JKSCI

한국컴퓨터정보학회논문지
Journal of The Korea Society of Computer and Information
Vol. 31 No. 5, pp. 149-163, May 2026
https://doi.org/10.9708/jksci.2026.31.05.149

Design of a Hardware Branch Tracing-Based Framework for External API Identification on Windows

Jun-Seob Kim*

*Professor, Dept. of AI Cyber Security, Korea University, Sejong, Korea

[Abstract]

In this paper, we propose a hardware branch tracing-based framework for identifying external APIs invoked during program execution on Windows. The proposed framework collects branch logs using Intel Last Branch Records (LBR), filters branches directed to external modules in a post-processing stage, and maps their destination addresses to function information of loaded DLLs to identify invoked external APIs. Because this approach is based on branch traces observed during execution, it does not rely only on statically exposed reference information or on predefined API hooking points. Experiments were conducted on four basic scenarios, namely normal-entry calls, dynamic-loading calls, native API direct calls, and internal-function entry calls. Additional experiments were also conducted on dynamic-loading variants with obfuscated API names and detection of debugging or hooking attempts. The results show that the proposed framework identified external APIs in all scenarios and achieved a 100% identification rate in the four basic scenarios. These findings indicate that the proposed method provides a practical way to identify external APIs while minimizing direct runtime intervention.

▸ **Key words**: External API, API Identification, Hardware Branch Tracing, Intel LBR, Native API, API Hooking

[요 약]

본 논문에서는 Windows 환경에서 프로그램 실행 중 호출된 외부 API를 식별하기 위한 하드웨어 분기 추적 기반 프레임워크를 제안한다. 제안 프레임워크는 Intel LBR로 분기 로그를 수집하고, 후처리 단계에서 외부 모듈로 향하는 분기의 목적지 주소를 실행 중 로드된 DLL의 함수 정보와 대응시켜 외부 API를 식별한다. 이 방식은 실행 중 관측된 분기 정보를 기반으로 하므로, 정적으로 드러난 참조 정보나 사전에 정의된 API 후킹 지점에만 의존하지 않는다. 기본 실험은 정상 진입 호출, 동적 로드 호출, 네이티브 API 직접 호출, 함수 내부 진입 호출의 네 가지 시나리오를 대상으로 하였으며, 함수 이름 난독화와 디버깅·후킹 탐지가 적용된 동적 로드 변형 시나리오도 추가로 평가하였다. 그 결과 제안 프레임워크는 모든 시나리오에서 외부 API를 식별했으며, 네 가지 기본 시나리오에서는 100%의 식별률을 보였다. 이는 제안 프레임워크가 실행 과정에 대한 직접 개입을 최소화하면서도 실제 호출된 외부 API를 식별할 수 있는 실용적인 방법임을 보여준다.

▸ **주제어**: 외부 API, API 식별, 하드웨어 분기 추적, Intel LBR, 네이티브 API, API 후킹

• First Author: Jun-Seob Kim, Corresponding Author: Jun-Seob Kim
*Jun-Seob Kim (jskim90@korea.ac.kr), Dept. of AI Cyber Security, Korea University
• Received: 2026. 04. 14, Revised: 2026. 05. 04, Accepted: 2026. 05. 12.

http://www.ksci.re.kr pISSN:1598-849X | eISSN:2383-9945

I. Introduction

프로그램은 실행에 필요한 모든 기능을 자체적으로 포함하기보다, 외부 모듈에서 제공하는 API(Application Programming Interface)를 이용하여 필요한 기능을 수행하도록 구성된다[1]. 이러한 외부 API 호출 정보는 실행 중인 프로그램의 행위를 분석하는 데 중요한 단서를 제공한다[2][3]. 특히 악성코드 분석과 같은 보안 분석 환경에서는 파일 접근, 프로세스 생성, 메모리 조작, 네트워크 통신과 같은 외부 API 호출 정보가 프로그램의 실제 의도를 파악하는 핵심 근거로 활용된다[2-4].

기존의 외부 API 식별 방식은 주로 정적 분석을 통해 코드에 정적으로 드러난 외부 API를 식별하거나[5][7-9], 프로그램을 실제 실행한 후 실행 과정에서의 외부 API 호출 여부를 관찰하는 방식에 의존해 왔다[2][3][6]. 이러한 접근은 일반적인 환경에서는 유용하지만, 분석을 방해하는 다양한 기법들이 적용된 경우에는 외부 API를 안정적으로 식별하는 데 한계가 있다[3][4]. 또한 프로그램의 실행 과정에 직접 개입하는 방식은 분석 대상이 개입을 탐지하거나, 해당 관측 방식을 우회할 가능성을 높인다[3][4]. 따라서 실행 과정에 대한 개입을 줄이면서도, 실행 중 호출된 외부 API를 보다 안정적으로 식별할 수 있는 새로운 방법이 필요하다.

이에 본 논문은 프로그램 실행 중 내부 코드 영역에서 외부 모듈 영역으로 이동하는 분기 정보를 하드웨어 수준에서 수집하고, 이를 실제로 로드된 모듈의 함수 정보와 대응시켜 외부 API를 식별하는 프레임워크를 제안한다. 이를 통해 실행 과정에서 관찰된 분기 정보를 기반으로 실행 중 호출된 외부 API를 식별함으로써, 정적 분석 기반 식별 방식과 실행 과정에 직접 개입하는 기존 동적 분석 방식이 가지는 한계를 보완하는 것을 목표로 한다.

다만 본 연구는 외부 API 호출의 전체 의미를 완전하게 해석하거나 인자와 반환값까지 분석하는 것을 목표로 하지는 않는다. 또한 본 논문은 제안 프레임워크의 기본 식별 가능성을 확인하기 위해 호출 구조를 명확히 통제할 수 있는 실험 시나리오를 중심으로 평가를 수행한다. 그럼에도 불구하고 분석 방해 기법이 적용된 환경에서는 기존의 외부 API 식별 방식만으로는 실행 중 호출된 외부 API 정보를 안정적으로 확보하기 어려울 수 있다[3][4]. 따라서 호출된 외부 API와 관련된 기본 정보를 확보하는 것 자체로도 중요한 의미를 가지며, 이러한 외부 API 식별 정보는 프로그램의 주요 행위를 해석하기 위한 핵심 근거로 활용될 수 있다.

본 논문의 주요 기여는 다음과 같다.

- 하드웨어 기반 분기 추적을 활용하여 프로그램 실행 과정에 대한 개입을 줄이면서 외부 모듈로의 분기 정보를 수집하고, 이를 바탕으로 외부 API를 식별할 수 있는 프레임워크를 제안한다.
- 실행 과정에서 관찰된 분기 정보와 실행 중에 로드된 모듈의 함수 정보를 연계하여, 외부 API를 식별하는 방법을 제시한다.
- 통제된 실행 시나리오를 통해 제안 프레임워크의 외부 API 식별 가능성을 평가한다.

본 논문의 구성은 다음과 같다. II장에서는 외부 API 식별, 분석 방해 기법, 하드웨어 기반 분기 추적, 그리고 관련 기존 연구를 포함한 배경지식을 설명한다. III장에서는 제안하는 외부 API 식별 프레임워크의 구조와 핵심 절차를 기술한다. IV장에서는 실험 환경, 평가 시나리오, 정량적 결과 및 사례 분석을 제시하고, V장에서는 본 연구의 결론과 한계를 정리하고 향후 연구 방향을 제시한다.

II. Background and Related Work

1. Background

1.1 External API Identification

외부 API 식별은 프로그램이 실행 과정에서 자체적으로 구현하지 않은 기능을 어떤 외부 인터페이스를 통해 수행하는지를 확인하는 과정을 의미한다[2][3]. Windows 환경에서 이러한 외부 기능들은 주로 DLL(Dynamic Link Library) 형태로 제공되며, 프로그램은 필요한 기능을 DLL이 제공하는 함수 호출을 통해 수행한다[1][5]. 이 과정에서 프로그램은 실행 시 참조할 외부 함수들의 정보를 IAT(Import Address Table)에 저장하며, DLL은 외부에 제공하는 함수들의 정보를 EAT(Export Address Table)에 저장한다[5]. 따라서 외부 API 식별은 프로그램이 어떤 DLL의 어떤 함수를 참조하고 사용하는지를 파악하는 문제로 볼 수 있다.

외부 API 식별은 크게 정적 분석 및 동적 분석 기반 접근으로 구분할 수 있다[2][3]. 정적 분석에서는 외부 API 식별을 위해서 프로그램의 IAT에 등록된 정보, 문자열, 디스어셈블 결과 등을 이용하여 사용할 가능성이 있는 외부 API를 추정한다[7-9]. 이때 가장 기본적으로 활용되는 정보가 IAT이며, 이를 통해 해당 실행 파일이 어떤 DLL의 어떤 함수를 참조 대상으로 포함하고 있는지를 확인할 수

있다[5]. 이러한 접근은 프로그램을 실제로 실행하지 않고 분석할 수 있다는 장점이 있으나, 실행 과정에서 어떤 API가 실제로 호출되었는지를 직접 확인할 수는 없다[2][4].

반면 동적 분석에서는 프로그램을 실제로 실행하면서 외부 API 호출을 관찰한다[2][3]. 대표적인 방법은 API 후킹이며, 이는 특정 API 사용 시점에 개입하여 관련 정보를 기록하는 방식이다[3][6][10]. 이러한 방법은 프로그램의 IAT 또는 DLL의 EAT에 기록된 호출 함수 주소를 다른 주소로 변경하여 호출이 후킹 함수로 먼저 전달되도록 하는 방식을 사용하거나, API 함수가 위치한 코드 영역 일부를 직접 변경하는 방식 등으로 구현될 수 있다[5][6][10]. 이러한 접근은 실제 실행 과정에서 호출된 외부 API를 확인할 수 있다는 장점이 있으나, 프로그램 코드나 실행 환경에 대한 개입이 수반될 수 있으므로 성능 저하나 오동작 가능성이 존재하며, 후킹 여부를 탐지하거나 이를 우회하는 기법의 영향을 받을 수 있다는 한계를 가진다[3][4].

