오픈소스 바이너리 코드 분석 기술 동향

Trends in Open Source Binary Code Analysis Technologies

조영후 (Y.H. Jo, yallk@etri.re.kr) 차세대시스템보안연구실 선임연구원
강동호 (D.H. Kang, dhkang@etri.re.kr) 차세대시스템보안연구실 책임연구원/실장

ABSTRACT

The increasing incidence of software supply chain attacks has significantly increased the demand for technologies capable of verifying the provenance and security of software components. Although significant research has been conducted on source-code-based analysis utilizing software bills of materials, such approaches are not always feasible in industrial environments where source code is unavailable. In these environments, open source detection techniques applied to compiled binary code have emerged as essential methods for determining the reuse of open source software, identifying security vulnerabilities, and ensuring license compliance. This paper reviews open source security threats and the most recent advancements in binary code analysis techniques for the detection of open source materials.

KEYWORDS SBOM, SW 공급망 보안, 바이너리 분석, 바이너리 취약점 분석, 오픈소스 취약점

I. 서론

데브옵스(DevOps) 기반 소프트웨어(SW) 생태계는 오픈소스 소프트웨어(OSS)의 광범위한 활용을 통해 지속적이고, 높은 생산성을 제공하고 있다. 개발자들이 복잡한 기능을 처음부터 구현하는 대신 검증된 오픈소스 라이브러리를 활용함으로써 개발 속도를 가속화하고 있지만, 동시에 새로운 형태의 보안 취약점을 야기하고 있다. SW에 어떤 구성 요소가 포함되어 있는지 정확히 파악하고, 그 구성 요소에 내재된 보안 취약점을 사전에 관리하는 것이 SW 공급망 보안의 핵심 과제로 부상하였다.

2020년 SolarWinds 공격과 같이 신뢰받는 SW 공급자의 업데이트 과정에 악성코드를 심어 유통시키는 방식의 대규모 공급망 공격 사례가 발생하면서 이러한 위협의 심각성은 더 주목받았다. 이 공격으로 인해 1만 8천 개 이상의 기업과 정부 기관이 피해를 입었으며, 피해 사실이 알려지기까지 무려 10

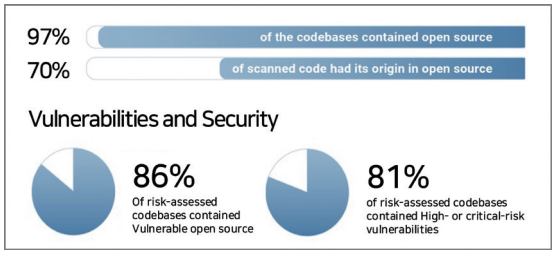
* DOI: https://doi.org/10.22648/ETRI.2025.J.400504
* 본 논문은 2024년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구 결과임[No. RS-2024-00437337, 프라이버시 보존형 내부 보안정보의 보안관제 연계·활용 기술 개발].

개월이 소요될 정도로 기존의 시그니처 기반 보안 솔루션으로는 탐지가 어려웠다. 이후 3CX, Okta 등 유사한 공급망 공격이 지속적으로 발생하며, 하나의 공격이 수많은 기업과 개인에게 광범위하고 연쇄적인 피해를 줄 수 있다는 점이 명확해졌다.

이러한 배경에서 SW 개발 수명주기 전반에 걸쳐 보안을 내재화하는 안전한 SW 개발체계(SSDF) 구축과 더불어 SW 구성 요소의 투명성을 확보하는 SW 구성 요소 목록(SBOM) 기술이 필수적인 요소로 대두되었다. 특히, 소스코드가 제공되지 않는 상용 소프트웨어나 임베디드 펌웨어의 경우, 소스코드 기반의 분석이 불가능하여 SW 구성 요소의 출처를 파악하고 보안 취약점을 탐지하는 데 한계가 있었다. 바이너리 구성 요소 분석은 컴파일러 최적화, 난독화, 실행 환경 차이 등 여러 기술적 난제에 직면해 있으며, 이러한 난제들을 극복하기 위한 연구가 꾸준히 진행되고 있다. 본고에서는 오픈소스 활용에 따른 보안 위협을 살펴보고, SW 공급망 보안을 강화하기 위한 오픈소스 바이너리 코드 분석의 기술 동향을 살펴보고자 한다.

