오픈소스 SW 취약점 분석 및 탐지기술 동향

우승훈, 홍현지, 이희조
Korea University

Abstract

혁신적인 소프트웨어 개발을 위해 개발 과정에서 오픈소스 소프트웨어를 활용하는 것은 하나의 트렌드로 자리잡았다. 비록 오픈소스 소프트웨어의 활용은 개발과정에서 많은 효율성을 제공하지만, 관리되지 않은 오픈소스 소프트웨어의 재사용은 보안 취약점의 전파와 같은 위협으로 이어질 수 있다. 본 고에서는, 이러한 오픈소스 소프트웨어 재사용으로 인해 발생할 수 있는 보안 위협들을 명시하고, 이에 대응하기 위한 다양한 오픈소스 보안 기술들을 소개한다.

Ⅰ. 서론 (Introduction)

오픈소스 소프트웨어(OSS)는 소스코드가 공개되어 있어, 라이선스를 준수하는 범위에서 누구나 특별한 제한 없이 재사용, 수정 및 배포가 가능한 소프트웨어를 말한다. 개발자들은 소프트웨어 개발 시 필요한 세부 기능들을 일일이 구현하지 않고, 이미 구현되어 있는 OSS 코드의 재사용을 통해 소프트웨어 개발시간 및 비용을 줄일 수 있다. 이러한 이점으로 OSS 재사용 수가 꾸준히 증가하고 있으며, 실제로 시놉시스 '오픈소스 보안 및 위험 분석 2022' 보고서는 소프트웨어 97%가 하나 이상의 오픈소스를 재사용 중임을 밝혔다 [1].

하지만, 재사용 중인 OSS를 적절히 관리하지 않으면 다양한 보안 문제를 일으킬 수 있다. 대표적으로 (1) 소프트웨어 취약점 전파, (2) 라이선스 위반 문제, (3) 공급망 공격과 같은 문제를 일으킨다. 실제로, 시놉시스 2022년도 보고서에서, OSS를 재사용 중인 소프트웨어 중 81%는 하나 이상의 취약점을 갖고 있는 것으로 밝혀졌다 [1]. 또한, 소나타입에서 발간한 2021년 '소프트웨어 공급망 현황' 보고서에 따르면, OSS의 관리 부족으로 이어진 공급망 공격이 2021년도에 650% 증가한 것을 나타냈다 [2].

이러한 위협을 해결하는 방법은 크게 두 가지로 구분할 수 있다: (1) OSS 구성요소 관리(OSS component management), 그리고 (2) 코드 감사(code audit)이다. OSS 구성요소 관리의 경우 검사 소프트웨어가 어떠한 OSS를 재사용 중인지 지속적으로 관리하여 보안 위협을 예방할 수 있다. 구체적으로, 공개 취약점 데이터베이스(예, NVD[1], CVE MITRE[2])에서 제공하는 취약한 소프트웨어 및 패키지 버전정보(CPE)를 활용하여, 사용중인 구성요소를 안전한 버전으로 업데이트 함으로써 발생 가능한 보안 위협에 대응할 수 있다. 코드 감사의 경우 주기적으로 정적, 동적 분석 도구를 이용하여 취약한 코드를 탐지 및 패치 함으로써 보안 위협을 예방하는데 활용될 수 있다.

본 고에서는 OSS 재사용으로 인해 발생하는 보안 위협을 해결할 수 있는 다양한 연구들을 소개한다. 구체적으로, 취약 코드를 탐지하기 위한 연구(부문 2.1, 부문 2.2, 부문 2.3, 부문 2.6) 및 OSS 구성요소를 정확히 탐지하기 위한 연구(부문 2.4, 부문 2.5)들을 소개한다. 특히, 본 고에서 제안하는 연구들은 소프트웨어 제품의 구성요소를 투명하게 관리할 것을 규제화해 나가고 있는 현 시점에서 보안 뿐만 아니라 소프트웨어 제품 관리 목적으로 범용성 있게 활용될 수 있다 [3].

Ⅱ. 오픈소스 취약점 탐지 연구 동향

본 고에서는 오픈소스 취약점 탐지 기술에 대해 [표1]과 같이 세분화하여, 각 구분에 대해 현황 및 문제점을 제시하고, 이를 위한 연구 방향을 제안한다.

표 1. 오픈소스 취약점 탐지 연구 기술 세분화

구분	취약 데이터베이스	구성요소 분석	취약코드 탐지
연구 현황 및 문제점	공개 취약점 데이터베이스에서 바로 가져올 수 있는 패치만 적재 공개 취약점 데이터베이스가 잘못된 취약점 원점 정보를 제공함	구성요소 탐지 분석 도구(SCA)를 이용해 타겟 프로그램에서 재사용된OSS 식별 수정된 OSS 재사용의 경우에 정확한 식별이 어려움	정적, 동적 분석 도구를 이용해서 취약 코드를 탐지 기존 전파된 취약점 탐지 기술은 많은 분석시간을 요하거나, 정확도가 떨어짐
연구 방향	취약 코드 탐지에 활용되는 취약 데이터베이스를 구축(부문 2.3) 취약점 원점정보를 탐지(부문 2.5) 개발자 사이트에서 안전하지 않은 코드 탐지(부문 2.6)	수정된 OSS 구성요소를 정확하게 탐지(부문 2.4)	알려진 취약 코드를 빠르고 확장성 있게 탐지(부문 2.1) 수정된 취약 코드를 정확하게 탐지(부문 2.2)

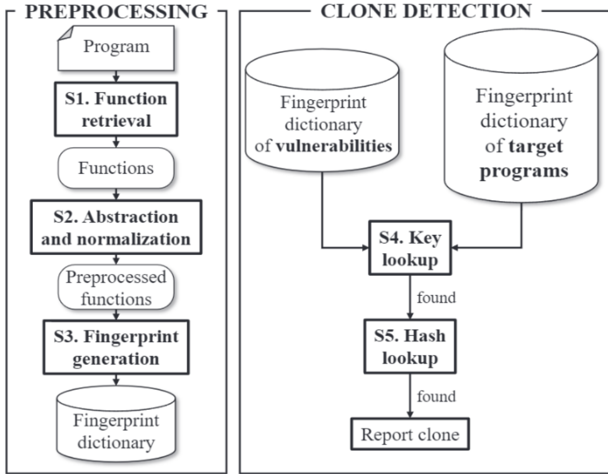
1) https://nvd.nist.gov/
2) https://cve.mitre.org/

2.1. 취약 코드클론 탐지 기술 (VUDDY)

2.1.1. 문제

먼저 취약한 소스코드 클론 탐지 기술인 VUDDY [4]에 대해 소개하고자 한다. 개발자들은 OSS 재사용 시 이미 구현되어 있는 코드의 일부 또는 전체를 복제하여 활용하는 경우가 많다. 이렇게 복제된 유사한 코드조각들을 코드 클론(code clone)이라 칭한다. 문제는, 원본 OSS의 취약한 코드를 클로닝 하는 경우, 해당 코드에 포함된 취약점 역시 전파된다는 점이다.