이처럼 외부 API 식별은 정적 분석 정보에 기반하여 사용 가능성을 추정하는 방법과, 실행 중 호출되는 외부 API를 관찰하는 방법으로 수행될 수 있다. 그러나 정적 분석은 실제 사용 여부를 직접 보장하지 못하고, 동적 분석은 개입에 따른 부담과 우회 가능성의 문제를 가진다. 따라서 외부 API 식별에서는 실제 실행 과정에서 호출되는 외부 API를 가능한 한 직접 개입과 같은 왜곡 없이 관찰하고 식별할 수 있도록 하는 방법이 필요하다.

1.2 Anti-analysis Considerations

분석 방해 기법은 분석 대상 프로그램이 분석 환경의 존재를 인식하고, 이를 탐지·우회하거나 분석 결과를 왜곡하기 위해 사용하는 기법을 의미한다[2][4]. 외부 API 식별은 실제 실행 과정에서 호출되는 외부 API를 관찰해야 하므로 동적 분석 기반 접근에 크게 의존하게 되며, 이러한 특성 때문에 분석 방해 기법의 영향을 직접적으로 받을 수 있다[2-4]. 특히 프로그램이 분석 환경의 존재를 인식할 경우, 원래와 다른 실행 경로를 선택하거나 특정 동작을 생략함으로써 외부 API 식별 결과의 신뢰성을 저하시킬 수 있다[4]. 동적 분석 기반의 외부 API 식별 방법들은 대체로 프로그램의 실행 과정에 일정 수준 개입한다. 예를 들어 API 후킹은 IAT 또는 EAT의 항목을 수정하거나, API 함수가 위치한 코드 영역 일부를 변경하는 방식으로 구현될 수 있다[6][10]. 또한 디버거를 이용해서 코드를 실행시키거나 DBI(Dynamic Binary Instrumentation)를 이용하는 방식 역시 프로그램의 실행 환경과 코드 수행 과정에서 추가적인 흔적을 남길 수 있다[4][11][12]. 그리고 이러한 개입은 분석 대상에게 관찰 가능한 변화로 나타날 수 있으며, 결과적으로 분석 방해 기법의 탐지 표면을 형성한다.

분석 대상 프로그램은 이러한 흔적을 이용하여 분석 환경의 존재를 추정할 수 있다. 예를 들어 참조 테이블이나 코드 영역의 변경 여부를 점검하거나, 비정상적인 모듈 로드 상태, 디버깅 관련 정보, 실행 시간 증가와 같은 간접적인 징후를 확인할 수 있다[2][4]. 이와 같은 탐지가 성공하면 프로그램은 특정 API 사용을 우회할 수도 있고, 일부 기능의 실행을 지연시키거나 아예 실행하지 않을 수 있다. 또한 분석 환경에서만 다른 행위를 수행할 수도 있다[4]. 그 결과 수집한 외부 API 정보가 실제 프로그램의 본래 동작을 충분히 반영하지 못할 수 있다.

따라서 외부 API 식별에서는 단순히 외부 API 호출 정보를 수집하는 것만으로는 충분하지 않으며, 분석 과정에서 프로그램의 실행에 미치는 영향을 함께 고려해야 한다. 특히 분석 방해 기법이 적용된 프로그램 분석 시에는 실행 과정에 대한 직접 개입을 최소화하면서, 실제 실행 흐름을 가능한 한 보존한 상태에서 외부 API를 식별할 수 있는 접근이 중요하다[3][4]. 이러한 배경은 다음 절에서 설명할 하드웨어 기반 분기 추적의 필요성과도 연결된다.

1.3 Hardware Branch Tracing

하드웨어 기반 분기 추적은 프로그램 실행 중 발생하는 제어 흐름 정보를 프로세서 하드웨어 수준에서 기록하거나 관찰하는 방법을 의미한다[13][14]. 이러한 접근은 프로그램 코드나 실행 경로를 직접 수정하지 않고도 실행 흐름을 추적할 수 있다는 점에서, API 후킹이나 DBI와 같은 기존 동적 분석 방식과 구별된다. Intel 아키텍처에서는 이러한 목적을 위한 여러 기능이 제공되며, 대표적으로 Intel PT(Intel Processor Trace), BTS(Branch Trace Store), LBR(Intel Last Branch Record)가 있다[13][14]. 이들 기능은 모두 분기 실행 정보를 기록한다는 공통점을 가지지만, 기록 방식과 정보의 범위, 활용 목적에는 차이가 있다.

PT는 프로그램의 실행 흐름을 정밀하게 추적할 수 있는 하드웨어 기능으로, 분기 흐름과 관련된 상세한 추적 정보를 제공한다[13]. 이러한 특성으로 인해 PT는 정밀한 실행 흐름 분석이나 API 호출 순서 복원과 같은 문제에 활용될 수 있다. 그러나 PT의 추적 결과만으로는 실제 실행된 명령어 흐름을 곧바로 복원하기 어렵다. 추적 해석을 위해서는 실행 시점의 코드 이미지와 그 변화 이력, 그리고 관련 부가 정보가 함께 확보되어야 하며, 실행 중 코드 이미지가 변하는 환경에서는 이러한 해석 과정이 더욱 복잡해질

수 있다[13]. 따라서 PT는 풍부한 정보를 제공한다는 장점이 있으나, 이를 실제 분석에 활용하기 위해서는 별도의 해석 절차와 지원 환경이 요구된다는 한계를 가진다.

BTS는 분기 실행 정보를 메모리 버퍼에 연속적으로 저장하는 하드웨어 기능으로, 긴 구간의 분기 이력을 기록할 수 있다[13]. 이러한 점에서 BTS는 일반적인 실행 흐름 분석이나 제어 흐름 재구성과 같은 문제에 활용될 수 있다. 그러나 BTS는 메모리 버퍼를 지속적으로 사용하여 분기 정보를 기록하므로 실행 환경과 기록 설정에 따라 추가적인 처리 부담을 수반할 수 있으며, 활용 가능 여부 또한 프로세서 및 실행 환경의 지원 범위에 영향을 받을 수 있다[13]. 따라서 BTS는 긴 분기 이력을 확보할 수 있지만, 기록 방식과 환경 지원 측면에서 제약이 있다.

반면 LBR은 최근에 실제로 수행된 분기들에 대한 시작 주소와 목적지 주소를 제한된 개수만큼 하드웨어 레지스터에 직접 기록하는 기능이다[13][14]. 이 방식은 기록 깊이는 짧지만, 최근 분기 정보를 제한된 하드웨어 기록 형태로 유지하며, 실행 흐름의 최근 분기 정보를 직접 확인할 수 있다는 장점이 있다[13][14]. 특히 프로그램의 제어 흐름이 내부 코드 영역에서 외부 모듈 영역으로 이동하는 지점을 비교적 단순한 형태로 관찰할 수 있으므로, 외부 API 식별과 같이 모듈 간 분기를 중심으로 실행 흐름을 포착하는 문제와 자연스럽게 연결된다. 그러나 LBR은 최근 분기만을 제한적으로 저장하므로 기록 깊이에 한계가 있으며, 저장되는 분기 주소 정보만으로는 특정 API 정보를 직접 제공하지는 않는다[14].

본 연구는 이러한 특성들을 종합적으로 고려하여 LBR을 외부 API 식별을 위한 핵심 추적 수단으로 선택한다. PT는 풍부한 실행 흐름 정보를 제공하지만 해석 부담이 크고, BTS는 더 긴 분기 이력을 기록할 수 있으나, 지속적인 메모리 기록과 실행 환경의 지원 범위에 따른 제약이 있다. 반면 LBR은 기록 깊이는 제한적이지만, 프로그램 코드나 실행 경로를 직접 변경하지 않고도 실제 수행된 분기의 시작 주소와 목적지 주소를 제한된 하드웨어 기록을 통해 확보할 수 있다. 본 연구의 목적은 단순히 분기 실행 정보를 수집하는 데 있지 않고, 수집된 LBR 기반 분기 정보를 실제 외부 API 식별로 연결하는 데 있다. 이를 위해 본 연구는 분기 정보와 함께 실제 로드된 모듈의 베이스 주소, 이미지 크기, 로드 상태와 같은 모듈 메타데이터를 함께 수집하고, 분기 목적지 주소를 해당 모듈의 주소 범위 및 함수 정보와 대응시킴으로써 실행 중 호출된 외부 API를 식별한다. 이러한 점에서 본 연구는 하드웨어 지원 분기 추적에서 얻은 저수준 분기 정보를 Windows 환경에서 외부 API 식별 문제로 직접 연결한다는 점에서 기존 연구와 차별성을 가진다. 다만 본 논문에서는 하드웨어 추적 방식 간의 정량적 오버헤드 비교보다는, LBR에서 수집한 분기 주소 정보를 외부 API 식별 문제로 연결하는 절차에 초점을 둔다.

2. Related Work

2.1 Static Analysis-Based API Identification

정적 분석 기반 API 식별은 실행 파일에 정적으로 드러난 정보로부터 외부 API 후보를 추정하거나 관련 특징을 추출하는 접근이다[2][3]. 이러한 접근은 이후 악성코드 식별 및 분류를 위한 특징 추출로 확장되었다. 예를 들어 Ye 등은 PE 파일의 Windows API 순서를 분석하여 IMDS를 제안하였고[7], Iwamoto와 Wasaki는 정적 분석으로 추출한 API 순서를 기반으로 악성코드 분류 방법을 제시하였으며[8], Naik은 임포트 해싱을 확장한 퍼지 임포트 해싱을 통해 IAT 기반 정적 특징의 활용 가능성을 보였다[9]. 그러나 이러한 접근은 실행 없이 분석이 가능하다는 장점이 있지만, 실제 실행 과정에서 어떤 외부 API가 사용되었는지를 직접 보장하지 못하고, 실행 중 동적으로 결정되는 API 사용이나 함수 내부 진입과 같은 호출 형태를 충분히 반영하기 어렵다[2][4].