II. 오픈소스 소프트웨어 보안 위협

오픈소스 소프트웨어를 활용하는 것은 비용 절감, 개발 속도 향상, 혁신 촉진 등의 장점이 있지만, SW 생태계에서 오픈소스 활용 비중이 높아지면서 오픈소스로 인한 보안 위험을 공격 벡터로 악용되는 사례가 증가하고 있다. 기업이 사용하는 SW 중 약 97%가 오픈소스를 포함하고 있는 것으로 보고되고 있다. 그중 코드 베이스의 약 86%는 한 개 이상의 취약점을 포함하고 있어 개발 과정에서 오픈소스의 활용 비중이 높으나 보안관리는 미흡하다는 것을 알 수 있다. 이 중에서 고위험군의 취약점은 약 81%로 오픈소스 재사용으로 인한 사이버 위협 심각성이 대두되고 있다(그림 1).



출처 Reproduced with permission from Black Duck, "2025 Open Source Security and Risk Analysis Report," Apr. 2025.

그림 1 2024년 오픈소스 활용 및 취약점 현황 분석

가트너는 2025년에 전 세계 조직의 45%가 SW 공급망에 대한 공격을 경험할 것이라고 경고하고 있으며, 오픈소스 SW 사용 증가에 따라 소프트웨어 공급망 공격으로 인한 피해액이 2023년 460억 달러에서 2031년 1,380억 달러로 증가할 것으로 추정한다[1].

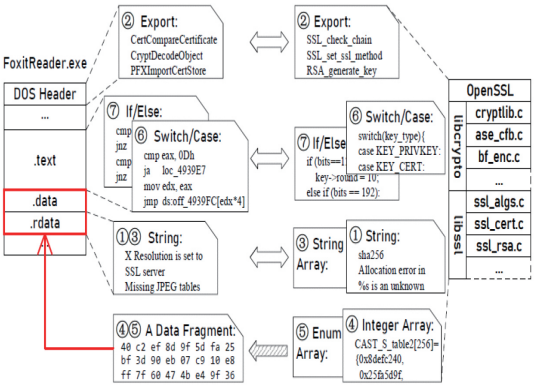
오픈소스는 라이브러리나 패키지로 구성되어 다단계로 의존성을 가지고 있어서, 이를 포함하는 SW에 대한 취약점 탐지는 기술적으로 매우 어렵다. 따라서, 조직은 오픈소스 사용 현황을 완전히 파악하

표 1 오픈소스 활용에 따른 보안 위협

보안 위협	세부내용
취약점이 포함된 오픈소스 구성 요소	오픈소스 라이브러리의 보안 취약점으로 인해 이를 사용하는 시스템에 취약점 전파
악의적인 코드 삽입	공격자가 기존의 신뢰할 수 있는 오픈소스 라이브러리나 패키지에 악성코드를 포함시키는 방식(타이포스쿼팅(Typosquatting), 의도적 취약점 삽입)
유지보수와 업데이트 부족	유지보수 및 보안 패치 미흡 등의 관리 부실로 인한 신규 취약점 대응 미흡
의존성 문제	패키지 내 의존성 있는 취약 라이브러리로 인한 취약점 전이
허위 패키지 및 악성 프로젝트	유명 오픈소스 프로젝트와 유사한 이름을 사용하거나, 신뢰할 수 있는 프로젝트로 위장하여 악성코드 배포

지 못하는 경우가 많으며, 그 결과 취약점을 즉시 수정하지 못하는 위험에 처할 수 있다. 대표적인 오픈소스 취약점을 이용한 보안 위협은 표 1과 같이 정리할 수 있다.

오픈소스 SW는 투명성과 커뮤니티의 활발한 참여 덕분에 빠른 발전을 이루고 있지만, 그만큼 보안 위협에 노출될 가능성도 크기 때문에 기업은 SW 공급망 보안 강화를 위해서는 오픈소스 라이브러리의 사용을 신중하게 관리하고, 주기적인 보안 점검 및 업데이트 방안을 제시해야 한다.



출처 Reprinted with permission from Z. Yuan et al., "B2SFinder: Detecting Open-Source Software Reuse in COTS Software," in Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng., (San Diego, CA, USA), Nov. 2019, pp. 1038-1049.

