로 길이는 1.7홉(hop)으로, 문서화된 API(Application Programming Interface) 호출만으로도 비교적 적은 단계를 거쳐 프라이빗 프레임워크에 도달할 수 있음을 확인하였다. 이러한 결과는 퍼블릭 프레임워크를 통한 프라이빗 프레임워크 접근이 예외적인 경우가 아니라 시스템 전반에 걸쳐 광범위하게 형성되어 있음을 의미한다.

Table 3. Public-to-Private framework reachability (FW: Framework, Dep: Dependency)

Measure	Count
Public FWs with direct Private Dep	183
Reachable Private FWs via Public paths	394
Avg. shortest path length	1.7
Max reachability depth	6

4.5 RQ3: XPC 서비스별 공격 표면 규모

RQ3은 다수의 XPC 엔드포인트를 노출하는 XPC 서비스에 공격 표면이 얼마나 집중되어있는지를 측정한다. Table 4는 노출하는 XPC 엔드포인트 수와 프라이빗 프레임워크 의존성 수를 기준으로 상위 5개 XPC 서비스의 특성을 보여준다.

단일 XPC 서비스가 최대 126개 프라이빗 프레임워크에 의존하면서 동시에 23개의 XPC 엔드포인트를 외부에 노출하는 사례가 관찰되었다. 이처럼 다수의 XPC 엔드포인트와 광범위한 프라이빗 프레임워크 의존성이 소수의 XPC 서비스에 집중되는 구조는, 해당 서비스가 공격자에게 넓은 접근 경로와 내부 의존성을 동시에 제공할 수 있음을 의미한다. 조사 결과, 상위 5개 XPC 서비스만으로도 전체 XPC 서비스가 노출하는 XPC 엔드포인트의 약 20%를 차지하며, 이는 소수의 XPC 서비스에 대한 우선적 분석이 효율적인 공격 표면 축소 전략이 될 수 있다는 것을 뜻한다.

Table 4. Top-5 XPC services (Priv: Private)

XPC Service	#Endpoints	#Priv.Dep.
callservicesd	23	126
assistantd	18	98
bluetoothd	15	87
corespeechd	12	74
passd	11	69

4.6 RQ4: 동적 의존성 식별

RQ4는 동적 의존성 분석을 통해 정적 분석만으로는 식별되지 않는 의존성이 얼마나 존재하는지를 측정한다.

문자열 패턴 기반 동적 의존성 탐지를 통해 Mach-O 정적 의존성 분석에서 식별되지 않은 18건의 추가 의존성을 식별하였다. 이 중 12건은 프라이빗 프레임워크로의 동적 의존성이었으며, 6건은 실행 시점에 로드하여 사용되는 동적 라이브러리 의존성이었다. 동적 의존성의 비율은 전체 엣지의 0.05% 미만으로 비교적 적은 수이지만, 정적 분석만으로는 식별할 수 없는 의존성 경로가 실제로 존재함을 확인하였다는 점에서 의미가 있다.

4.7 실제 취약점의 공격 표면 분석

마지막으로, 이 장에서는 과거 공개된 2개의 취약점의 공격 경로가 의존성 그래프에서 실제로 관찰되는지 확인한다. Fig. 5와 Fig. 6는 실제 취약점 사례[15, 16]의 공격 경로를 보여준다. 여기에서 녹색, 붉은색, 노란색 노드는 각각 퍼블릭 프레임워크, 프라이빗 프레임워크, XPC 서비스를 의미한다. 또한, 마름모 노드는 XPC 서비스를 의미하고 그 외에는 둥근 노드로 표현했다.

먼저, Pwn2Own 2018에서 공개된 CVE-2018-4193 취약점[15]은 WindowServer의 Mach 메시지 처리 과정에서 발생한 취약점으로, 사파리 브라우저 샌드박스 환경의 애플리케이션이 CoreGraphics 프레임워크를 통해 SkyLight 프라이빗 프레임워크를 거쳐 WindowServer와의 IPC 경로를 발생시킴으로써 권한 상승이 가능하다고 보고되었다(Fig. 5).

두 번째로, CVE-2025-43401 취약점[16]은 사파리 브라우저에서 시작되지만, 실제 서비스 거부 공격은 WindowServer 프로세스에서 발생한다(Fig. 6). 취약점의 근본 원인은 CoreAnimation에 존재하지만, 호출 경로는 사파리 브라우저에서 WebKit, QuartzCore(Core Animation), SkyLight을 거쳐 WindowServer로 이어지며, 퍼블릭 프레임워크를 통해 프라이빗 프레임워크와 시스템 서비스로 전달되는 구조를 갖는다. 이러한 사례는 퍼블릭 프레임워크가 내부적으로 프라이빗 프레임워크 및 다양한 XPC 서비스와 연결되면서 숨겨진 공격 표면이 형성될 수 있음을 보여준다.