기존의 전파된 취약 코드 탐지 기술은 다음의 두 가지 한계점이 있었다: (1) 많은 분석 시간 소요로 인한 낮은 확장성(scalability), (2) 많은 오탐으로 인한 낮은 정확도(accuracy). 따라서, 대규모의 코드 베이스를 분석하여 정확하게 취약 코드를 탐지해 낼 수 있는 기술이 필요하게 되었다.



〈그림 1. VUDDY 전체 흐름도〉

2.1.2. 방법론

핵심 방법론
함수레벨 세분성 활용 추상화 및 정규화를 활용하여 일부 변형된 취약코드클론도 탐지 가능

이에 따라 소스코드 레벨에서, 취약한 코드 클론을 빠르고 확장성 있게 탐지할 수 있는 기술인 VUDDY를 제안한다. [그림1]과 같이 VUDDY는 총 두 단계로 구성된다: (1) 전처리(Preprocessing), 그리고 (2) 클론 탐지(Clone detection).

전처리 단계에서는 소프트웨어 프로그램에서 함수를 추출하여 각 함수의 지문(fingerprint)을 생성하는 것을 목표로 한다. VUDDY는 함수 단위를 소프트웨어 세분성으로 활용한다. 이는 함수 단위가 소스코드의 구문적(syntatic) 및 기호(symbolic) 정보를 모두 내포하기 때문에, 취약 코드 탐지 시 높은 정확도를 보장할 수 있기 때문이다. VUDDY는 프로그램의 소스코드에서 함수를 추출하고 함수의 바디(function body)를 대상으로 정규화(normalization) 및 추상화(abstraction)를 진행하여 취약코드 탐지 정확도를 높인다. 정규화 과정을 통해 화이트스페이스(whitespace) 문자 (예, 공백(space), 탭(tab) 문자 및 개행(new line) 문자) 및 주석(comment)들을 함수로부터 제거한다. 정규화 이후 추상화를 수행하는데, 이는 함수 내의 변수 이름 및 변수 자료형을 동일한 문자열로 일반화하여 코드의 작은 형상 변화들은 유연하게 커버할 수 있도록 하는 기술이다. 실제로, 소스코드를 복제 시 개발자가 변수의 이름, 자료형을 수정하는 경우가 많은데, VUDDY가 지원하는 추상화 기술로 이를 유연하게 탐지할 수 있다.

[그림2]에서 볼 수 있듯이, VUDDY는 다섯가지의 추상화 레벨을 지원한다: (1) 추상화를 적용하지 않음, (2) 함수의 파라미터를 FPARAM으로 대체, (3) 로컬 변수 이름을 LVAL로 대체, (4) 데이터 타입을 DTYPE로 대체, (5) 함수 호출을 FUNCCALL로 대체. 여기서 상위 레벨 추상화는 하위 레벨의 추상화까지 포함한다. 이후 추상화 단계를 거친 함수 바디에 대해 해시 함수(hash function)에 기반하여 해시 값을 추출한다. 이때 함수 바디의 길이를 키(key) 값으로, 해시 값(value)과 함께 딕셔너리(dictionary) 자료 구조로 저장한다.

```
    Level 0: No abstraction.
1   void avg (float arr[], int len) {
2     static float sum = 0;
3     unsigned int i;
4     for (i = 0; i < len; i++);
5       sum += arr[i];
6     printf("%f %d",sum/len,validate(sum));
7   }
    Level 1: Formal parameter abstraction.
1   void avg (float FPARAM[], int FPARAM) {
2     static float sum = 0;
3     unsigned int i;
4     for (i = 0; i < FPARAM; i++)
5       sum += FPARAM[i];
6     printf("%f %d",sum/FPARAM,validate(sum);
7   }
    Level 2: Local variable name abstraction.
1   void avg (float FPARAM[], int FPARAM) {
2     static float LVAR = 0;
3     unsigned int LVAR;
4     for (LVAR = 0; LVAR < FPARAM; LVAR++)
5       LVAR += FPARAM[LVAR];
6     printf("%f %d",LVAR/FPARAM,validate(LVAR));
7   }
    Level 3: Data type abstraction.
1   void avg (float FPARAM[], int FPARAM) {
2     DTYPE LVAR = 0;
3     unsigned DTYPE LVAR;
4     for (LVAR = 0; LVAR < FPARAM; LVAR++)
5       LVAR += FPARAM[LVAR];
6     printf("%f %d",LVAR/FPARAM,validate(LVAR));
7   }
    Level 4: Function call abstraction.
1   void avg (float FPARAM[], int FPARAM) {
2     DTYPE LVAR = 0;
3     unsigned DTYPE LVAR;
4     for (LVAR = 0; LVAR < FPARAM; LVAR)
5       LVAR += FPARAM[LVAR];
6     FUNCCALL("%f %d",LVAR/FPARAM,FUNCCALL(LVAR));
7   }
```

〈그림 2. VUDDY 추상화 레벨〉

클론 탐지 단계에서는, 취약 코드들의 지문 집합과 타겟 소프트웨어 전체 함수의 지문 집합을 비교하여 취약 코드 클론을 찾아낸다. 취약 코드 지문 집합 및 타겟 소프트웨어 함수 지문 집합에 대한 정수형의 키를 먼저 비교하고, 만약 동일한 키를 보유하고 있다면, 축소된 검색 범위에 포함된 해시 값의 비교를 수행하여 취약 코드 클론을 탐지한다. 이 길이 기반 필터링(length-based filtering)을 통해 VUDDY는 상수 시간 내에 코드 클론을 탐지할 수 있으므로, 대용량의 오픈 소스 소프트웨어의 취약 코드 클론을 빠르고 효율적으로 탐지할 수 있다.