그림 2 B2SFinder의 7가지 코드 특징 추출

III. 오픈소스 바이너리 코드 분석 기술

1. 정적 바이너리 코드 분석 기술

정적분석은 바이너리 코드를 실행하지 않고 소스 코드를 이진 코드로 컴파일하는 과정에서 컴파일러가 수행하는 최적화, 구조 재구성, 중복성 제거로 인해 발생하는 정보 손실에 영향을 덜 받거나, 분석 과정에서 복원이 가능한 문자열, 그래프 기반 제어흐름, 해시 정보 등을 특징으로 추출하여 유사성을 비교하는 기법이다[2].

1.1 문자열 매칭 기법

바이너리 코드 내에 컴파일러 최적화나 난독화에도 잘 변하지 않는 문자열(파일 경로, 패키지 이름, 버전 정보 등)을 추출하여 바이너리 코드 유사성을 비교하는 기술이다. 분석 대상 바이너리 파일에서 추출한 문자열과 오픈소스에서 추출한 문자열 데이터베이스를 조회하여 일치하는 패키지를 비교 검색하고, 여러 패키지에 공통으로 존재하는 짧은 문자열에는 낮은 가중치를 부여하고, 고유한 문자열에는 높은 가중치를 부여하여 정확도를 높이는 방식을 채택하고 있다. 대표적으로 B2SFinder[3]는 소스 코드를 이진 코드로 컴파일하는 과정에서 비교적 잘 보존되는 7가지 추적 가능 특징(문자열, 문자열 배열, Export 함수, 정수 배열, 열거형 배열, If/Else 구문, Switch/Case 구문)을 기반으로 대상 바이너리에서 추출하여 이미 구축된 오픈소스 데이터베이스의 정보와 매칭하는 기술을 적용하고 있다. B2SFinder의 7가지 코드 특징 추출하는 과정은 그림 2와 같다.

1.2 그래프 매칭 기법

그래프 매칭 기반 기법은 바이너리 코드를 프로그램의 구조적 및 논리적 흐름을 반영하는 그래프로 변환한 후, 두 그래프 간의 유사도를 측정하여 바이너리의 유사성을 판단한다. 대표적인 그래프 형태로는 명령어 실행 경로를 나타내는 제어 흐름 그래프(CFG: Control Flow Graph), 데이터의 정의와 사용 관계를 표현하는 데이터 흐름 그래프(DFG: Data Flow Graph), 그리고 함수 호출 관계를 나타내는 호출 그래프(Call Graph)가 있다. 대표적인 호출 그래프 매칭 기반 도구인 BinDiff[4]가 있다. 그래프 매칭 기법은 코드의 구조적, 논리적 흐름을 반영하여 단순 문자열 매칭보다 정확도가 높고, 컴파일러 최적

화로 인한 구문적 변화에 어느 정도 강건한 장점이 있다. 다만, 계산 복잡도가 매우 높아 대규모 바이너리 분석에 적용하기 어렵고, 컴파일러 최적화나 난독화가 그래프의 토폴로지를 변경할 경우, 코드 식별에 많은 오탐이 발생할 수 있다.

1.3 Hashing 매칭 기법

Hashing 기반 기법은 바이너리를 실행하지 않고 Hashing 매칭을 통해 유사성을 분석하는 정적 분석 방법으로, 빠른 분석이 가능하다는 장점이 있다. 기존의 암호화 해시(MD5, SHA 등)는 파일 내용이 단 1비트만 달라져도 완전히 다른 해시값을 생성하여 파일의 동일성을 검증하는 데 주로 사용되었다. 그러나 바이너리 유사성을 파악하는 데는 한계가 있다. 이러한 한계를 극복하기 위해 퍼지 해싱(Fuzzy Hashing) 기술이 등장했다. 퍼지 해싱은 파일 전체가 아닌 일정 크기 단위로 해시를 생성하여, 샘플 간의 유사성 비율을 파악하는 데 도움을 준다. 대표적으로 지역성 의존 해싱(LSH: Locality-Sensitive Hashing) 기법이 있다. LSH는 유사한 입력 항목을 높은 확률로 동일한 '버킷'에 할당하는 기술로, 기존 해싱과 달리 해시 충돌을 최소화하는 대신 유사한 항목 간의 충돌을 극대화한다. 이를 통해 대규모 데이터셋에서 효율적인 유사성 검색과 데이터 클러스터링을 수행할 수 있다. Ghidra[5]의 BSim 플러그인은 이 LSH 기술을 활용해 함수별 특징 벡터를 생성하고 데이터베이스에 인덱싱함으로써 구조적으로 유사한 함수를 빠르게 찾아내고 있다.