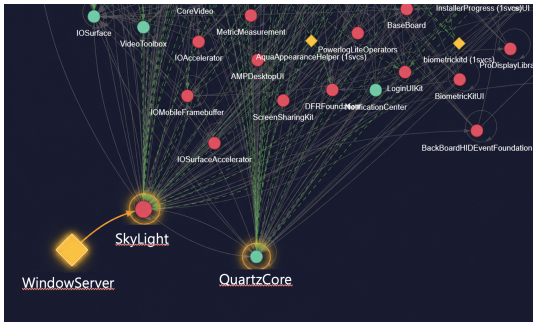


Fig. 5. CVE-2018-4193 in Dependency Graph

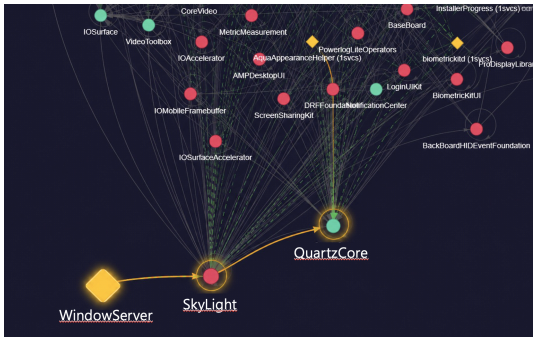


Fig. 6. CVE-2025-43401 in Dependency Graph

V. 논 의

본 연구에서 제안한 방법론은 문서화된 API만을 대상으로 하는 기존 접근과 달리, 퍼블릭 프레임워크에서 프라이빗 프레임워크를 경유하여 XPC 서비스로 이어지는 숨겨진 의존성 경로를 체계적으로 식별할 수 있다. 특히, Mach-O 정적 의존성 분석, ObjC 메타데이터 분석, 동적 의존성 분석을 통합함으로써 단일 단계 분석만으로는 포착되지 않는 프레임워크 간 의존 관계를 세밀한 수준에서 식별할 수 있다. 또한 구축된 의존성 그래프를 바탕으로 다수의 XPC 엔드포인트와 광범위한 프라이빗 프레임워크 의존성을 동시에 보유하는 XPC 서비스를 식별할 수 있어, 보안 감사 시 우선적으로 분석해야 할 대상을 효율적으로 선별할 수 있다.

본 연구는 전적으로 정적 분석에 기반하므로, 실행 중 결정되는 동적 행위를 완전하게 포착하지 못한다는 한계가 있다. 보다 구체적으로, 난독화·압축된 문자열에 포함된 프레임워크 경로, 동적으로 등록되는 XPC 서비스, 조건 분기에 의해 선택적으로 실행되는 dlopen 호출 등은 본 연구의 분석 범위 밖에 있다. 또한 문자열 패턴 기반 동적 의존성 탐지는 일부 오탐을 포함할 수 있다. 따라서, 본 연구에서 구축된 의존성 그래프는 실제 공격 표면의 하한으로 해석되어야 한다.

VI. 결 론

본 연구는 macOS의 문서화되지 않은 인터페이스가 형성하는 숨겨진 공격 표면을 식별하고 정량화하는 다단계 정적 분석 방법론을 제안하였다. 제안 방법론은 XPC 서비스·XPC 엔드포인트 식별, 정적 의존성 추출, ObjC 메타데이터 분석, 동적 의존성

탐지의 4단계를 통합하여 퍼블릭 프레임워크에서 프라이빗 프레임워크를 경유해 XPC 서비스로 이어지는 의존성 그래프를 구축한다. 2,918개 바이너리를 대상으로 한 실험에서 다단계 분석이 단일 단계 분석 대비 약 4.8배 많은 의존성을 식별하였으며, 퍼블릭 프레임워크의 68.4%가 프라이빗 프레임워크로의 직접 진입점으로 기능할 수 있음을 확인하였다.

제안 방법론은 macOS 보안 감사, 취약점 분석, 악성코드 분석 등에 활용될 수 있으며, 특히 우선적 감사 대상 XPC 서비스를 선별하는 데 유용한다. 향후 연구에서는 동적 분석과 결합을 통한 실행 중 도달 가능성 검증을 수행할 계획이다.

References

[1] A. Kharlamova, Y. Sun, and T. Yu, "Exposing Hidden Interfaces: LLM-Guided Type Inference for Reverse Engineering macOS Private Frameworks," arXiv, 2601/01673, Jan. 2026.

[2] D. L. Miro, J. Carrillo-Mondejar, and R. J. Rodriguez, "Characterizing tactics, techniques, and procedures in the macOS threat landscape," Computers & Security, vol. 162, Mar. 2026.