2.2 Hooking-Based API Identification

API 후킹 기반 API 식별은 프로그램 실행 중 특정 API 진입 시점에 개입하여 외부 API 호출 정보를 직접 관찰하는 접근이다[3][6]. 대표적인 방법은 IAT 수정, EAT 수정, 코드 영역 변경과 같은 방식의 API 후킹이며[6][10], HookTracer와 같은 연구는 이러한 후킹 분석을 자동화하였다[15]. 이와 유사하게 디버거나 DBI와 같은 직접 개입 기반 관찰 방식도 API 사용 여부를 확인할 수 있으나[11][12], 이들 방식은 모두 프로그램의 실행 과정이나 실행 환경에 흔적을 남기므로 실행 오버헤드, 비정상 동작, 탐지 및 우회 가능성의 영향을 받을 수 있다[2-4]. 따라서 이 계열의 접근은 실제 호출된 외부 API를 직접 식별할 수 있다는 장점을 가지지만, 개입과 탐지 가능성이라는 구조적 한계를 함께 가진다.

2.3 Execution-Flow-Based API Identification

실행 흐름 기반 API 식별은 API 후킹과 같은 직접 개입 없이 프로그램의 실제 실행 흐름이나 라이브러리 호출 흐름을 추적·관찰함으로써 외부 API 또는 라이브러리 호출을 식별하는 접근이다[3]. 대표적으로 Ether는 게스트 운영체제 외부에서 실행을 추적하는 방법을 제시하였고[16],

IntroLib은 가상머신 외부에서 사용자 모드 라이브러리 호출을 추적하는 접근을 제안하였다[17]. 또한 VMP는 가상머신 모니터(VMM, Virtual Machine Monitor)와 가상 머신 관찰 기법(VMI, Virtual Machine Introspection)을 이용하여 실행 중 호출된 외부 API와 그 인자 및 반환값을 기록하는 방법을 제시하였다[18]. 이러한 접근은 API 후킹 없이도 실행 중 호출 정보를 수집할 수 있다는 점에서 의미가 있으나, 대체로 하드웨어 가상화와 VMM/VMI와 같은 별도의 분석 인프라에 의존하므로 일반 Windows 환경에 직접 적용이 가능한 경량 프레임워크로 사용하기에는 제약이 있다[17][19]. 따라서 별도의 가상화 기반 분석 환경에 의존하지 않으면서도 실행 과정에 대한 직접 개입을 최소화할 수 있는 외부 API 식별 방법이 필요하다.

2.4 Hardware-Assisted Execution-Flow Analysis

하드웨어 지원 실행 흐름 분석을 활용한 기존 연구는 크게 세 가지 방향으로 구분될 수 있다.

첫째, PT를 이용하여 정밀한 실행 흐름이나 API 호출 순서를 복원하려는 연구가 있다. SeqTrace는 PT와 VMI를 결합하여 API 호출 순서를 추적하고 의미 정보를 복원하는 방법을 제안하였다[20].

둘째, LBR을 이용하여 특정 보안 이벤트를 감시하는 연구가 있다. 예를 들어 kBouncer는 LBR을 이용하여 민감한 시스템 호출 직전의 간접 분기 이력을 확인함으로써 ROP(Return-Oriented Programming) 공격 징후를 탐지하는 방법을 제시하였다[21].

셋째, LBR과 BTS를 함께 활용하여 일반적인 실행 분석과 제어 흐름 재구성을 수행하는 연구가 있다. LibIHT는 LBR과 BTS를 이용하여 실행 추적과 제어 흐름 그래프 재구성을 수행하는 프레임워크를 제안하였다[22].

이러한 연구들은 하드웨어 기반 분기 추적이 낮은 개입 수준으로 실행 흐름을 관찰할 수 있음을 보여준다[20-22]. 그러나 기존 연구의 초점은 API 호출 순서 복원, 특정 분기 이벤트 감시, 또는 일반적인 실행 분석과 제어 흐름 재구성에 맞추어져 있었다[20-22]. 즉, 하드웨어 기반 분기 정보를 실행 흐름 관찰이나 보안 이벤트 감시에 활용한 사례는 있었지만, 이를 실행 중 로드된 모듈 정보 및 모듈의 함수 정보와 결합하여 외부 API를 직접 식별하려는 연구는 제한적이었다. 따라서 하드웨어 기반 분기 추적에서 얻은 분기 정보를 실제 외부 API 식별 문제로 연결하는 접근이 필요하다. 특히 실행 중에 관측된 분기 목적지 주소를 로드된 모듈 정보와 함수 정보에 대응시켜 해석하는 과정이 함께 필요하다.

III. The Proposed Framework

본 장에서는 LBR 기반 분기 로그를 활용하여 프로그램 실행 중 호출된 외부 API를 식별하는 전체 절차를 설명한다. LBR은 실행 과정에 직접 개입하지 않고 분기의 시작 주소와 목적지 주소를 확보할 수 있지만, 해당 주소 정보만으로는 호출된 외부 API를 곧바로 판단할 수 없다. 따라서 제안 프레임워크는 분기 로그, 실행 중 로드된 모듈 정보, 외부 DLL의 함수 정보를 단계적으로 결합하여 외부 API 식별 결과를 생성한다.

이어지는 절에서 전체 처리 흐름, 분기 로그 및 모듈 정보 수집, 외부 API 분기 후보 추출, DLL 함수 매핑, 그리고 설계상 고려 사항과 한계를 순서대로 설명한다.

1. Framework Overview

이 절에서는 제안 프레임워크의 전체 처리 흐름을 설명한다. 전체 흐름은 Fig. 1에서와 같이 데이터 수집, 데이터 처리, 결과 저장의 세 단계로 구성된다. 먼저 데이터 수집 단계에서는 대상 프로그램이 실행되는 동안 커널 모드의 드라이버를 통해 LBR 분기 정보를 수집하고, 동시에 실행 중 로드되는 모듈 정보를 추적한다. 사용자 모드 수집 프로그램은 커널 드라이버에서 전달된 분기 로그와 모듈 정보를 후속 분석에 활용할 수 있는 형태로 정리하여 저장한다. 이 단계의 결과는 프로그램 실행 중 관찰된 저수준 분기 정보와 로드 모듈 정보가 포함된 추적 로그이다.

이후 데이터 처리 단계에서는 추적 로그로부터 대상 프로그램과 외부 DLL 사이에서 발생한 분기만을 선별하여 외부 API 분기 후보를 추출한다. 이어서 외부 DLL의 함수 정보를 이용하여 선별된 분기 목적지 주소를 실제 함수 정보와 대응시키고, 이를 통해 외부 API를 식별한다.

최종적으로 식별된 결과는 관련 분기 정보 및 함수 정보와 함께 CSV 파일로 저장된다. 즉, 제안 프레임워크는 실행 중 수집한 분기 로그, 로드된 모듈 정보, 그리고 DLL 함수 정보를 단계적으로 결합함으로써 최종적인 외부 API 식별 결과를 생성한다.

2. Branch Logs and Module Information

외부 API 식별의 첫 단계는 프로그램 실행 중에 발생하는 분기 정보와 해당 분기의 해석에 필요한 모듈 정보를 함께 수집하는 것이다. LBR은 실행 중에 발생하는 여러 형태의 분기에 대해서 시작 주소와 목적지 주소를 제공한다. 하지만, 해당 주소가 어떤 모듈에 속하며 실제로 어떤 외부 함수와 관련되는지는 제공되는 정보만으로 식별할

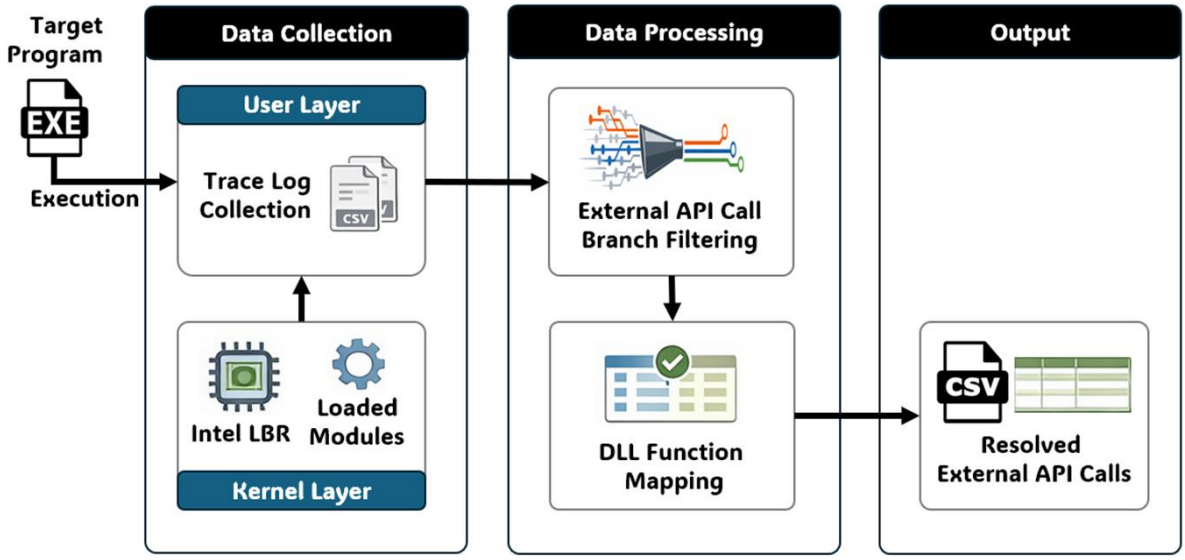

Fig. 1. Overview of the proposed pipeline

수 없다. 따라서 본 논문에서는 분기 로그와 함께 실행 중 로드되는 모듈 정보를 동시에 수집하여, 이후 단계에서 분기 목적지를 외부 DLL의 함수 정보와 연결할 수 있도록 한다.