2. AI 기반 바이너리 코드 분석 기술

인공지능(AI) 기술의 발전에 따라, 바이너리 코드를 분석하는 연구 분야에서도 AI를 접목하려는 시도가 꾸준히 증가하고 있다. 바이너리 코드 분석을 위해서는 바이너리 내 함수들의 고유한 특징을 정의하는 과정이 선행되어야 한다. 이러한 특징으로는 함수의 실행 흐름을 나타내는 제어 흐름 그래프(CFG)의 구조적 형태와 함수의 어셈블리어 코드 자체가 있다. AI 기반 바이너리 코드 분석에서는 CFG의 구조적 특징을 그래프 임베딩(Graph Embedding) 기법으로, 어셈블리 코드는 자연어 처리 기반의 코드 임베딩(Code Embedding) 기법으로 각각 벡터화할 수 있다. 이렇게 얻은 벡터 표현은 함수 간 의미적 유사도를 비교하는 데 활용된다.

2.1 그래프 임베딩

바이너리 코드 분석 분야에서 그래프 임베딩은 연속된 명령어 집합인 기본 블록(Basic Block)과 조건문 및 점프를 통한 분기(Edge)로 구성된 CFG를 함수 간 비교를 위해 벡터화하는 데 활용된다. 그러나 그래프 구조만을 함수의 특징으로 사용할 경우, 실행 흐름 구조는 유사하지만 내부적으로 서로 다른 의미를 가지는 함수를 구분하기 어렵다. 이러한 한계를 극복하기 위해 CFG의 기본 블록에 해당 블록의 고유한 특성을 반영할 수 있는 추가적인 특징 정의가 필요하게 되었으며, 이를 처음 제안한 것이

표 2 Genius에 사용된 기본 블록의 특징

유형	특징명
통계적 특징	• 문자열 • 숫자 상수들 • 분기문 관련 개수 • 함수 호출 개수 • 명령어들의 개수 • 산술 연산 명령어 개수
구조적 특징	• 연결된 자식 노드들의 수 • 해당 기본 블록이 전체 함수에서 차지하는 중심성(Betweenness)

출처 Reproduced with permission from Q. Feng et al., "Scalable Graph-based Bug Search for Firmware Images," in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., (Vienna, Austria), Oct. 2016, pp. 480-491.

Genius[6] 논문에서의 속성 기반 제어 흐름 그래프(ACFG: Attributed Control Flow Graph)이다.

ACFG는 기본 블록에 통계적(Statistical) 및 구조적(Structural) 특징을 반영하여 표현력을 향상시킨다. 통계적 특징에는 문자열 및 상수, 분기문 관련 명령어 개수, 함수 호출 명령어 개수, 해당 블록의 전체 명령어 수, 그리고 산술 연산 명령어 개수가 포함된다. 구조적 특징에는 해당 블록과 직접 연결된 자식 노드(Child Node)의 수와 그래프 내에서 해당 블록이 차지하는 중심성(Betweenness)이 포함된다. 이러한 특징들은 표 2에 요약되어 있다.

Genius는 함수의 CFG를 ACFG로 변환한 후, 코드북(Codebook) 기반 그래프 임베딩을 통해 각 함수를 벡터화하고, 이를 이용하여 함수 간 유사도를 비교하였다. 그러나 이 방법은 학습 기반 임베딩이 아닌 클러스터링 기반 표현이므로, 코드북 내 대표 ACFG와의 유사도 벡터를 생성할 뿐 미묘한 문맥 차이를 포착하기 어렵고, 임베딩 계산 과정에서 상당한 시간이 소요된다.