[3] D. Toldo, J. Classen, and M. Hollick, "Attaching InternalBlue to the proprietary macOS IOBluetooth framework," Proceedings of the 2020 ACM Conference on Security and Privacy in Wireless and Mobile Networks, pp. 328-330, Jul. 2020.

[4] M. Blochberger, J. Rieck, C. Burkert, T. Mueller, and H. Federrath, "State of the Sandbox: Investigating macOS Application Security," Proceedings of the 2019 ACM Workshop on Privacy in the Electronic Society, pp. 150-162, Nov. 2019.

[5] X. Bai, L. Xing, M. Zheng, and F. Qu, "iDEA: Static Analysis on the Security of Apple Kernel Drivers," Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pp.1185-1202, Nov. 2020.

[6] Apple Inc., "OS X ABI Mach-O File Format Reference," Apple Developer Documentation, https://github.com/aidansteele/osx-abi-macho-file-format-reference, Accessed 25 May 2026.

[7] Apple Inc., "Objective-C Runtime Programming Guide," Apple Developer Documentation, https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ObjCRuntimeGuide/, Accessed 25 May 2026.

[8] Apple Inc., "Creating Launch Daemons and Agents," Apple Developer Documentation, https://developer.apple.com/library/archive/documentation/MacOSX/Conceptual/BPSystemStartup/Chapters/CreatingLaunchdJobs.html, Accessed 25 May 2026.

[9] Apple Inc., "XPC Services," Apple Developer Documentation," Apple Developer Documentation, https://developer.apple.com/documentation/xpc, Accessed 25 May 2026.

[10] T. Yin, Z. Gao, Z. Zheng, Z. Ma, M. Zheng, and C. Zhang, "KextFuzz: Fuzzing macOS Kernel EXTensions on Apple Silicon via Exploiting Mitigations," Procedings of the 2023 Usenix Security Symposium, pp. 5039-5054, Aug. 2023.

[11] J. Zhu, M. Lin, T. Yin, Z. Cai, Y. Wang, R. Chang, and W. Shen, "CrossFire: Fuzzing macOS Cross-XPU Memory on Apple Silicon," Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communication Security, pp. 3749-3762, Dec. 2024.

[12] W. Chen, Y. Wang, Z. Zhang, and Z. Qian, "SyzGen: Automated Generation of Syscall Specification of Closed-

Source macOS Drivers," Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communication Security, pp. 749-763, Nov. 2021.

[13] Apple Inc., "Dynamic Library Programming Topics," Apple Developer Documentation, https://developer.apple.com/library/archive/documentation/DeveloperTools/Conceptual/DynamicLibraries/, Accessed 25 May 2026.

[14] Apple Inc., "dyld," Apple OSS Distributions, https://github.com/apple-oss-distributions/dyld, Accessed 25 May 2026.

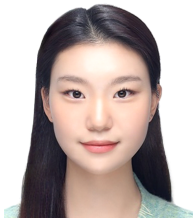
[15] CVE, "CVE-2018-4193," https://nvd.nist.gov/vuln/detail/cve-2018-4193, Accssed 25 May 2026.

[16] CVE, "CVE-2025-43401," https://nvd.nist.gov/vuln/detail/CVE-2025-43401, Accssed 25 May 2026.

[17] D. Lee, M. Kang, G. Han, J. Park, S. Jeon, "Automated Attack Surface Identification via XPC and IOKit in macOS," Proceedings of Conference of Information Security and Cryptography-Winter, Nov. 2025

〈저자소개〉

이 동 하 (Dongha Lee) 학생회원
2023년 3월~현재 : 가천대학교 스마트보안학과 재학
〈관심분야〉 시스템 보안



강 민 주 (Minju Kang) 학생회원
2026년 2월 : 가천대학교 스마트보안학과 졸업
2026년 3월~현재 : 가천대학교 스마트보안학과 석사과정
〈관심분야〉 모바일 보안

한 규 상 (Kyusang Han) 학생회원
2020년 3월~현재 : 가천대학교 인공지능학과 재학
〈관심분야〉 프로그램 정적분석

박 정 우 (JungWoo Park) 학생회원
2021년 3월~현재 : 가천대학교 컴퓨터공학과 재학
〈관심분야〉 AI 기반 취약점 분석

전 승 호 (Seungho Jeon) 정회원
2016년 2월 : 명지대학교 컴퓨터공학과 졸업
2018년 2월 : 고려대학교 정보보호학과 석사
2022년 8월 : 고려대학교 정보보호학과 박사
2024년 3월~현재 : 가천대학교 스마트보안학과 조교수
〈관심분야〉 정보보호, 시스템 보안, 소프트웨어 보안