2.1.3. 실험 결과

주요 실험 결과
높은 정확도(오탐율 0%)로 취약점 탐지 가능 기존 기술 대비 2배 빠른 속도로 취약점을 탐지함으로써 높은 확장성 보장

- VUDDY는 대규모의 OSS의 취약 코드 클론을 빠르고 확장성 있게 탐지할 수 있는 기술이며, 추상화 탐지까지도 지원하여 24% 더 많은 취약 코드 클론을 탐지함
- 높은 확장성: 실행시간 측정시10억 줄의 코드에 대해 14시간 17분이 소요됨, 기존 기술 ReDeBug [5] 과 비교시 약 두 배 빠름
- 높은 정확도: Android 펌웨어 소스코드에서 166개(추상화 적용하지 않음) 및 206개(추상화 적용함)의 취약 코드 클론을 오탐 없이 탐지, ReDeBug 는 탐지 결과 6개중 1개가 오탐으로 확인됨 (17.6%의 오탐율)

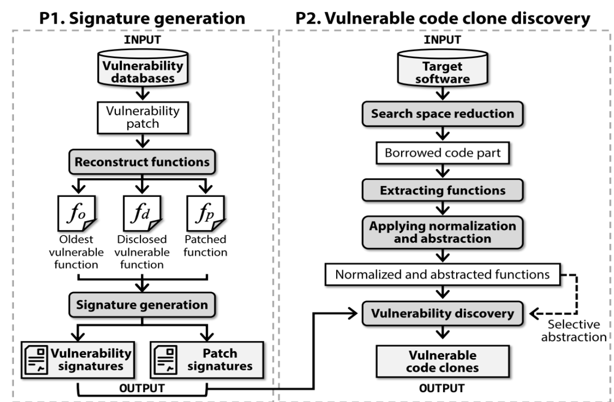
2.2. 수정된 오픈소스 구성요소의 수정된 취약 코드 탐지 기술 (MOVERY)

2.2.1. 문제

다음은 수정된 OSS 구성요소에 포함된, 수정된 취약 코드 클론을 정확하게 탐지할 수 있는 MOVERY [6]에 대해 소개하고자 한다. 앞서 소개한 VUDDY는 구문 수정이 거의 발생하지 않은 취약 코드를 확장성 있게 탐지하는 관점에서 강점을 갖고 있다. 하지만 OSS는 코드 구문이 수정되어 재사용되는 경우가 많고, 취약한 코드 역시 다양한 코드 구문(syntax diversity) 형태로 전파될 가능성이 농후하다. 따라서 다양한 코드 구문으로 전파된 취약코드를 정확하게 탐지하는 기술인 MOVERY를 소개한다.

2.2.2. 방법론

핵심 방법론
취약점 패치에서 핵심적인 코드라인만 추출하여 시그니처 생성



〈그림 3. MOVERY 전체 흐름도〉

[그림 3]에서 볼 수 있듯이, MOVERY는 두 가지 단계로 이루어진다: (1) 시그니처 생성(Signature generation), 그리고 (2) 취약 코드클론(Vulnerable Code Clone, 약어로 VCC로 칭함) 탐지(Vulnerable code clone discovery)이다.

시그니처 생성에서 MOVERY의 핵심 아이디어는, 취약점과 직접적으로 관련이 깊은, 보안 패치 내의 가장 핵심적인 코드라인만 고려하는 것이다. 핵심 기술을 설명하기 앞서 다음의 세 가지 용어를 정의한다. [그림 4]에서 볼 수 있듯이, 공개 시점의 취약한 OSS 함수를 f_d, 패치 함수를 f_p, f_d 와 형태가 다르면서 해당 OSS의 가장 오래된 버전의 취약 함수를 f_o 라고 가정하자.

〈그림 3. MOVERY 전체 흐름도〉

MOVERY는 가장 오래된 취약 함수와 공개된 취약 함수를 대조하여, 핵심적인 코드라인만 추출함으로써 취약점 시그니처를 생성한다. 구체적으로 MOVERY는 다음의 세 가지 핵심 코드라인을 고려한다: 필수 코드 라인(essential code lines), 종속적 코드 라인(dependent code lines), 그리고 제어 흐름 코드 라인(control flow code lines). MOVERY는 최초 보안 패치에서 삭제된 코드 라인 중, f_o 와 f_d 모두 존재하는 코드라인을 필수 코드 라인으로 정의하고 추출한다. 이후 필수 코드 라인이 속한 함수 내에서, 필수 코드 라인과 제어 및 데이터 의존성이 존재하는 코드 라인들을 종속적 코드 라인으로 고려하고 추출한다. 마지막으로 필수 코드 라인이 속한 함수의 시작부터 필수 코드 라인까지 도달하는 경로 내에 존재하는 모든 제어 흐름 코드 라인들을 추출한다. 이 필수, 종속적, 그리고 제어 흐름 코드 라인들을 모아 취약점 시그니처를 생성한다. 다음으로 패치에서 추가된 라인을 필수 코드 라인으로 고려하여, 같은 작업을 진행함으로써 패치

시그니처까지 생성한다.

취약 코드 클론 탐지 단계에서는 타겟 소프트웨어에서 VCC를 탐지한다. MOVERY는 확장성 있는 VCC 탐지를 위해, 다른 OSS로부터 가져온 코드 파트에만 집중하여 타겟 소프트웨어의 검색범위를 축소시킨다. 그런 다음 MOVERY는 취약점 시그니처와, 패치 시그니처를 타겟 소프트웨어의 가져온 코드 파트와 비교한다. 이때 타겟 소프트웨어의 함수가 (1) 취약점 시그니처의 모든 코드라인을 가지고 있고, (2) 패치 시그니처의 어떠한 코드라인도 가지고 있지 않으며, (3) 함수의 구문이 f_o 또는 f_d와 유사할 경우 VCC라고 판단한다.