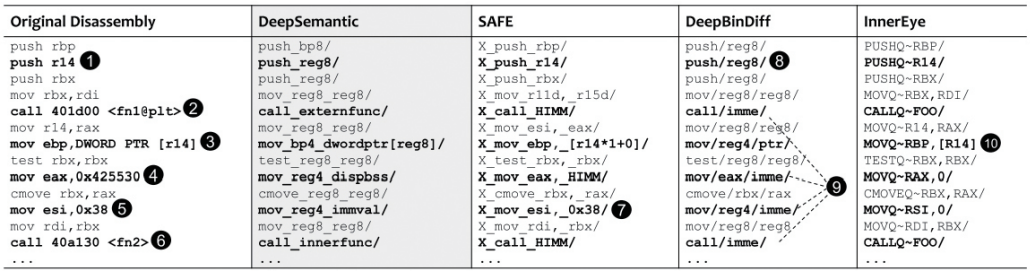
이후 등장한 Gemini[7]는 Genius에서 정의한 ACFG를 그대로 사용하되, 학습 가능한 Structure2vec[8] 모델을 이용해 그래프 임베딩을 생성하고, Siamese Network를 통해 함수 유사도를 비교한다. 또 다른 후속 연구인 BugGraph[9]는 기존 ACFG를 유지하면서도, 학습 가능한 그래프 신경망(GNN: Graph Neural Network) 구조로 그래프 어텐션 네트워크(GAT: Graph Attention Network)를 도입하였다. 또한, 전통적인 Siamese Network 대신 Triplet Loss 기반의 랭킹 학습을 적용하여, 임베딩 공간에서 Type-1 > Type-2 > Type-3 > Different 순서로 거리가 유지되도록 학습한다. 여기서 Type-1은 동일한 소스 코드를 다른 컴파일 설정(컴파일러, 최적화 옵션 등)으로 빌드한 경우(Literally Same), Type-2는 의미는 동일하지만 변수명, 주석, 약간의 문법 차이 등만 있는 경우(Syntactically Equivalent), Type-3는 코드의 구조나 제어 흐름이 부분적으로 변경되었지만 여전히 유사성이 있는 경우(Syntactically Similar)를 의미한다. 이러한 유형 분류와 랭킹 학습 결합을 통해 BugGraph는 다양한 코드 변형 환경에서의 바이너리 코드의 특징을 잘 표현할 수 있도록 하였고, 바이너리 코드들끼리의 유사도 비교 성능을 향상시켰다.

2.2 코드 임베딩

코드 임베딩은 소스 코드나 바이너리 코드를 자연어처럼 인식 · 처리하여 각 함수를 벡터로 표현하는 기법이다. 이러한 접근은 자연어 처리(NLP: Natural Language Processing) 기술의 발전과 함께 진화해 왔다.

가장 초기의 연구로는 asm2vec[10]이 있다. asm2vec은 NLP 분야의 word2vec을 확장한 PV-DM (Paragraph Vector-Distributed Memory) 모델을 변형하여 어셈블리 코드를 해석하고, 이를 기반으로 코드 유사도를 탐지한다. 여기서 함수는 문서(Paragraph)로, 명령어는 단어(Word)로 간주되며, 특정 명령어를 예측하기 위해 그 이전 · 이후 명령어를 함께 학습한다. 또한, 함수의 CFG를 반영하기 위해 CFG 기반 랜덤 워크를 통해 명령어 시퀀스를 생성하고 이를 PV-DM 모델의 입력으로 사용한다.

다음으로 제안된 SAFE[11]는 asm2vec과 달리 CFG를 사용하지 않는 순수 명령어 시퀀스 기반 접근을 채택한다. SAFE는 어셈블리 명령어에 word2vec(skip-gram)을 적용한 instruction2vec(i2v)로 토큰을 벡터화한 뒤, 양방향 순환신경망(Bi-RNN)과 셀프 어텐션(Self-Attention)으로 함수 임베딩을 생성한다. 이후 Siamese Network 구조를 통해 Positive/Negative 함수 쌍을 학습하며, 손실 함수는 유사도 예측 오차와 어텐션 정규화 항으로 구성된다. SAFE는 x86 아키텍처에만 적용된 asm2vec과 달리 ARM



출처 Reprinted from H.J. Koo et al., "Binary Code Representation with Well-Balanced Instruction Normalization," IEEE Access, vol. 11, 2023, pp. 29183-29198.