대상 프로그램이 실행되면 LBR을 이용하여 분기 정보를 지속적으로 수집하고, 동시에 대상 프로세스와 실행 중 로드되는 외부 DLL의 베이스 주소, 크기, 경로와 같은 모듈 정보를 추적한다. 분기 정보에는 분기 시작 주소와 목적지 주소가 포함되어 기록되며, 모듈 정보에는 로드된 모듈의 베이스 주소와 크기가 기록된다. 이 두 정보를 결합하면 각 분기의 주소를 어느 모듈 범위에 대응시킬 수 있는지 판별할 수 있게 된다.

또한 사용자 모드의 수집 프로그램에서는 커널에서 수집된 분기 로그와 모듈 정보를 전달받아 이를 관리하고, 이후 처리 단계에서 사용할 수 있는 형태의 추적 로그로 저장한다. 따라서 이 과정은 단순한 분기 기록 수집이 아니라, LBR이 제공하는 주소 중심의 분기 정보를 외부 API 식별이 가능한 분석 대상으로 확장하기 위한 기반 데이터를 생성하는 과정이다.

2.1 Kernel-Mode Collection of Branch Logs

분기 로그 수집에는 CPU가 최근에 수행된 분기의 시작 주소와 목적지 주소를 기록하는 기능인 LBR을 사용한다. LBR은 최근 분기 항목을 스택 형태로 유지하며, 각 항목은 Table 1과 같이 구성된다.

Table 1. Branch information in kernel mode

Field	Description
From	Source address of a branch
To	Destination address of a branch

LBR의 이러한 특성은 최근 분기 정보를 주소 수준에서 직접 확보할 수 있게 해준다. 또한 시작 주소와 목적지 주소를 이용하면 외부 DLL로 향하는 분기를 식별할 수 있으므로, 외부 API 식별의 핵심 입력으로 사용된다. Fig. 2는 LBR 기반의 분기 로그 수집 과정을 보여준다.

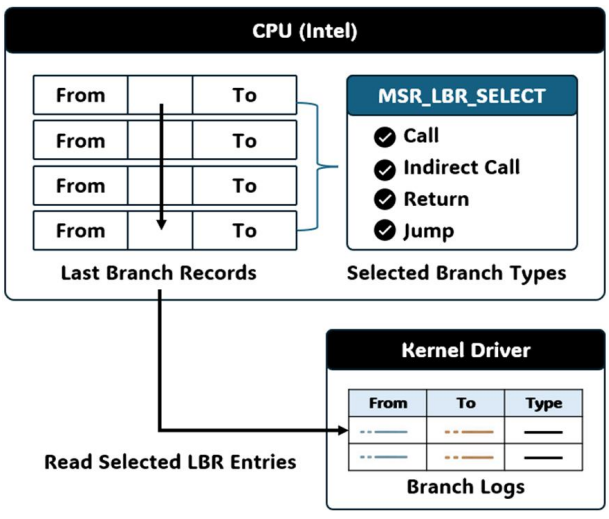

Fig. 2. Intel LBR-based collection of branch logs

LBR의 사용을 위해서는 CPU의 세부 기능을 제어하거나 상태를 읽기 위해 프로세서가 제공하는 레지스터 집합인 MSR(Model-Specific Register)에 접근해야 하는데,

이를 위해서는 커널 수준의 접근 권한이 필요하다. 따라서 본 논문에서는 커널 드라이버에서 MSR 레지스터들 중 하나인 MSR_LBR_SELECT 레지스터를 이용해 기록할 분기 타입을 설정하고, CPU가 유지하는 최근 분기 항목을 읽어 분기 로그로 저장하도록 구성하였다.

LBR은 동일 코어에서 실행된 분기들만 저장되는 특성이 있기 때문에 Fig. 3과 같이 대상 프로그램을 동일한 특정 코어에서만 작동하도록 고정한다.

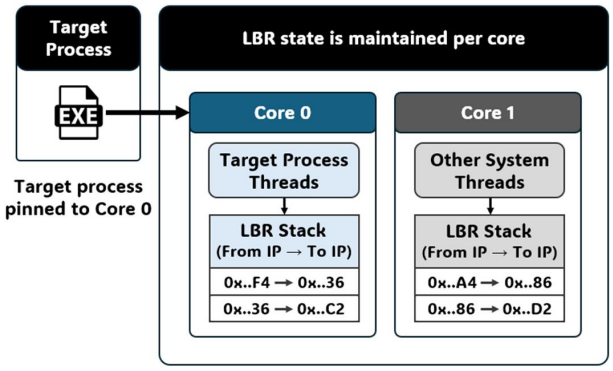

Fig. 3. Per-core LBR and core-pinned execution

또한, 제한된 깊이의 LBR 정보를 안정적으로 확보하기 위해서 PMI(Performance Monitoring Interrupt) 기반의 수집 경로를 사용한다. PMI는 CPU의 성능 모니터링 기능인 PMU(Performance Monitoring Unit)에서 설정한 특정 계수기가 임계값에 도달했을 때 발생하는 인터럽트로, LBR과 같이 코어별로 작동한다. 본 논문에서는 실행된 인스트럭션 수를 계수하는 카운터를 사용한다. 지정된 개수의 인스트럭션이 수행될 때마다 인터럽트가 발생하면 커널 드라이버는 LBR의 내용을 읽어 분기 로그로 저장한다. 이를 통해 LBR 정보를 주기적으로 확보하여 누락을 줄이고 안정적으로 분기 정보를 수집할 수 있다. Fig. 4는 PMI 기반의 LBR 로그 수집을 보여준다.

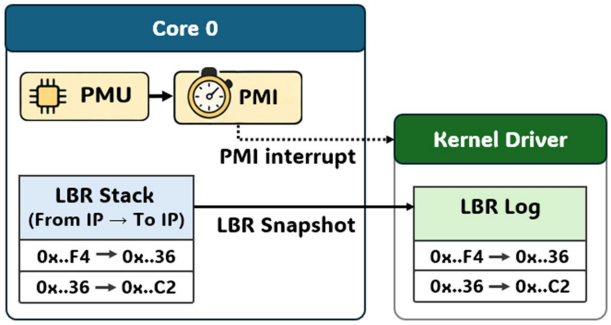

Fig. 4. PMI-based collection of LBR records

2.2 Kernel-Mode Tracking of Loaded Modules

LBR을 통해 수집한 분기 로그에는 분기의 시작 주소와 목적지 주소가 포함되지만, 해당 주소가 어떤 모듈에 속하는지는 제공되지 않는다. 따라서 외부 API를 식별하기 위해서는 관측된 분기 목적지 주소가 대상 프로세스 내부에 속하는지, 아니면 외부 DLL 내부에 속하는지를 먼저 판별할 필요가 있다. 따라서 본 논문에서는 Table 2와 같이 분기 로그 수집과 함께 실행 중 로드되는 모듈 정보를 동시에 추적하여, 이후 단계에서 분기 목적지 주소를 실제 로드된 모듈과 대응시킬 수 있도록 한다.

Table 2. Module information in kernel mode

Field	Description
Base	Base address of a loaded module
Size	Image size of a loaded module
Path	Full path of a loaded module

프로그램 실행 중에 단순히 외부에서 조회하는 것은 특정 시점의 모듈 상태만 제공하므로, 실행 중 변화하는 모듈 로드 상태를 누락 없이 반영하기 어렵다. 따라서 본 논문에서는 실행 과정에서의 직접 개입을 최소화하면서도, 이후 분기 목적지 주소를 유효한 모듈 범위와 안정적으로 대응시키는 데 필요한 모듈 정보를 확보하기 위해서 커널 모드에서 정보를 수집한다.

이를 위해 커널 드라이버는 대상 프로세스의 생성 및 종료, 그리고 실행 중에 발생하는 이미지 로드 이벤트를 감시하여 모듈 상태를 추적한다. 대상 프로세스나 외부 DLL이 로드될 때마다 커널 드라이버는 해당 모듈의 베이스 주소, 이미지 크기, 모듈 경로와 같은 메타데이터를 수집하고, 이를 프로세스별 모듈 테이블에 반영한다.

이와 같이 수집된 모듈 테이블은 분기 로그와 결합되어 각 분기의 시작 주소와 목적지 주소가 어느 모듈의 주소 범위에 속하는지를 판별하는 기준으로 사용된다. 이를 통해 프로세스 내부의 분기와 외부 DLL로 향하는 분기를 구분할 수 있으며, 외부 DLL로 판단된 분기 목적지 주소를 이후 DLL의 함수 정보와 연결하기 위한 기초 데이터로 활용할 수 있다. 따라서 모듈 추적 과정은 단순한 부가 정보 수집이 아니라, LBR이 제공하는 주소 중심의 분기 로그를 외부 API 식별이 가능한 분석 대상으로 변환하기 위한 필수 단계이다.

2.3 User-Mode Trace Management

커널 드라이버에서 수집된 분기 로그와 모듈 정보는 사용자 모드의 수집 프로그램으로 전달되며, 수집 프로그램은 이를 이후 분석에 사용할 수 있도록 하나의 결과 파일로 저장한다. 이 저장 결과는 이후 단계에서 분기 목적지 주소를 모듈 범위와 대응시키고, 외부 API 후보를 추출하기 위한 입력으로 사용된다.