2.2.3. 실험 결과

주요 실험결과
기존 기술 대비 높은 정확도로 최대 6배 더 많은 취약코드 클론 탐지 가능 알려진 취약코드와 형상이 많이 다른 취약코드까지 탐지 가능

- 4,219개의 CVE (C/C++ 소프트웨어 취약점)를 데이터셋으로 10개의 소프트웨어 대상으로 실험한 결과, 415개의 VCC를 탐지함
- MOVERY는 96% 정밀도(precision)와 96% 재현율(recall)을 보인 반면, 기존 기술인 ReDeBug [5]와 VUDDY [4]는 각각 163개, 72개의 VCC를 탐지했으며, 최대 77% 정밀도와 38% 재현율을 보임
- 탐지된 VCC의 91%는 공개 시점의 취약점 (fd) 코드 형상과 많이 다른 것을 확인함, 이는 수정된 OSS 구성요소의 취약한 코드를 정확하게 찾는 기술의 필요성을 보임. MOVERY는 수정되어 전파된 취약점도 정확하게 탐지해 냄으로 전파된 취약점으로 인한 잠재적 위협을 해결하는 데 활용할 수 있음

2.3. 취약 데이터베이스 구축 기술 (xVDB)

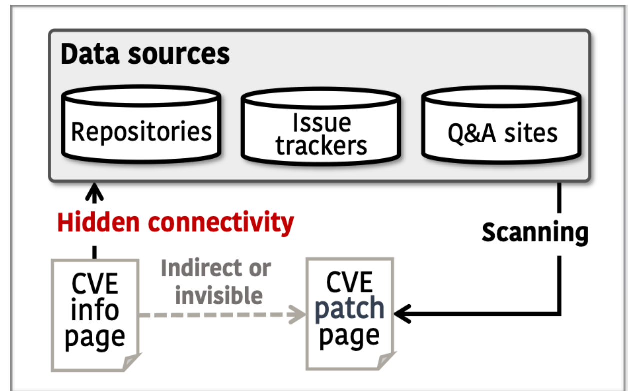
2.3.1. 문제

앞서 소개한 두 개의 취약코드 탐지 기술의 효율성은, 취약점 패치를 수집하여 구축하는 취약점 데이터베이스의 정확성에 의존한다. 따라서, 효과적으로 취약점 데이터베이스를 구축하는 xVDB [7]에 대해 소개한다. 보안 패치는 원데이 취약점(1-day; 취약점에 대해 패치가 공개되었으나 적용되지 않은 취약점)을 탐지하고 수정하는데 중요한 역할을 한다. 이러한 취약 데이터베이스의 중요성에도 불구하고, 기존의 보안 패치 수집에 관련한 연구들은 두 가지의 한계점이 있다: (1) GitHub에서만 보안 패치를 수집하는 한정된 데이터소스 문제와, (2) 취약공개 데이터베이스(예, NVD)의 참고 URL에서 직접 가져올 수 있는 패치만 수집하는 얕은 스캐닝 방식의 문제이다.

2.3.2. 방법론

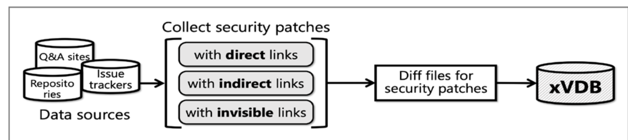
핵심 방법론
공개 취약점 데이터베이스와 보안 패치 사이에 숨겨진 연결성을 활용

이러한 문제를 해결하고자, 본 연구에서는 다양한 데이터소스를 고려하여 알려진 보안 패치를 수집하는 xVDB를 제안한다. 본 연구의 핵심 아이디어는 공개 취약점 데이터베이스와 보안 패치 사이에 숨겨진 연결성을 활용하여 알려진 보안 패치를 찾는 것이다 ([그림 5] 참고).



<그림 5. xVDB 패치 수집 모델>

먼저 한정된 데이터소스 문제를 해결하고자, 소프트웨어 관리 시 가장 많이 사용되고 취약점 이슈가 많이 공유되고 있는 세가지 데이터 소스 선택했다: (1) 저장소(repositories), (2) 이슈 트래커(issue trackers), 그리고 (3) Q&A 사이트(Q&A sites)이다. 다음으로 얕은 스캐닝 문제를 해결하기 위해, 공개 취약점 데이터베이스에서 보안패치를 제공하는 방식에 따라 수집 방식을 분류한다: (1) 직접(direct) 패치 링크 수집 방식, (2) 간접(indirect) 패치 링크 수집 방식, 그리고 (3) 보이지 않는(invisible) 패치 링크 수집 방식이다.



<그림 6. xVDB 전체 흐름도>

따라서 [그림 6]에서 볼 수 있듯이, 세가지 데이터소스에서 각 링크 방식을 통해 보안 패치를 수집하여 xVDB를 구축할 수 있다.

다음으로, 링크에 따른 각 수집 방식에 대해 설명하고자 한다.

- 직접 패치 링크 수집: 공개 취약점 데이터베이스에서 제공하는 참고 URL에서 바로 패치 페이지로 접속 가능한 경우에 보안 패치를 수집할 수 있다. 이러한 수집 방식은 문자열 매칭 등의 과정으로 간단히 이루어지기 때문에,

대부분의 기존 연구들은 직접링크 수집방식만 고려하여 수집했다.

- 간접 패치 링크 수집: 공개 취약점 데이터베이스에서 제공하는 참고 URL에 접속하여 보안패치를 수집하는 방식이다. 구체적으로, 웹사이트 내에 보안 패치 링크를 수집하거나, 보안 패치를 수집할 수 있는 힌트 (예, 버그 아이디 혹은 커밋 아이디)를 추출하는 방식을 통해 수집할 수 있다. 해당 경우 저장소 및 이슈 트래커에 적용할 수 있다.
- 보이지 않는 패치 링크 수집: 공개 취약점 데이터베이스에서 패치 페이지로 직간접적인 링크가 존재하지 않는 경우에 수집하는 방식이다. 이는 저장소 및 이슈트래커에서 커밋 메시지에 "CVE-20" 키워드가 있고 커밋에 소스 코드의 변화를 확인하여 보안 패치로 볼 수 있는 특징이 하나 이상 발견될 시 수집한다. 여기서 DICOS [8] 연구에서 발견한 보안 특징을 활용하여, 코드에 제어흐름이 변경되었거나, 보안에 민감한 API (예, strcpy)의 변경이 있는지 확인한다. 또한, Q&A 사이트에 경우 DICOS [8] 연구를 적용하여 안전하지 않은 포스트를 바로 수집할 수 있다.