그림 3 바이너리 코드와 정규화를 적용한 모델들의 코드 비교

및 AMD64 아키텍처까지 지원하며, CFG를 처리하지 않기 때문에 속도 측면에서도 유리하다. 그러나 asm2vec과 SAFE 모두 Skip-gram/CBOW 기반 word2vec 모델을 사용하므로, 학습 이후 임베딩이 고정되며, 어휘집(Vocabulary)에 없는 단어(OOV: Out-Of-Vocabulary)는 처리할 수 없고, 멀리 떨어진 명령어 간 관계를 충분히 반영하기 어렵다는 한계가 있다.

이러한 한계를 극복하기 위해 NLP 분야에서 제안된 BERT 개념이 코드 임베딩에 도입되었으며, 이를 어셈블리 코드 분석에 맞게 적용한 연구가 jTrans[12]이다. jTrans는 BERT 구조에 어셈블리 코드의 흐름과 구조를 반영할 수 있도록 점프 인지(Jump-Aware) Position Embedding을 도입하였다. 구체적으로, jTrans는 어셈블리 코드에서 분기가 일어나는 명령어와 그 분기가 도착하는 목적지 명령어를 한 쌍으로 묶어 처리한다. 이때 두 명령어 쌍은 동일한 좌표계에서 표현되도록 임베딩과 위치(Position) 정보의 학습 파라미터를 공유한다. 이렇게 하면, 모델이 분기 명령어에 높은 주목도(Attention)를 부여할 때, 해당 분기 목적지에도 자연스럽게 비슷한 수준의 주목도가 전달된다. 쉽게 말해, 점프 소스와 목적지를 '같은 표식이 있는 두 지점'으로 만들어, 한쪽을 주목하면 자동으로 다른 쪽도 함께 주목하도록 설계하였다. 또한, 사전학습 단계에서 Masked Language Model(MLM)과 Jump Target Prediction(JTP)을 결합하고, 미세조정(Fine-Tuning) 단계에서는 대조학습(Contrastive Learning)을 통해 유사한 함수가 임베딩 공간에서 가깝게 위치하도록 학습한다.

한편, 어셈블리 코드를 그대로 학습할 경우 의미와 무관한 노이즈나 OOV 문제가 발생할 수 있다. 이를 해결하기 위해 각 연구에서는 어셈블리 코드에 대한 고유한 정규화(Normalization) 방식을 정의·적용한다. 정규화 방법은 모델 성능에 큰 영향을 미치며, 더 나은 정규화 전략을 제시한 연구로 DeepSemantic[13]이 있다. 이 연구는 균형 잡힌 명령어 정규화(Well-Balanced Instruction Normalization)를 목표로 하며 BERT에 적용하여, 코드 의미 보존과 임베딩 품질을 동시에 개선하였다(그림 3).

2.3 대형 언어 모델(LLM) 기반 임베딩

대형 언어 모델(LLM: Large Language Model)은 Transformer 디코더를 기반으로 한 생성형 언어 모델로, 초기에는 주로 텍스트 생성, 요약, 번역 등 자연어 처리 분야에 활용되었으며, 바이너리 코드 분석 분야에서는 상대적으로 관심이 적었다. 그러나 모델 규모와 학습 데이터양이 급격히 증가하고, 고

성능 하드웨어의 보급이 확산되면서 LLM의 표현 능력과 범용성이 크게 향상되었다. 이에 따라 LLM의 사전학습 표현력을 활용한 바이너리 코드 분석 및 유사도 비교에 적용하려는 시도가 활발히 진행되고 있다.

CLAP[14] 연구는 소스 코드를 컴파일하여 얻은 어셈블리 코드와 해당 소스 코드를 입력으로, ChatGPT를 통해 관련 코드 설명을 생성하고 이를 쌍(Pair)으로 저장하는 방식으로 시작된다. 이후 LLaMA 및 Vicuna 모델을 이용하여 추가적인 설명을 확장하고, 이렇게 수집된 데이터셋을 기반으로 RoBERTa 아키텍처를 변형한 CLAP-ASM 모델을 학습하여 어셈블리 코드를 벡터로 변환한다. 동시에, 자연어 코드 설명은 MPNet-base-v2를 기반으로 하는 CLAP-TEXT 인코더를 통해 벡터로 변환된다. 두 인코더는 대조학습(Contrastive Learning)을 통해 동일 함수의 어셈블리 벡터와 설명 벡터가 임베딩 공간에서 근접하도록 학습된다. 이 방식은 어셈블리 코드 간 유사도 비교뿐만 아니라 입력된 어셈블리 코드가 정렬, 시간, 오디오 처리 등 어떤 기능 범주에 속하는지를 분류하는 데도 활용된다.