이 단계에서 저장되는 결과는 최종적인 외부 API 식별 결과가 아니라, 이후 DLL 함수 정보 대응을 포함한 후속 분석을 위한 입력 데이터이다. 따라서 사용자 모드의 데이터 관리 과정은 커널 드라이버에서 수집된 실행 정보를 후속 분석에 활용할 수 있는 형태로 정리하는 단계라고 볼 수 있다. 제안 프레임워크는 수집 단계와 해석 단계를 분리하여 처리한다. 수집 단계에서는 분기 주소와 모듈 정보를 중심으로 추적 로그를 저장하고, DLL 함수 매핑과 진입 유형 판별은 실행 종료 후 후처리 단계에서 수행한다. 이를 통해 실행 중 처리 부담을 줄이고, 동일한 로그를 이용한 재분석이 가능하도록 하였다.

3. External API Call Branch Filtering

LBR을 통해 수집된 분기 로그에는 분기의 시작 주소와 목적지 주소가 포함되지만, 저장된 모든 분기가 외부 API 호출과 직접적으로 관련되는 것은 아니다. 따라서 본 단계에서는 대상 프로세스 내부에서 외부 DLL로 향하는 분기만을 선별하여 외부 API 후보 분기를 추출한다.

후보 추출은 각 분기의 시작 주소와 목적지 주소를 대상으로 수행한다. 분기 시작 주소가 대상 프로세스의 주소 범위에 속하고, 분기 목적지 주소가 외부 DLL의 주소 범위에 속하는 경우, 해당 분기를 외부 API 후보로 판단한다. 반대로 시작 주소와 목적지 주소가 모두 대상 프로세스 내부에 속하는 경우에는 내부 제어 흐름으로 간주하며, 외부 DLL의 주소 범위와 무관한 분기 역시 후보에서 제외한다. 이를 통해 외부 기능 호출과 직접적으로 관련될 가능성이 높은 분기만을 선별할 수 있다.

이 단계에서 추출되는 결과는 최종적인 외부 API 식별 결과가 아니라, 외부 DLL로 향하는 분기 후보의 집합이다. 즉, 본 단계는 분기 목적지 주소를 주소 수준에서 먼저 정제하는 역할을 수행하며, 이후 단계에서 후보 분기의 목적지 주소를 DLL 함수 정보와 대응시켜 실제 외부 API를 식별한다. 따라서 외부 API 후보 추출 과정은 분기 로그와 모듈 정보를 결합하여, 최종적인 API 식별에 앞서 분석 대상을 정제하는 단계라고 볼 수 있다.

4. DLL Function Mapping

외부 DLL로 향하는 분기만 선별한 결과만으로는, 분기 목적지 주소가 외부 DLL의 어떤 함수와 관련되는지까지는 알 수 없다. 따라서 본 단계에서는 Fig. 5와 같이 실행 중 로드된 외부 DLL로부터 함수 정보를 추출하고, 이를 분기 로그 및 모듈 정보와 결합하여 최종적으로 어떤 DLL의 어떤 함수에 해당하는지를 식별한다.

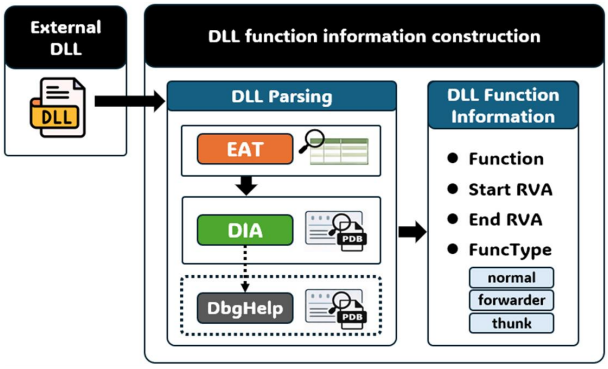

Fig. 5. Construction of DLL function information

DLL의 함수 정보는 먼저 EAT를 통해 DLL이 외부에 공개하는 함수 이름과 함수 시작 RVA를 수집하여 기본 항목으로 사용한다. 이 과정에서 다른 DLL의 함수로 연결되는 항목은 'forwarder'로 판단하여 별도로 기록한다. 이후에는 심볼 정보와 함수 범위 구성 결과를 함께 사용하여, 해당 항목이 일반 함수인지 또는 함수 진입 전 짧은 중계 코드나 간접 분기를 거치는 'thunk' 항목인지 구분한다. 이와 같이 각 항목의 함수 유형은 일반 함수, 'forwarder', 'thunk'로 구분하여 기록한다. 다만 실행 중 관측된 분기 목적지 주소가 항상 함수 진입점과 정확히 일치하는 것이 아니기 때문에 EAT 정보만으로 함수를 정확하게 식별하는 것은 한계가 있다.

본 논문에서는 이런 한계를 보완하기 위해 DIA(Debug Interface Access)와 DbgHelp를 함께 사용한다. DIA는 PDB(Program Database)에 저장된 디버그 심볼 정보에 접근하여 함수 이름, 시작 RVA, 길이와 같은 정보를 얻기 위한 라이브러리이며, DbgHelp는 Windows에서 심볼 로딩과 조회를 지원하는 디버깅 보조 라이브러리이다. 먼저 DIA를 통해 함수로 기록된 심볼과 이름 및 주소가 제공되는 공개 심볼을 수집한다. 함수로 기록된 심볼에 대해서는 가능한 경우 함수 길이 정보를 이용하여 함수 범위를 직접 구성하고, 길이 정보가 없는 경우에는 인접한 심볼 경계를 이용하여 가능한 함수 범위를 추정한다. 또한 DIA만으로 충분한 정보를 얻기 어려운 경우에는 DbgHelp를 이용하

여 심볼 정보를 추가로 수집하고, 이를 함수 시작점과 범위 구성에 보조적으로 활용한다.

이후 외부 DLL로 향하는 분기 후보의 목적지 주소는 해당 DLL의 베이스 주소를 기준으로 RVA로 변환되며, 변환된 목적지 RVA는 구성된 함수 정보와 비교된다. 목적지 RVA가 특정 함수의 시작 RVA와 동일한 경우에는 해당 분기를 그 DLL 함수에 대한 정상 진입으로 판단한다. 반면 목적지 RVA가 함수 시작 RVA와 일치하지 않더라도 함수 범위 내부에 포함되는 경우에는 중간 진입으로 식별한다. 이 경우에는 함수 시작 RVA로부터의 오프셋도 함께 기록하여, 분기 목적지 주소가 함수 내부 어느 위치에 해당하는지를 나타낸다. 이를 통해 함수 진입점과 정확히 일치하는 경우뿐 아니라, 함수 범위 내부의 중간 지점으로 분기하는 경우에도 식별할 수 있다.

최종 식별 결과에는 분기 시작 주소와 목적지 주소, DLL 이름과 함수 이름, 함수 정보 유형, 진입 유형, 그리고 함수 시작점 기준 오프셋이 포함되며, 주요 필드는 Table 3과 같다.

Table 3. Resolved external API call information

Field	Description
From	Source branch address
To	Destination branch address
DLL	Matched external DLL name
Function	Matched function name
FuncType	Matched function type
EntryType	Normal entry or middle entry
Offset	Offset from the function start

Algorithm 1은 LBR 분기 로그로부터 외부 API 식별 결과를 생성하는 핵심 절차를 나타낸다.

Algorithm 1 External API Identification

Require: Branch records B, modules M, function map F
Ensure: Resolved API set R

$R \leftarrow \emptyset$
for all $b \in B$ **do**
  $s \leftarrow \text{MOD}(M, b.from)$
  $d \leftarrow \text{MOD}(M, b.to)$
  if s is target **and** d is external **then**
    $r \leftarrow b.to - d.base$
    $f \leftarrow \text{FUNC}(F[d], r)$
    if $f \neq \emptyset$ **then**
      if $r = f.start$ **then**
        $e \leftarrow normal$
        $o \leftarrow 0$
      else
        $e \leftarrow middle$
        $o \leftarrow r - f.start$
      end if
      $R \leftarrow R \cup \{(b, d, f, e, o)\}$
    end if
  end if
end for
return R

여기서 s와 d는 각각 분기 시작 주소와 목적지 주소가 속한 모듈을 의미하며, r은 목적지 주소의 DLL 기준 RVA, f는 대응되는 DLL 함수, e와 o는 각각 진입 유형과 함수 시작점 기준 오프셋을 의미한다.

본 단계에서는 이러한 외부 API 식별 결과를 결과 파일로 저장한다. 이렇게 저장된 결과는 실행이 종료된 이후에도 별도의 재수집 없이 외부 API 호출 내역을 확인하고 비교할 수 있도록 하며, 후속 분석과 검증의 입력으로 활용될 수 있다.

5. Design Considerations and Limitations

본 논문에서는 실행 흐름에 직접 개입하지 않고 LBR 기반 분기 로그와 외부 DLL 정보를 결합하여 외부 API를 식별하도록 설계하였다. 이러한 설계는 API 후킹과 같이 사전에 정의한 함수 진입점을 기준으로 개입하는 방식에 비해, 실행 과정에 대한 직접 개입을 최소화하면서 실제 수행된 분기 정보를 확보할 수 있도록 한다. 또한 DLL 함수와의 매핑 단계에서는 함수 진입점의 정확한 일치뿐 아니라 함수 범위 내부의 중간 진입까지 함께 고려함으로써, 함수 진입점 중심의 관찰 방식보다 더 넓은 식별 범위를 제공한다.