2.3.3. 실험 결과

주요 실험결과
기존 기술 대비 약 4배 많은 CVE 패치를 탐지하고, 추가적으로 개발자 사이트에서 안전하지 않은 포스트까지 수집 수집된 CVE 패치 중 절반 이상이 간접 및 보이지 않는 링크로 수집됨

- 저장소와 이슈 트래커에서 12,432 CVE 패치(C, C++, Java, JavaScript, Go, Python 고려)를 수집, Q&A 사이트에서 12,458개의 안전하지 않은 포스트(C, C++, Android 포스트 고려)를 수집
- 이는 기존 기술들(예, V0Finder [11])에서 수집한 CVE 취약점 패치보다 4배이상 더 많은 수치임
- 연도별로 수집된 CVE 패치 중 절반 이상이 간접 및 보이지 않는 링크로 수집되었으며, 또한 수집된 보안 패치의 출처를 확인한 결과, 상당수의 보안 패치는 이슈 트래커를 통해 수집됨
- 이를 통해, 다양한 데이터소스를 고려하고 직접적인 연결 뿐만아니라 간접 혹은 보이지 않는 연결까지 고려한 xVDB의 필요성을 보임

2.4. OSS 구성요소 탐지 기술 (CENTRIS)

2.4.1. 문제

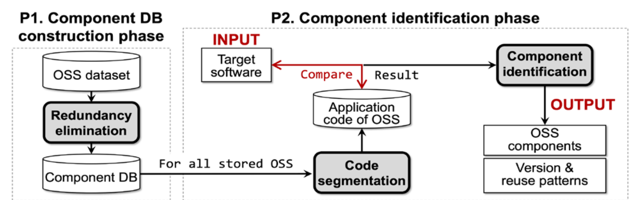
다음은 OSS 구성요소를 정확하고 확장성 있는 방식으로 탐지 할 수 있는 CENTRIS [9]에 대해 소개하고자 한다. 개발자들은 재사용중인 OSS 구성요소를 체계적으로 관리함으로써 보안 위협을 완화할 수 있다. 따라서 OSS 위험을 완화하기 위한 가장 첫번째 걸음으로, OSS 구성요소를 식별해야 할 필요가 있다.

하지만 정확한 OSS 식별은 다음의 두가지 이유로 도전적이다: (1) 수정된 OSS 재사용으로 인한 많은 미탐, 그리고 (2) 중첩된 OSS 구성요소로 인한 많은 오탐 발생. 많은 개발자들은 OSS를 재사용할 때, 필요한 기능만 활용하거나 (부분 재사용), 일부 코드 및 구조를 변경하여 재사용한다. 또한 재사용되는 OSS는 많은 OSS의 하위 컴포넌트를 포함하고 있고, 이러한 요인들은 OSS 구성요소 탐지 기술의 정확성과 확장성을 저해한다.

2.4.2. 방법론

핵심 방법론
높은 확장성을 위한 중복 제거 (redundancy elimination) 기술 고안 높은 정확도를 위한 코드 세분화 (code segmentation) 기술 활용

본 연구에서는 대규모의 OSS 집합으로부터 특정 소프트웨어가 재사용 중인 OSS 구성요소 리스트와 그들의 재사용 패턴을 확장성 있고 정확한 방식으로 탐지 할 수 있는 CENTRIS를 제안한다. CENTRIS 핵심 기술은 총 두 단계로 구성된다: (1) 중복제거(redundancy elimination)기반의 구성요소 데이터베이스(component DB) 구축 단계와, (2) 코드 세분화(code segmentation) 기법을 통해 수정된 구성요소까지도 정확하게 탐지해내는 구성요소 식별 단계이다. ([그림 7] 참고)

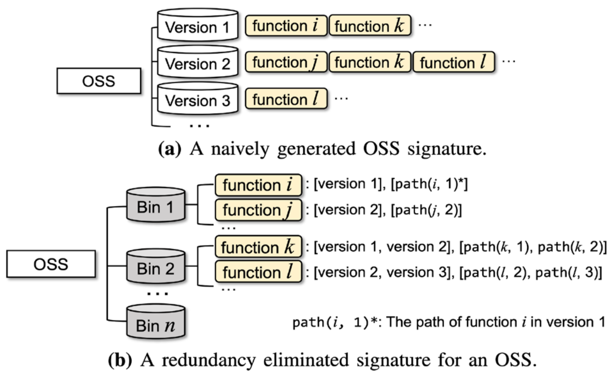


〈그림 7. CENTRIS 전체 흐름도〉

구성요소 데이터베이스 구축 단계에서는 탐지 확장성을 높이기 위해 OSS들의 특성에 주목했다. OSS 버전이 업데이트 될 때 모든 소스코드가 매번 새로 작성되지는 않는다. 따라서 OSS내 여러 버전 사이에는 공통적인 코드 부분이 항상 존재하고, 이러한 공통부분들은 OSS 구성요소를 탐지할 때 중복되어 매칭에 여러 번 사용된다. CENTRIS는 이러한 불필요한 중복 매칭들을 제거하여 확장성을 극대화했다. 등장빈도에 관계없이 OSS의 모든 버전에서 한 번 이상 등장한 함수의 해시값을 key로, 해당 함수가 속하는 버전들을 value로 하여 딕셔너리(dictionary) 자료구조에 저장한다. 이후, 함수의 등장빈도에 따른 Bin에 전체 함수를 매핑한다 ([그림8] 참고). 수집된

모든 OSS에 대해 해당 작업을 수행하고 난 후 생성된 데이터들을 모아 component DB를 구축한다.

구성요소 식별 단계에서는 재사용 중인 OSS 구성요소를 식별한다. 이를 위해, 우리는 OSS의 코드 베이스를 고유한 코드 부분(application code)과 다른 소프트웨어의 코드가 포함 되어있는 빌려온 코드 부분(borrowed code)으로 세분화한다. 수정된 구성요소를 탐지할 때 발생하는 대부분의 오탐은 이 빌려온 코드 부분으로 인해 발생한다. 따라서, 우리는 OSS의 고유한 코드 부분만을 타겟 소프트웨어와의 매칭에 사용하여 구성요소를 식별한다.



〈그림 8. 중복제거를 통한 인덱싱 결과 예시〉

특정 OSS의 고유한 코드 부분을 탐지하기 위해, 우리는 먼저 해당 OSS (S라고 하자)와 component DB에 존재하는 모든 다른 OSS (X라고 하자) 사이의 코드유사도를 측정한다. 이때 S와 X에 공통으로 존재하는 함수 중, X에서 더 일찍 생성된 함수들의 집합을 G라고 하자. 만약 G에 속하는 함수 개수를 X의 전체 함수 개수로 나눈 값이 임계 값(10%) 이상이라면 ([수식 1] 참고), 우리는 S의 빌려온 코드 부분에 X가 포함 되어있다고 판단한다. 즉 X의 전체 함수 중 충분히 많은 부분이 S의 코드 베이스에 포함 되어있으면서, 해당 함수는 S보다 X에서 더 먼저 생성되었기 때문에, S가 X를 재사용하고 있다고 판단하는 것이다.

$$\phi(S, X) = \frac{|G|}{|X|},$$

$$\text{where } G = \{f \mid (f \in (S \cap X)) \wedge (birth(f, X) \leq birth(f, S))\}$$

〈수식 1. S의 고유한 코드부분을 찾기 위한 S와 X의 유사도 비교 수식〉

다음으로 S의 고유한 코드 부분만을 얻기 위해, S의 전체 함수 집합에서 G에 포함되는 함수들을 모두 제거한다. 이 과정을 component DB의 모든 X에 대해 진행하고 나면, S에는 고유한 코드 부분에 속하는 함수들만 남게 된다. 최종적으로, 타겟 소프트웨어를 component DB의 모든 OSS의 고유한 코드 부분과 비교하여, 코드 유사도가 10% 이상인 OSS들을 타겟 소프트웨어의 구성요소로 판단한다.