BinaryAI[15] 연구는 LLM을 단순히 텍스트 생성에 국한하지 않고, 인코더 기반 벡터화 모델로 재구성하여 바이너리 함수 매칭을 한층 더 발전시켰다. 이 방법은 바이너리 내 어셈블리 함수를 Ghidra를 통해 C 형태의 의사 코드(Pseudocode)로 변환하고, 소스 코드와 함께 학습하여 서드파티 라이브러리(TPL: Third-Party Library) 탐지 및 유사 함수 매칭을 수행한다. 임베딩 생성을 위해 사용된 LLM은 pythia-410m이며, 대조학습을 기반으로 학습된다. 이때 손실 함수로는 이미지-텍스트 쌍 학습에서 널리 쓰이는 CLIP(Contrastive Language-Image Pre-training) 방식이 적용되었다.

BinaryAI는 단순 임베딩 기반 비교의 한계를 극복하기 위해 Locality-Driven Matching 기법을 제안하였다. 이는 TPL 내부에 유사한 함수가 다수 존재하는 특성상, 단일 임베딩만으로 정확한 매칭이 어려운 문제를 해결하기 위함이다. 해당 기법은 링크(Link) 과정에서 동일한 .o 파일 단위로 함수들이 바이너리에 포함되는 경향을 활용하여, 후보 함수 매칭 정확도를 높인다. BinaryAI는 논문뿐만 아니라 현재 서비스 형태[16]로도 제공되고 있다.

IV. 결론

본고에서는 오픈소스 소프트웨어의 보안 위협과 오픈소스 바이너리 분석 기술을 살펴보았다. 오픈소스 바이너리 구성 요소 탐지를 위한 바이너리 분석 기술은 컴파일러 최적화, 난독화 등으로 변형된 바이너리에 대하여 소스코드가 가지는 구조적 · 의미적 특징을 동일하게 추출하여 분석하는데 여전히 한계를 보이고 있다.

문자열 매칭 기법은 구현이 간단하고 속도가 빠르지만, 난독화나 컴파일러 최적화에 취약하여 정밀한 분석에는 한계가 있고, 그래프 매칭 기법은 코드의 구조적 특징을 반영하여 비교적 높은 정확도를 제공하지만, 계산 복잡도가 높고 그래프 구조가 변형될 경우 유사성을 놓치기 쉽다. 해싱 매칭 기법은 대규모 데이터셋에 대한 빠른 검색과 분류에 최적화된 기술로, 저장 공간 효율성이 높지만 정밀한 유사성 분석에 적합하지 않다.

최근에는 이러한 한계를 극복하기 위해 임베딩 기반 기법이 활발히 연구되고 있다. 그래프 임베딩 기반 접근은 제어 흐름 그래프(CFG)를 속성 기반 제어 흐름 그래프(ACFG)로 확장하고, 학습 가능한 그래프 신경망(GNN)을 활용해 구조적 · 통계적 특징을 벡터화함으로써 코드 구조 변형에도 견고한 유사도 비교를 가능하게 한다. 코드 임베딩 기반 접근

은 어셈블리 명령어 시퀀스를 자연어 처리 기법으로 학습하여 의미적 관계를 포착하며, 자연어 처리의 word2vec 계열 모델에서 BERT와 같은 사전학습 언어 모델로 발전하면서, OOV 문제와 장거리 의존성 반영 한계를 개선하였다. 더 나아가 대형 언어 모델(LLM) 기반 기법은 사전 학습된 모델의 풍부한 표현력을 활용하여, 어셈블리, 의사코드, 자연어 설명 간 의미 공간을 통합하고, 대규모 대조학습을 통해 코드 유사도 비교뿐만 아니라 기능 분류, 서드파티 라이브러리 탐지 및 취약점이 있는 코드 탐지 등 다양한 과제를 동시에 수행할 수 있도록 확장되고 있다.