반면 LBR은 최근 분기만을 제한된 깊이로 저장하므로, 실행 중 발생하는 모든 외부 분기를 단일 실행에서 항상 완전하게 기록하지는 못한다. 따라서 분기 로그에서 외부 분기 정보의 누락 가능성이 의심되는 경우에는 반복 실행을 통해 이를 보완적으로 확인할 필요가 있다. 다만 반복 실행 시 식별 결과의 일관성을 정량적으로 평가하기 위해서는 동일한 실행 조건, PMI 수집 주기, 코어 고정 조건, 로그 저장 정책 등을 통제해야 하므로, 이는 본 논문의 기본 식별 가능성 평가와는 구분되는 확장 평가 항목이다. 또한 최종 결과는 실행 중 관측된 분기 정보에 기반하므로, 실제로 실행되지 않은 코드에 포함된 외부 API는 식별되지 않는다. DLL 함수 매핑의 정확도 역시 EAT와 심볼 정보의 가용성에 영향을 받기 때문에, 일부 분기는 특정 함수와 대응되지 않아 미해결 상태로 남을 수 있다. 본 논문은 외부 DLL 및 관련 함수의 식별에 초점을 두므로, 호출 인자, 반환값, 또는 호출의 의미적 의도까지 복원하는 것은 범위에 포함하지 않는다. 또한 본 논문은 제안 프레임워크의 외부 API 식별 가능성 검증에 초점을 두었으며, 하드웨어 추적 방식 간의 정량적 오버헤드 비교는 별도의 실험 범위로 남겨두었다. LBR 기반 수집은 대상 프로세스의 코어 고정, PMI 기반 수집 주기, 로그 저장 방식 등에 영향을 받기 때문에 일반 실행 환경과 단순 실행 시간만으

로 직접 비교하기 어렵다. 따라서 본 논문에서는 특정 하드웨어 추적 기술 대비 오버헤드 감소율을 주장하지 않고, 공정한 오버헤드 비교는 향후 연구 방향으로 정리하였다.

그럼에도 불구하고 제안하는 프레임워크는 실행 과정에 대한 직접 개입을 줄이면서, 실행 중 실제로 발생한 외부 API 호출 정보를 확보할 수 있는 실용적인 방법이라는 점에서 의의를 가진다. 특히 사전에 정의된 함수 진입점이나 감시 대상 API 집합에 의존하지 않고, 관측된 분기 목적지 주소를 바탕으로 외부 API를 식별할 수 있다는 점에서 기존 방식과 구별된다.

IV. Experiments and Evaluation

1. Experimental Setup

본 절에서는 제안 프레임워크의 외부 API 식별 가능성을 평가하기 위한 실험 환경, 샘플 구성, 그리고 실험 시나리오를 설명한다. 본 논문의 목적은 프로그램 실행 중 관측된 분기 정보를 바탕으로 실제 호출된 외부 API를 식별하는 것이므로, 실험 역시 단순히 실행 여부를 확인하는 수준이 아니라 분기 목적지가 어느 DLL의 어느 함수에 대응되는지, 그리고 해당 진입이 함수 시작점에 대한 정상 진입인지 또는 함수 범위 내부의 중간 진입인지까지 구분할 수 있는지를 중심으로 구성하였다.

1.1 Experimental Environment

실험은 Intel LBR을 지원하는 CPU가 설치된 64비트 Windows 환경에서 수행하였다. 대상 프로그램은 32비트 사용자 모드 실행 파일로 구성하였는데, 이는 64비트 운영체제에서 32비트 응용 프로그램이 함께 동작하는 일반적인 Windows 실행 환경을 반영하기 위한 것이다[23]. 또한 비교 실험의 구현과 구성을 단순화하기 위해 실험 샘플은 모두 32비트로 통일하였다. 반면 LBR 수집 드라이버, 분기 로그 수집 프로그램, 그리고 DLL 정보 추출 프로그램은 대상 프로그램 내부에 삽입되어 동작하는 구성요소가 아니라, 대상 프로그램과 독립적으로 실행되면서 분기 수집과 후처리를 담당하는 외부 분석 구성요소이다. 따라서 이들 도구는 호스트 운영체제 환경에 맞추어 64비트로 빌드하였다. 이때 커널 드라이버 빌드에는 WDK(Windows Driver Kit)를 사용하였고[24], DLL 정보 추출 프로그램에서는 함수 및 심볼 관련 정보 구성을 위해 DIA와 DbgHelp 라이브러리를 사용하였다[25][26]. 세부 환경은 Table 4와 Table 5에 정리하였다.

Table 4. Test hardware and OS environment

Item	Description
CPU	Intel Core i7-1360P 2.2GHz
Memory	32GB
Host OS	Microsoft Windows 11 64-Bit (OS build 26200.8037)

Table 5. Tool build setup

Program	Build setup
LBR driver	x64, VS 2022, WDK
LBR collector	x64, VS 2022
DLL extractor	x64, VS 2022, DIA and DbgHelp

1.2 Sample Configuration

실험 샘플은 제안 프레임워크의 외부 API 식별 가능성을 호출 형태별로 확인할 수 있도록 직접 구현한 프로그램들로 구성하였다. 모든 샘플은 Visual Studio 2022를 사용하여 32비트 사용자 모드 실행 파일로 빌드하였으며, Table 6과 같이 파일, 레지스트리, 메모리, 프로세스, 스레드 관련 Win32 API 10개를 포함하도록 구성하였다. 또한 Win32 API를 호출하는 대신 대응하는 네이티브 API를 직접 호출하는 시나리오에도 적합한 함수들로 구성하였다. 실험 샘플을 직접 구현한 이유는 호출 형태별 정답 정보를 명확히 통제하기 위해서이다. 본 논문의 평가는 단순히 외부 API가 관측되었는지를 확인하는 것이 아니라, 호출 방식, 외부 DLL 및 함수 이름, 함수 진입 유형, 함수 시작점 기준 오프셋과 같은 세부 식별 결과가 의도한 호출 구조와 일치하는지를 확인하는 데 목적이 있다. 일반 프로그램이나 실제 악성코드에서는 이러한 정답 정보를 완전하게 확보하기 어렵기 때문에, 본 연구에서는 정상 진입, 동적 로드, 네이티브 API 직접 호출, 함수 내부 진입과 같은 호출 형태를 명확히 구분할 수 있는 자체 구현 샘플을 사용하였다.

Table 6. Win32 API categories

Category	Functions
File	GetFileInformationByName, GetFileInformationByHandleEx,
Registry	RegOpenKeyExW, RegCreateKeyExW, RegQueryValueExW, RegSetValueExW
Memory	VirtualAlloc, VirtualQuery
Process	OpenProcess
Thread	OpenThread

기본 실험 샘플은 호출 형태에 따른 외부 API 식별 결과를 비교하기 위해서 Table 7과 같이 외부 API 호출 경로를 다르게 구성하였다.

Table 7. Basic sample configuration

Sample	Call Path
Normal	Caller → API entry
Dynamic	Caller → runtime resolution → API
Native	Caller → native API entry
Middle	Caller → API internal entry

추가 실험 샘플은 정적 정보가 제한되거나 직접 개입 기반 관측 방식이 영향을 받을 수 있는 조건에서의 외부 API 식별 가능성을 확인할 수 있도록 Table 8과 같이 구성하였다.

Table 8. Additional sample configuration

Sample	Description
Obfuscated	Obfuscated API name
Guarded	Anti-debug and integrity check

1.3 Tool Configuration

기본 실험은 호출 형태에 따른 외부 API 식별 결과를 비교하기 위해 수행하였다. 이를 위해 Visual Studio 2022에 포함된 IAT 기반 정적 도구인 DumpBin[27]과 API 후킹 도구인 WinAPIOverride[28]를 사용하였다. 정적 비교에서는 실행 파일의 IAT에 정적으로 등록된 외부 API 집합을 확인하였고, 동적 비교에서는 실행 중 API 호출을 관찰하였다. 이때 WinAPIOverride는 후킹을 위해 대상 API를 사전에 설정해야 하므로, Table 6의 API들을 감시 대상으로 설정하였다.

추가 실험의 비교 도구로는 범용 정적 분석 도구인 Ghidra[29]와 API 후킹 도구인 WinAPIOverride를 사용하였다. Ghidra는 문자열, 참조 정보, 디스어셈블을 통해 확인 가능한 정보를 분석하고, 필요시 디버거로 실행하면서 API 호출을 보조적으로 확인하는 데 사용하였다. 또한 WinAPIOverride는 동일한 조건에서 실행 중 API 호출을 관찰하는 데 사용하였다.

1.4 Experimental Scenarios

실험 시나리오는 기본 실험과 추가 실험으로 구분하여 구성하였다. 기본 실험은 호출 형태에 따른 외부 API 식별 결과를 비교하기 위한 것으로, Table 9와 같이 정상 진입 호출, 동적 로드 호출, 네이티브 API 직접 호출, 함수 내부 진입 호출의 네 가지 시나리오를 대상으로 하였다. 각 시나리오에서는 제안 프레임워크가 외부 DLL과 함수 이름을 식별할 수 있는지를 평가하였으며, 함수 내부 진입 호출의 경우에는 진입 유형과 함수 시작점 기준 오프셋도 함께 확인하였다.

Table 9. Basic experimental scenarios

No	Scenario	Evaluation focus
S1	Normal	Direct-entry identification
S2	Dynamic	Dynamic-load identification
S3	Native	Native-direct-call identification
S4	Middle	Middle-entry identification

추가 실험은 함수 이름 난독화나 직접 개입 탐지와 같이 정적 정보가 부족하거나 후킹 기반 관찰이 제한될 수 있는 조건에서 외부 API 식별 결과를 비교하기 위한 것이다. 이를 위해 Table 10과 같이 함수 이름 난독화가 적용된 동적 로드 호출과 직접 개입 탐지가 적용된 동적 로드 호출의 두 가지 시나리오를 구성하였다.