2.4.3. 실험 결과

주요 실험결과
대부분(95%)의 OSS 구성요소가 수정되어 재사용됨에도, 91% 정밀도 및 94%의 재현율로 OSS 구성요소 탐지 가능 100 만 코드라인으로 구성된 소프트웨어의 구성요소를 1분이내에 탐지 가능

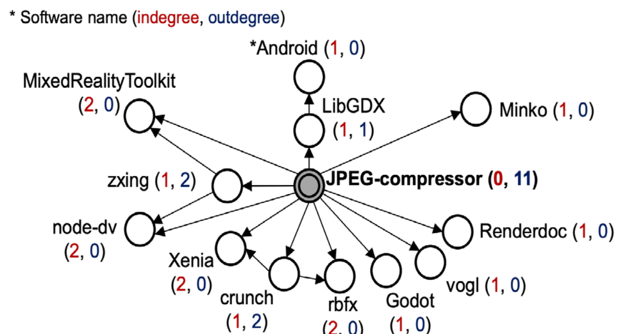
- 10,241개 소프트웨어 (C/C++ 소프트웨어)와 나머지 OSS들과 비교하여 OSS 구성요소를 탐지한 결과, 4,434개의 소프트웨어(44%) 가 다른 OSS를 최소 하나 이상 재사용중이며, 탐지된 OSS구성요소의 95%는 하나 이상의 코드 및 구조 수정과 함께 재사용됨
- 높은 정확도: 91%의 정밀도와 94%의 재현율로 OSS 구성요소들을 탐지(관련 기술인 OSSPolice [10]의 경우 약 82%의 정확도를 보임)
- 높은 확장성: 매칭에 소요된 시간은 전체 약 100시간으로, 하나의 소프트웨어와 10,240개의 다른 오픈소스 소프트웨어를 비교하는 데에 평균 1분 이내의 시간이 소요됨. 이는 CENTRIS가 충분히 빠르다는 것을 입증
- OSS의 하위 구성요소까지 정확하게 식별할 수 있는 CENTRIS를 통해, 개발자들은 OSS를 체계적으로 관리하여 보안 위협을 완화할 수 있음

2.5. 취약점 원점 소프트웨어 탐지 기술 (V0Finder)

2.5.1. 문제

다음은 취약점의 올바른 원점 정보를 탐지 할 수 있는 V0Finder [11] 에 대해 소개하고자 한다. 취약점의 최초 근원지(Vulnerability Zero, 약어로 VZ로 칭함)가 잘못 보고된 경우, 이는 패치 배포의 지연 및 개발자들에게 전파된 취약점을 간과하게 만드는 등의 보안 문제를 야기한다.

하나의 예제로, [그림 9]는 CVE-2017-0700 취약점의 취약점 전파 그래프를 나타낸다. 노드는 해당 취약점을 갖고 있



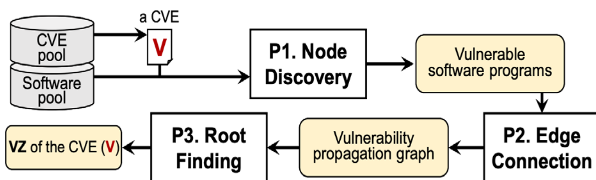
〈그림 9. CVE-2017-0700 취약점 전파 그래프 예제〉

는 소프트웨어들이며, 그래프의 간선은 취약점 전파 방향을 의미한다. 취약점의 올바른 원점은 JPEG-compressor이지만, 공개 취약점 데이터베이스에서는 해당 취약점의 원점이 Android로 알려져 있다. 즉, JPEG-compressor를 사용하는 LibGDX, Godot 등은 취약점이 존재하는지 여부조차 알 수 없고, 적시에 보안 패치를 적용하기도 어렵다.

2.5.2. 방법론

핵심 방법론
취약점 전파 그래프를 생성하여 올바른 원점을 탐지

본 연구에서는 올바른 취약점 원점 정보를 탐지하는 V0Finder를 제안한다. V0Finder의 핵심 아이디어는 그래프 기반 방식을 활용하여 취약점 전파 그래프를 생성하는 것이다. 생성된 그래프의 루트(root)를 찾음으로써 취약점의 올바른 원점을 탐지하는 것이다. 구체적으로 V0Finder는 세가지 단계로 구성된다: (1) 특정 취약점을 가지고 있는 소프트웨어 탐지(노드(node) 탐지), (2) 취약 소프트웨어 중 전파된 경로 식별(간선(edge) 연결), 그리고 (3) 전파 경로 역 추적(backtracking)을 통한 취약 원점 탐지(루트 탐지) 단계이다.



〈그림 10. V0Finder 전체 흐름도〉

첫번째 단계에서 취약 코드 클론 탐지기술을 활용하여 취약 소프트웨어를 탐지한 후 노드를 구성한다. 구문이 수정된 취약한 코드클론까지 탐지하기 위해, 지역 민감 해싱(Locality Sensitive Hashing, 약어로 LSH 라 칭함)을 활용한다.

두번째 단계는 탐지된 취약 소프트웨어(노드) 사이의 간선 연결을 위한 단계이다. 여기서 취약 소프트웨어 사이의 전파 방향을 식별하기 위해 재사용 관계에 주목한다. 예를 들어, S1 소프트웨어가 S2를 재사용하면서 두 소프트웨어에 같은 취약점이 포함되어 있는 경우, S2의 취약점이 S1으로 전파된 것으로 판단할 수 있다. 따라서, V0Finder는 재사용 관계를 추적하기 위해 다음 세가지 조건 중 하나이상이 만족되는지를 판단한다: (1) S2의 전체 코드베이스를 S1이 포함하고 있는지, (2) S2의 소스코드의 파일 위치가 S1의 파일 위치에 속하는지, 그리고 (3) S1이 S2의 메타데이터 파일(예, OSS내에 README, LICENSE, COPYING 파일)들을 포함하고 있는지 확인한다.