용어해설

SSDF(Secure Software Development Framework) 소프트웨어의 계획, 설계, 구현, 테스트, 배포, 유지보수 전 과정에서 보안을 체계적으로 내재화하기 위한 개발 프레임워크

SBOM(Software Bill of Materials) 소프트웨어를 구성하는 모든 오픈소스 및 상용 구성 요소, 라이브러리, 모듈, 버전, 공급자 정보 등을 목록화한 문서 또는 데이터 형식

Embedding 단어, 문장, 코드, 그래프와 같은 이산적인 데이터를 고차원 벡터 공간의 연속적인 벡터로 변환하는 표현 방식

codebook 데이터 집합을 대표하는 기준 벡터(또는 그래프)들의 모음으로, 새로운 입력 데이터를 벡터로 표현할 때 기준들과의 유사도를 계산하여 embedding을 생성하는 데 사용

Skip-gram 단어 임베딩 학습 기법 중 하나로, 중심 단어를 입력으로 받아 해당 단어의 주변 단어를 예측하는 모델 구조

CBOW(Continuous Bag of Words) 단어 임베딩 학습 기법으로, 주변 단어들을 입력으로 받아 중심 단어를 예측하는 모델 구조

참고문헌

[1] Gartner, “Leader’s Guide to Software Supply Chain Security,” 2024. 6.

[2] 강동호, 조영후, “바이너리 코드 유사성 분석 기술 동향,” 정보보호학회지, 제35권 제1호, 2025, pp. 17-24.

[3] Z. Yuan et al., “B2SFinder: Detecting Open-Source Software Reuse in COTS Software,” in Proc. IEEE/ACM Int. Conf. Autom. Softw. Eng., (San Diego, CA, USA), Nov. 2019, pp. 1038-1049.

[4] BinDiff. https://github.com/google/bindiff

[5] Ghidra. https://github.com/NationalSecurityAgency/ghidra

[6] Q. Feng et al., “Scalable Graph-based Bug Search for Firmware Images,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., (Vienna, Austria), Oct. 2016, pp. 480-491.

[7] X. Xu et al., “Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection,” in Proc. ACM SIGSAC Conf. Comput. Commun. Secur., (Dallas, TX, USA), Oct. 2017, pp. 363-376.

[8] H. Dai et al., “Discriminative Embeddings of Latent Variable Models for Structured Data,” in Proc. Int. Conf. Mach. Learn., (New York, NY, USA), Jun. 2016, pp. 2702-2711.

[9] Y. Ji et al., “BugGraph: Differentiating Source-Binary Code Similarity with Graph Triplet-Loss Network,” in Proc. ACM Asia Conf. Comput. Commun. Secur., Jun. 2021, pp. 702-715.

[10] S.H.H. Ding et al., “Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search Against Code Obfuscation and Compiler Optimization,” in Proc. IEEE Symp. Secur. Privacy, (San Francisco, CA, USA), Sep. 2019, pp. 472-489.

[11] L. Massarelli et al., “SAFE: Self-Attentive Function Embeddings for Binary Similarity,” in Proc. Int. Conf. Detect. Intrusions Malware Vulnerability Assess., (Gothenburg, Sweden), Jun. 2019, pp. 309-329.

[12] H. Wang et al., “JTrans: Jump-Aware Transformer for Binary Code Similarity Detection,” in Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal., Jul. 2022, pp. 1-13.

[13] H.J. Koo et al., “Binary Code Representation with Well-Balanced Instruction Normalization,” IEEE Access, vol. 11, 2023, pp. 29183-29198.

[14] H. Wang et al., “CLAP: Learning Transferable Binary Code Representations with Natural Language Supervision,” in Proc. ACM SIGSOFT Int. Symp. Softw. Test. Anal., (Vienna, Austria), Sep. 2024, pp. 503-515.

[15] L. Jiang et al., “BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching,” in Proc. IEEE/ACM Int. Conf. Softw. Eng., (Lisbon Portugal), Apr. 2024, pp. 1-13.

[16] BinaryAI Website. https://www.binaryai.cn/