Table 10. Additional experimental scenarios

No	Scenario	Evaluation focus
S5	Obfuscated	Name-obfuscation condition
S6	Guarded	Anti-analysis condition

2. Evaluation Metrics

기본 실험에서는 각 시나리오에서 의도적으로 호출한 외부 API 집합을 기준으로 식별 결과를 평가하였다. 공통 비교 지표는 API 식별률(API Identification Rate, AIR)로 정의하였으며, 이는 정답 외부 API 수 대비 올바르게 식별된 외부 API 수의 비율로 계산하였다. 본 논문의 실험 샘플은 주요 Win32 API 10개를 기준으로 구성하였으므로, 각 시나리오에서 전체 API 중 몇 개를 정확히 식별하였는지를 기준으로 AIR을 산출하였다.

$$AIR(\%) = \frac{Correctly\ \ identified\ \ APIs}{Target\ \ APIs} \times 100$$

여기서 올바른 식별은 실행 중 관측된 외부 분기 결과가 정답 DLL과 함수 이름에 정확히 대응되는 경우로 정의하였다. 또한 함수 내부 진입 호출 시나리오에서는 식별된 함수 이름뿐 아니라 진입 유형과 함수 시작점 기준 오프셋이 의도한 위치와 일치하는지도 함께 확인하였다. 이 지표는 제안 프레임워크, IAT 기반 정적 도구, 그리고 API 후킹 도구의 결과를 동일한 기준에서 비교하기 위해 사용하였다.

추가 실험에서는 함수 이름 난독화나 직접 개입 탐지가 적용된 동적 로드 호출을 대상으로 하였으며, AIR 중심의 정량 비교가 아닌 범용 정적 분석 도구의 확인 가능 여부, API 후킹 도구의 관찰 가능 여부, 그리고 제안 프레임워크의 식별 결과를 평가하였다.

한편, 제안 프레임워크는 DLL 및 함수 이름과 함께 진입 유형과 함수 시작점 기준 오프셋도 기록한다. 따라서

중간 진입 호출 시나리오에 대해서는 API 이름 식별 여부와 함께 진입 유형의 일치 여부와 오프셋 기록의 적절성도 추가로 확인하였다. 다만 이 항목들은 IAT 기반 정적 도구나 API 후킹 도구의 결과와 직접적으로 대응되지 않으므로, 공통 비교 지표가 아니라 제안 방식의 세부 식별 능력을 확인하기 위한 보조 평가 항목으로 사용하였다.

3. Experimental Results and Analysis

본 절에서는 기본 실험과 추가 실험의 결과를 구분하여 제시한다. 기본 실험에서는 대표적인 호출 형태에 따른 외부 API 식별 결과를 API 식별률을 기준으로 비교하였고, 추가 실험에서는 함수 이름 난독화 및 직접 개입 탐지가 적용된 동적 로드 변형 조건에서 각 도구의 확인·관찰·식별 가능 여부를 비교하였다. 또한 제안 프레임워크가 제공하는 진입 유형과 오프셋 정보는 별도의 보조 결과로 정리하였다.

3.1 Results on Basic Experimental Scenarios

기본 실험 시나리오에 대한 IAT 기반 정적 도구, API 후킹 도구, 그리고 제안 프레임워크의 식별 결과는 Table 11과 같다. 기본 실험 결과는 API 식별률(AIR)을 기준으로 비교하였으며, 정적 비교는 IAT에 정적으로 드러난 외부 API 노출 여부를 기준으로 비교하였다. 이를 통해 각 시나리오에서의 호출 형태에 따라 정적 확인, 후킹 기반 관찰, 그리고 제안 프레임워크의 식별 결과가 어떻게 달라지는지를 비교하였다.

Table 11. Results in basic scenarios

Scenario	IAT-based Static tool	API Hooking tool	Proposed framework
Normal	10 (100%)	10 (100%)	10 (100%)
Dynamic	0 (0%)	10 (100%)	10 (100%)
Native	0 (0%)	0 (0%)	10 (100%)
Middle	0 (0%)	0 (0%)	10 (100%)

정상 진입 호출 시나리오에서는 대상 Win32 API가 IAT에 등록되어 있으므로, IAT 기반 정적 도구를 통해 해당 API의 정적 노출 여부를 확인할 수 있었다. 또한 API 후킹 도구와 제안 프레임워크도 모두 이들 API를 식별하였다. 반면 동적 로드 호출 시나리오에서는 실행 중 API 주소를 획득하여 호출하므로, IAT 기반 정적 도구로는 확인하지 못하였다. 그러나 API 후킹 도구와 제안 프레임워크는 모두 실행 중 호출을 식별할 수 있었다.

네이티브 API 직접 호출 시나리오에서는 Win32 API 호출을 우회하기 위해서 대응하는 네이티브 API를 직접 호출하도록 구성하였기 때문에, 사전에 정의된 Win32 API 진입점을 감시하는 API 후킹 도구는 이를 식별하지 못하였다. 또한 함수 내부 진입 호출 시나리오에서도 API 후킹 도구는 함수 시작점이 아닌 내부 지점으로의 직접 진입을 식별하지 못하였다. 반면 제안 프레임워크는 실행 중에 발생한 외부 분기 주소를 DLL 및 함수 정보와 대응시킴으로써 두 시나리오에서 모두 외부 API를 식별할 수 있었다. 이러한 결과는 제안 프레임워크가 사전에 정의된 감시 대상 API 집합이나 함수 진입점에 의존하지 않고, 실제 실행 결과를 기반으로 외부 API를 식별한다는 점을 보여준다.

3.2 Results on Additional Experimental Scenarios

추가 실험은 함수 이름 난독화와 직접 개입 탐지가 적용된 동적 로드 호출을 대상으로 수행하였다. 기본 실험의 AIR 중심 정량 비교와 달리 추가 실험에서는 범용 정적 분석 도구의 확인 가능 여부, API 후킹 도구의 관찰 가능 여부, 그리고 제안 프레임워크의 식별 결과를 중심으로 비교하였다. 실험 결과는 Table 12와 같다.

Table 12. Results in additional scenarios

Method	Obfuscated	Guarded
Generic static analysis	X	X
API hooking	O	X
Proposed framework	O	O

먼저 함수 이름 난독화가 적용된 Obfuscated 시나리오에서는 호출 대상 API 이름이 실행 파일에 평문 형태로 드러나지 않으므로, 정적 분석만으로는 실제 호출 대상을 직접 확인하기 어려웠다. 반면 실행 중에는 API 주소가 복원된 뒤 정상적으로 호출되므로, API 후킹 도구는 해당 호출을 관찰할 수 있었고, 제안 프레임워크 역시 실행 중 관측된 외부 분기 주소를 기반으로 외부 API를 식별할 수 있었다. 이는 제안 프레임워크가 정적 문자열 노출 여부에 의존하지 않고, 실제 실행 중에 발생한 외부 분기를 바탕으로 식별 결과를 생성할 수 있음을 보여준다.

다음으로 직접 개입 탐지가 적용된 Guarded 시나리오에서는 정적 분석만으로 실행 시점에 실제로 호출되는 API를 확정하기 어려웠고, 디버깅을 통한 동적 확인이나 API 후킹 도구에 의한 관찰은 탐지되어 실행이 중단되었다. 반면 제안 프레임워크는 실행 흐름에 대한 직접 개입 없이 분기 로그를 수집한 뒤 후처리 단계에서 외부 API를 식별하므로, 해당 조건에서도 식별 결과를 생성할 수 있었다. 이러한 결과는 제안 방식이 정적 노출이 제한되거나 직접 개입 기반 관찰이 영향을 받을 수 있는 조건에서도

적용이 가능함을 보여준다.

3.3 Identification Details

제안 프레임워크는 DLL 이름과 함수 이름뿐 아니라 진입 유형과 함수 시작점 기준 오프셋도 함께 기록하며, 그 결과는 Table 13과 같다.

Table 13. Identification details

Scenario	Entry type	Offset
Normal	normal	-
Dynamic	normal	-
Native	normal	-
Middle	middle	all correct

정상 진입 호출, 동적 로드 호출, 네이티브 API 직접 호출 시나리오에서는 모두 함수 시작점으로 진입하므로 진입 유형이 normal로 기록되었고 오프셋은 나타나지 않았다. 반면 함수 내부 진입 호출 시나리오에서는 진입 유형이 middle로 기록되었으며, 함수 시작점 기준 오프셋도 의도한 위치에 맞게 산출되었다. 이는 제안 프레임워크가 외부 API 이름 식별뿐 아니라 함수 내부 진입 여부까지 함께 구분할 수 있음을 보여준다.

V. Conclusions

본 논문에서는 Windows 환경에서 실행 중 호출된 외부 API를 식별하기 위한 하드웨어 분기 추적 기반 프레임워크를 제안하였다. 제안 프레임워크는 Intel LBR로 분기 로그를 수집하고, 후처리 단계에서 외부 모듈로 향하는 분기를 선별한 뒤, 실행 중 로드된 DLL의 함수 정보와 대응시켜 외부 API 식별 결과를 생성한다. 이 과정에서 제안 프레임워크는 IAT 기반 정적 정보나 사전에 정의된 감시 대상 API 및 함수 진입점을 전제로 하지 않고 외부 API 식별 결과를 생성하도록 설계하였다.

실험 결과, 제안 프레임워크는 정상 진입 호출, 동적 로드 호출, 네이티브 API 직접 호출, 함수 내부 진입 호출, 그리고 함수 이름 난독화 및 직접 개입 탐지가 적용된 시나리오에서도 외부 API를 식별할 수 있었다. 특히 IAT 기반 정적 도구나 API 후킹 도구로는 식별이 어려운 경우에도 식별 결과를 생성할 수 있었으며, 함수 내부 진입 호출에 대해서는 진입 유형과 함수 시작점 기준 오프셋까지 함께 제공할 수 있음을 확인하였다. 이는 제안 방식이 실행 과정에 대한 직접 개입을 최소화하면서도 실제 호출된 외부 API를 식별할 수 있는 실용적인 방법임을 보여준다.