세 번째 단계에서는 취약점 전파 그래프의 루트를 확인함으로써, 올바른 취약점의 원점 소프트웨어를 탐지한다. [그림 9]에서 볼 수 있듯이, JPEG-compressor 가 그래프의 루트이므로, 해당 소프트웨어의 VZ인 것을 알 수 있다.

2.5.3. 실험 결과

주요 실험결과
높은 정확도(98% 정밀도 및 95% 재현율)로 취약점의 원점 탐지 가능 현재의 공개 취약점 데이터베이스가 잘못된 원점정보를 제공하는 96개의 CVE 취약점을 탐지

- 높은 정확도: 5,617개의 CVE (C/C++ 소프트웨어 취약점)를 데이터 셋으로 하여 찾은 VZ와 CPE를 비교하는 방식으로 정확도를 측정했을 때, V0Finder는 98%의 정밀도 및 95%의 재현율을 보임
- 96 개의 CVE (1.7%)가 공개 취약점 데이터베이스에서 잘못된 원점정보를 제공함을 확인
- 올바르지 않은 VZ로 취약점이 보고되었을 경우, (1) 취약점은 3배 더 많은 소프트웨어 프로그램에 전파되었고, (2) 전파된 취약점의 36%만 개발자들이 탐지하는데 성공했으며 (올바른 VZ는 85% 탐지), 그리고 (3) 취약점을 탐지하고 패치하는 시간이 더 오래 걸림을 확인 (올바른 VZ의 경우 308일이 소요되는 반면, 올바르지 않은 VZ는 521일 소요됨)

2.6. 개발자 사이트의 안전하지 않은 포스트 탐지 기술 (DICOS)

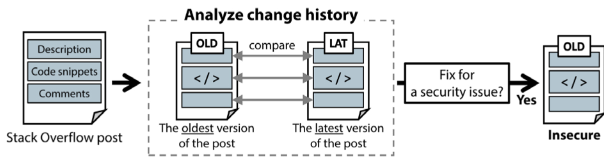
2.6.1. 문제

본 장에서는 개발자 사이트에서 안전하지 않은 코드 스니펫을 탐지하는 기술인 DICOS [8]에 대해 소개하고자 한다. 개발자들은 보통 소프트웨어 개발 시 Stack Overflow [3]와 같은 개발자 사이트를 많이 활용하게 되는데, 적절한 코드 이해 없이 재사용하게 될 경우 해당 코드가 삽입된 소프트웨어의 보안을 손상시키고 취약점 전파 문제로 이어질 수 있다. 이러한 문제는 개발자 사이트의 안전하지 않은 포스트를 탐지함으로써 위협을 방지할 수 있는데, 이를 해결하고자 했던 기존 연구들은 두 가지 이유로 제한적이다: (1) 보안 관련 API 가 포함된 코드 스니펫만 고려한 미탐, (2) 불필요하게 많은 특징(feature) 선택으로 인한 오탐이다.

2.6.2. 방법론

핵심 방법론
개발자 사이트의 사용자 토론을 활용하여 안전하지 않은 코드스니펫을 탐지

3) https://stackoverflow.com/



〈그림 11. DICOS 전체 흐름도〉

이를 해결하고자, 본 연구에서는 안전하지 않은 코드스니펫을 발견하는 DICOS를 제안한다. DICOS의 핵심 아이디어는 개발자 사이트의 사용자 토론(user discussion)을 활용하여 안전하지 않은 코드스니펫을 찾는 것이다. 포스트의 답변자들은 댓글 등을 통해 그들의 코드스니펫에 보안적 이슈가 있는 것을 확인했을 때, 포스트 수정 기록과 함께 포스트를 안전한 버전으로 수정한다. DICOS는 포스트의 핵심 구성요소의 변화된 히스토리를 분석하여 보안 패치의 특징이 나타나는 포스트를 찾는다. 구체적으로, DICOS는 세가지 단계로 안전하지 않은 포스트를 찾는다: (1) 포스트의 히스토리 변화 추출, (2) 포스트 히스토리 분석, 그리고 (3) 안전하지 않은 코드스니펫 판별이다 ([그림 11] 참고).

첫번째 단계에서, DICOS는 포스트의 핵심 구성요소의 변화된 히스토리를 추출한다. 여기서 포스트의 핵심 구성요소란 보안적 이슈로 포스트가 변할 때 특징적으로 변하는 세가지 구성요소인 설명(description), 코드스니펫(code snippet), 댓글(comment)을 말한다 ([그림 12] 참고). 이때, 포스트의 가장 오래된 버전과 최신 버전 사이의 차이(즉, 디프(diff)를 지칭)를 추출하며, 이때 추출된 디프가 두 버전의 모든 변경 패턴을 포함하고 있다고 가정한다.

두번째 단계에서, DICOS는 추출된 디프에 보안 패치의 특징이 나타나는지 확인함으로서 변화된 히스토리를 분석한다. 여기서 보안 패치 특징은 사전 연구를 통해 세가지로 정의했다: (1) 보안에 민감한 API들의 변화, (2) 보안과 관련한 키워드의 변화, 그리고 (3) 제어흐름의 변화이다. 따라서, 코드 스니펫의 디프에서 보안에 민감한 API가 삭제되었는지, 제어 흐름이 수정되거나 추가되었는지 확인하고, 포스트의 설명과 댓글에 보안 관련한 키워드가 추가되었는지 확인한다.

How do I trim leading/trailing whitespace in a standard way?

187

Is there a clean, preferably standard method of trimming leading and trailing whitespace from a string in C? I'd roll my own, but I would think this is a common problem with an equally common solution.

c string whitespace trim

Question

178

If you can modify the string:

Description

```
char *trimwhitespace(char *str)
{
  char *end;

  // Trim leading space
  while(isspace((unsigned char)*str)) str++;

  if(*str == 0)  // All spaces?
    return str;

  // Trim trailing space
  end = str + strlen(str) - 1;
  while(end > str && isspace((unsigned char)*end)) end--;

  // Write new null terminator character
  end[1] = '\0';

  return str;
}
```

Code snippet

Answer

12 @Raj: There's nothing inherently wrong with returning a different address from the one that was passed in. There's no requirement here that the returned value be a valid argument of the `free()` function. Quite the opposite -- I designed this to avoid the need for memory allocation for efficiency. If the passed in address was allocated dynamically, then the caller is still responsible for freeing that memory, and the caller needs to be sure not to overwrite that value with the value returned here.