다만 본 논문의 실험은 호출 구조와 정답 정보를 통제할 수 있는 자체 구현 샘플을 대상으로 수행되었다. 이는 호출 방식, 함수 진입 유형, 함수 시작점 기준 오프셋과 같은 세부 정답 정보를 기준으로 제안 프레임워크의 식별 결과를 평가하기 위한 선택이다. 일반 프로그램이나 실제 악성코드에서는 이러한 정답 정보를 완전하게 확보하기 어렵기 때문에, 실제 샘플 기반 검증을 위해서는 별도의 정답 생성 방법과 재현 가능한 실행 환경이 함께 필요하다. 또한 제안 방식은 LBR의 제한된 기록 깊이와 수집 시점, PMI 수집 주기, 코어 고정 조건, 로그 저장 방식, 그리고 DLL 함수 정보와 심볼 정보의 가용성에 영향을 받을 수 있다. 따라서 반복 실행 시 식별 결과의 일관성, 수집 오버헤드, 실제 프로그램 및 악성코드 대상 확장 검증은 향후 연구에서 별도의 통제된 실험 설계를 통해 평가할 필요가 있다. 향후에는 자동화 분석 환경과 연동하여 LBR 로그 수집, 모듈 정보 저장, DLL 함수 매핑, API 식별 리포트 생성을 하나의 분석 파이프라인으로 구성하고, 실제 샘플과 다양한 실행 환경을 대상으로 검증 범위를 확장할 계획이다.

ACKNOWLEDGEMENT

This work was supported by a Korea University Grant. This work was also supported by the IITP(Institute of Information & Communications Technology Planning & Evaluation)-ITRC(Information Technology Research Center) grant funded by the Korea government(Ministry of Science and ICT) (IITP-2026-RS-2022-00164800).

REFERENCES

[1] Microsoft, "Run-Time Dynamic Linking," Microsoft Learn, *https://learn.microsoft.com/en-us/windows/win32/dlls/run-time-dynamic-linking*, Accessed: May 5, 2026.

[2] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A Survey on Automated Dynamic Malware-Analysis Techniques and Tools," ACM Computing Surveys, Vol. 44, No. 2, Article 6, pp. 1-42, Feb. 2012. DOI: 10.1145/2089125.2089126

[3] D. C. D'Elia, S. Nicchi, M. Mariani, M. Marini, and F. Palmaro, "Designing Robust API Monitoring Solutions," IEEE Transactions on Dependable and Secure Computing, Vol. 20, No. 1, pp. 392-406, Jan.-Feb. 2023. DOI: 10.1109/TDSC.2021.3133729

[4] A. Afianian, S. Niksefat, B. Sadeghiyan, and D. Baptiste, "Malware Dynamic Analysis Evasion Techniques: A Survey," ACM Computing Surveys, Vol. 52, No. 6, Article 126, pp. 1-28, Nov. 2019. DOI: 10.1145/3365001

[5] Microsoft, "PE Format," Microsoft Learn, *https://learn.microsoft.com/en-us/windows/win32/debug/pe-format*, Accessed: May 5, 2026.

[6] J. Lopez, L. Babun, H. Aksu, and A. S. Uluagac, "A Survey on Function and System Call Hooking Approaches," Journal of Hardware and Systems Security, Vol. 1, No. 2, pp. 114-136, Jun. 2017. DOI: 10.1007/s41635-017-0013-2

[7] Y. Ye, D. Wang, T. Li, and D. Ye, "IMDS: Intelligent Malware Detection System," Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, San Jose, CA, USA, pp. 1043-1047, Aug. 2007. DOI: 10.1145/1281192.1281308

[8] K. Iwamoto and K. Wasaki, "Malware Classification Based on Extracted API Sequences Using Static Analysis," Proceedings of the 8th Asian Internet Engineering Conference, Bangkok, Thailand, pp. 31-38, Nov. 2012. DOI: 10.1145/2402599.2402604

[9] N. K. Naik, P. Jenkins, N. Savage, L. Yang, T. Boongoen, and N. Iam-On, "Fuzzy-Import Hashing: A Malware Analysis Approach," 2020 IEEE International Conference on Fuzzy Systems (FUZZ-IEEE), Glasgow, United Kingdom, pp. 1-8, Jul. 2020. DOI: 10.1109/FUZZ48607.2020.9177636

[10] G. Hunt and D. Brubacher, "Detours: Binary Interception of Win32 Functions," Proceedings of the 3rd USENIX Windows NT Symposium, Seattle, WA, USA, pp. 135-143, Jul. 1999.

[11] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, Chicago, IL, USA, pp. 190-200, Jun. 2005. DOI: 10.1145/1065010.1065034

[12] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent Dynamic Instrumentation," Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments, London, England, UK, pp. 133-144, Mar. 2012. DOI: 10.1145/2151024.2151043

[13] Intel Corporation, "Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B, 3C, & 3D): System Programming Guide," Intel, *https://cdrdv2-public.intel.com/843836/325384-sdm-vol-3abcd-dec-24.pdf*, Accessed: May 5, 2026.

[14] Intel Corporation, "Intel(R) 64 and IA-32 Architectures Software Developer's Manual, Volume 4: Model-Specific Registers," Intel, *https://cdrdv2-public.intel.com/774500/335592-sdm-vol-4.pdf*, Accessed: May 5, 2026.

[15] A. Case, M. M. Jalalzai, M. Firoz-Ul-Amin, R. D. Maggio, A. Ali-Gombe, M. Sun, and G. G. Richard III, "HookTracer: A System for Automated and Accessible API Hooks Analysis," Digital Investigation, Vol. 29, Supplement, pp. S104-S112, Jul. 2019. DOI: 10.1016/j.diin.2019.04.011

[16] A. Dinaburg, P. Royal, M. Sharif, and W. Lee, "Ether: Malware Analysis via Hardware Virtualization Extensions," Proceedings of the 15th ACM Conference on Computer and Communications Security, Alexandria, VA, USA, pp. 51-62, Oct. 2008. DOI: 10.1145/1455770.1455779

[17] Z. Deng, D. Xu, X. Zhang, and X. Jiang, "IntroLib: Efficient and Transparent Library Call Introspection for Malware Forensics," Digital Investigation, Vol. 9, Supplement, pp. S13-S23, Aug. 2012. DOI: 10.1016/j.diin.2012.05.013

[18] S.-W. Hsiao, Y.-S. Sun, and M. C. Chen, "Virtual Machine Introspection Based Malware Behavior Profiling and Family Grouping," arXiv preprint arXiv:1705.01697, May 2017.

[19] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," Proceedings of the 10th Network and Distributed System Security Symposium, San Diego, CA, USA, pp. 191-206, Feb. 2003.

[20] Z. Ding, Y. Guo, H. Xu, L. Yan, L. Cui, Y. Peng, F. Cheng, and Z. Hao, "SeqTrace: API Call Tracing Based on Intel PT and VMI for Malware Detection," Proceedings of the 22nd International Conference on Algorithms and Architectures for Parallel Processing, Copenhagen, Denmark, pp. 98-116, Oct. 2022. DOI: 10.1007/978-3-031-22677-9_6

[21] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Transparent ROP Exploit Mitigation using Indirect Branch Tracing," Proceedings of the 22nd USENIX Security Symposium, Washington, DC, USA, pp. 447-462, Aug. 2013.

[22] C. Zhao, Y. Beugin, J.-C. Noirot Ferrand, Q. Burke, G. Li, and P. McDaniel, "LibIHT: A Hardware-Based Approach to Efficient and Evasion-Resistant Dynamic Binary Analysis," Proceedings of the 1st Workshop on Software Understanding and Reverse Engineering, Taipei, Taiwan, pp. 89-101, Oct. 2025. DOI: 10.1145/3733822.3764670

[23] Microsoft, "Running 32-bit Applications," Microsoft Learn, *https://learn.microsoft.com/en-us/windows/win32/winprog64/running-32-bit-applications*, Accessed: May 5, 2026.

[24] Microsoft, "Windows Driver Kit (WDK)," Microsoft Learn, *https://learn.microsoft.com/en-us/windows-hardware/drivers/download-the-wdk*, Accessed: May 5, 2026.

[25] Microsoft, "Debug Interface Access SDK," Microsoft Learn, *https://learn.microsoft.com/en-us/visualstudio/debugger/debug-interface-access/debug-interface-access-sdk*, Accessed: May 5, 2026.

[26] Microsoft, "SymFromAddr function," Microsoft Learn, *https://learn.microsoft.com/en-us/windows/win32/api/dbghelp/nf-dbghelp-symfromaddr*, Accessed: May 5, 2026.
[27] Microsoft, "DUMPBIN Reference," Microsoft Learn, *https://learn.microsoft.com/en-us/cpp/build/reference/dumpbin-reference*, Accessed: May 5, 2026.
[28] "WinAPIOverride," *http://jacquelin.potier.free.fr/winapioverride32*, Accessed: May 5, 2026.
[29] National Security Agency, "Ghidra Software Reverse Engineering Framework," *https://ghidra-sre.org*, Accessed: May 5, 2026.

Authors

Jun-Seob Kim received the B.S. degree in Computer Engineering from Chungbuk National University in 1997, and the M.S. and Ph.D. degrees in Information Security from Sejong University in 2022 and 2026, respectively.

He joined the faculty of the Department of AI Cyber Security at Korea University, Sejong, Korea, in 2022. He is currently a Professor in the Department of AI Cyber Security at Korea University. He is interested in malware analysis and endpoint security.