3 You have to cast the argument for `isspace` to `unsigned char`, otherwise you invoke undefined behavior.

Comments

〈그림 12. Stack Overflow 답변 게시물 예제〉

마지막 단계에서, 분석한 결과를 기반으로 한 가지 포스트에서 두 가지 이상의 특징이 탐지되면 안전하지 않은 포스트로 판단한다. DICOS는 단일 특징이 아닌 효율적인 특징을 조합하여 판단하기 때문에, 다양한 보안 이슈를 다룰 수 있다.

2.6.3. 실험 결과

주요 실험결과
높은 정확도(91% 정밀도, 93% 재현율)로 안전하지 않은 코드 스니펫 탐지 최신 버전 OSS 중 8%에 안전하지 않은 코드 스니펫이 전파됨

- Stack Overflow내 1,958,283개의C/C++/Android 포스트를 대상으로 실험을 진행했을 때, 12,458개의 안전하지 않은 포스트(14,719개 코드 스니펫)을 91% 정밀도 및 93% 재현율로 탐지 가능
- 개발자 사이트에서 (1) 매년 2%가량의 안전하지 않은 포스트가 게재되며, (2) 버퍼 오버플로우, 예기치 못한 동작 등 다양한 유형의 취약점이 탐지되었고, (3) 최신 버전의 OSS 중 8%(151개)에 안전하지 않은 코드스니펫이 전파됨, 이를 통해 자동화된 방법으로 안전하지 않은 코드스니펫을 탐지하여 보안 위협을 완화할 수 있는 DICOS의 필요성을 시사함

Ⅲ. 결론 (Conclusion)

OSS 활용의 이점으로 OSS 재사용 수가 꾸준히 증가하는 반면, 재사용 중인 OSS를 적절히 관리하지 않으면 취약점 전파 및 공급망 공격 등 다양한 보안 문제를 야기한다. 이러한 문제를 방지하기 위해 OSS 구성요소를 꾸준히 관리하여 안전한 OSS 버전을 사용하거나, 취약한 코드를 탐지 및 수정함으로써 관리해 나가야한다.

결론적으로, 본 고에서는 OSS 재사용으로 인한 보안 위협을 해결하기위해 다음의 세가지 대응 방안을 제안한다.

- 소스코드 레벨의 취약 코드를 탐지하는 VUDDY, MOVERY를 통해 대규모의 OSS에서 전파 취약점을 탐지하여 잠재적 보안 위협을 완화해야 함
- 취약점 정보를 확보하기 위한 연구인 xVDB, DICOS 등을 활용하여 풍부한 취약점 데이터베이스를 구축 해야함
- 오픈소스 구성요소 및 취약점 원점 식별기술인

CENTRIS, V0Finder 연구를 통해 구성요소를 정확히 식별하고 취약점의 정확한 원점 소프트웨어를 찾아냄으로써 OSS를 체계적으로 관리해 나가야함

Acknowledgement

본 연구는 과학기술정보통신부 및 정보통신기획평가원의 지역전략산업 융합보안 핵심인재 양성 사업(No.2022-0-01198), 블록체인 플랫폼 보안취약점 자동분석 기술개발 사업(No.2019-0-01697), 및 SW공급망 보안을 위한 SBOM 자동생성 및 무결성 검증기술 개발 사업(No.2022-0-00277) 결과로 수행되었음.

REFERENCES

[1] Synopsys Inc. Open Source Security and Risk Analysis Report. 2022; Available from: https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html.

[2] Synopsys Inc. State of the Software Supply Chain Report. 2021; Available from: https://www.sonatype.com/resources/white-paper-2021-state-of-the-software-supply-chain-report-2021.

[3] The United States Department of Commerce. The Minimum Elements For a Software Bill of Materials (SBOM). 2021; Available from: https://www.ntia.doc.gov/files/ntia/publications/sbom_minimum_elements_report.pdf.

[4] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh, VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. 2017. p. 595--614.

[5] Jiyong Jang, Abeer Agrawal, and David Brumley, ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. 2012. p. 48--62.

[6] Seunghoon Woo, Hyunji Hong, Eunjin Choi, and Heejo Lee. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. in 31st USENIX Security Symposium (USENIX Security 22). 2022. Boston, MA: USENIX Association.

[7] Hyunji Hong, Seunghoon Woo, Eunjin Choi, Jihyun Choi, and Heejo Lee, xVDB: A High-Coverage Approach for Constructing a Vulnerability Database. IEEE Access, 2022.

[8] Hyunji Hong, Seunghoon Woo, and Heejo Lee, DICOS: Discovering Insecure Code Snippets from Stack Overflow Posts by Leveraging User Discussions. 2021. p. 194--206.

[9] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE). 2021. IEEE.

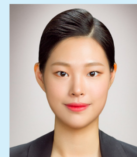
[10] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee, Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. 2017. p. 2169--2185.

[11] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich, V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. 2021. p. 3041--3058.

Biographies

Seunghoon Woo (우승훈)

2016: 고려대학교 컴퓨터공학 학사
2022: 고려대학교 컴퓨터공학 석,박사 통합
2022-현재: LABRADOR LABS Inc. CSO
2022-현재: 고려대학교 소프트웨어보안연구소(CSSA) 연구 교수
주요 관심 분야: 오픈소스 소프트웨어 보안, 소프트웨어 구성요소 분석, 소프트웨어 취약점 탐지, 코드클론 탐지
Email: seunghoonwoo@korea.ac.kr



Hyunji Hong (홍현지)

2020: 한신대학교 컴퓨터공학 학사
2020-현재: 고려대학교 컴퓨터공학 석사과정
주요 관심 분야: 오픈소스 소프트웨어 보안, 취약점 탐지 및 분석
Email: hyunji_hong@korea.ac.kr

Heejo Lee (이희조)

1993: 포항공과대학교 컴퓨터공학 학사
1995: 포항공과대학교 컴퓨터공학 석사
2000: 포항공과대학교 컴퓨터공학 박사
2000-2001: Purdue Univ. CS 및 CERIAS 연구원
2001-2003: 안철수연구소 CTO
2004-현재: 고려대 컴퓨터공학 교수
2015-현재: 고려대 소프트웨어보안연구소(CSSA) 연구소장
2018-현재: LABRADOR LABS Inc. Co-CEO
주요 관심 분야: DDoS 방지, 봇넷 탐지, 악성코드 탐지
Email: heejo@korea.ac.kr