관 인 생 략

출 원 번 호 통 지 서

출 원 일 자	2025.09.24
특 기 사 항	심사청구(유) 공개신청(무) 참조번호(W25P0335KR)
출 원 번 호	10-2025-0137821 (접수번호 1-1-2025-1091065-58) (DAS접근코드D6A3)
출원인 명칭	고려대학교 산학협력단(2-2004-017068-0)
대리인 성명	특허법인 위솔(9-2022-100001-9)
발명자 성명	이희조 김두영
발명의 명칭	보안 패치를 이용한 코드 라인의 의존성 기반 의미적 쌍 매핑을 통한 소프트웨어 취약 버전 식별 방법 및 장치

특 허 청 장

<< 안내 >>

1. 귀하의 출원은 위와 같이 정상적으로 접수되었으며, 이후의 심사 진행상황은 출원번호를 이용하여 특허로 홈페이지(www.patent.go.kr)에서 확인하실 수 있습니다.
2. 출원에 따른 수수료는 접수일로부터 다음날까지 동봉된 납입영수증에 성명, 납부자번호 등을 기재하여 가까운 은행 또는 우체국에 납부하여야 합니다.
※ 납부자번호 : 0131(기관코드) + 접수번호
3. 귀하의 주소, 연락처 등의 변경사항이 있을 경우, 즉시 [특허고객번호 정보변경(경정), 정정신고서]를 제출하여야 출원 이후의 각종 통지서를 정상적으로 받을 수 있습니다.
4. 기타 심사 절차(제도)에 관한 사항은 특허청 홈페이지를 참고하시거나 특허고객상담센터(☎ 1544-8080)에 문의하여 주시기 바랍니다.
※ 심사제도 안내 : https://www.kipo.go.kr-지식재산제도

【서지사항】

【서류명】 특허출원서

【참조번호】 W25P0335KR

【출원구분】 특허출원

【출원인】

【명칭】 고려대학교산학협력단

【특허고객번호】 2-2004-017068-0

【대리인】

【명칭】 특허법인 위솔

【대리인번호】 9-2022-100001-9

【지정된변리사】 나강은, 김경용, 강현모, 편진호, 이영규

【발명의 국문명칭】 보안 패치를 이용한 코드 라인의 의존성 기반 의미적 쌍 매핑을 통한 소프트웨어 취약 버전 식별 방법 및 장치

【발명의 영문명칭】 METHOD AND APPARATUS FOR IDENTIFYING VULNERABLE SOFTWARE VERSIONS THROUGH SEMANTIC PAIR MAPPING BASED ON CODE LINE DEPENDENCY USING SECURITY PATCHES

【발명자】

【성명】 이희조

【성명의 영문표기】 Heejo Lee

【국적】 KR

【주민등록번호】

【우편번호】

【주소】

【거주국】 KR

【발명자】

【성명】 김두영

【성명의 영문표기】 Kim Duyeong

【국적】 KR

【주민등록번호】

【우편번호】

【주소】

【거주국】 KR

【출원언어】 국어

【심사청구】 청구

【이 발명을 지원한 국가연구개발사업】

【과제고유번호】 2710068997

【과제번호】 II220277

【부처명】 과학기술정보통신부

【과제관리(전문)기관명】 정보통신기획평가원

【연구사업명】 정보보호핵심원천기술개발(R&D,정보화)

【연구과제명】 SW공급망 보안을 위한 SBOM 자동생성 및 무결성 검증기술 개발

【과제수행기관명】 고려대학교산학협력단

【연구기간】 2025.01.01 ~ 2025.12.31

【이 발명을 지원한 국가연구개발사업】

【과제고유번호】 2710069045

【과제번호】 00440780

【부처명】 과학기술정보통신부

【과제관리(전문)기관명】 정보통신기획평가원

【연구사업명】 정보보호핵심원천기술개발(R&D,정보화)

【연구과제명】 SW공급망 전주기 보안 내재화를 위한 SBOM과 VEX 연계 기반 취약점 통합 관리 플랫폼 기술

【과제수행기관명】 고려대학교산학협력단

【연구기간】 2025.01.01 ~ 2025.12.31

【이 발명을 지원한 국가연구개발사업】

【과제고유번호】 2710074877

【과제번호】 II201819

【부처명】 과학기술정보통신부

【과제관리(전문)기관명】 정보통신기획평가원

【연구사업명】 디지털기술선도핵심인재양성(R&D)

【연구과제명】 ICT명품인재양성(고려대학교)

【과제수행기관명】 고려대학교산학협력단

【연구기간】 2025.01.01 ~ 2025.12.31

2025-09-24

【취지】 위와 같이 특허청장에게 제출합니다.

대리인 특허법인 위솔 (서명 또는 인)

【수수료】

【출원료】	0	면	원
【가산출원료】	43	면	원
【우선권주장료】	0	건	원
【심사청구료】	10	항	원
【합계】		원	
【감면사유】	전담조직(50%감면)[1]		
【감면후 수수료】		원	

【첨부서류】 1.기타첨부서류_1통

1 : 기타첨부서류

PDF 파일 첨부

【발명의 설명】

【발명의 명칭】

보안 패치를 이용한 코드 라인의 의존성 기반 의미적 쌍 매핑을 통한 소프트웨어 취약 버전 식별 방법 및 장치{METHOD AND APPARATUS FOR IDENTIFYING VULNERABLE SOFTWARE VERSIONS THROUGH SEMANTIC PAIR MAPPING BASED ON CODE LINE DEPENDENCY USING SECURITY PATCHES}

【기술분야】

【0001】본 발명은 소프트웨어 보안 기술 분야에 관한 것으로, 보다 구체적으로는 보안 패치 정보를 이용하여 취약점 발생에 직접적으로 관련된 코드 라인과, 이와 데이터 또는 제어 의존성에 의해 의미적으로 연결된 코드 라인을 의미적 쌍으로 매핑함으로써 소프트웨어의 취약 버전을 효과적으로 식별하는 기술에 관한 것이다.

【발명의 배경이 되는 기술】

【0002】소프트웨어 보안 취약점은 사이버 공격의 주요 진입점으로 작용하며, 특히 C/C++ 언어 기반 소프트웨어에서는 메모리 관리 오류나 포인터 사용상의 취약점과 같은 저수준 보안 결함이 빈번하게 발생하여 보안 사고의 원인이 된다. 이러한 보안 취약점을 조기에 식별하고, 영향을 받는 소프트웨어 버전을 명확히 특정하는 것은 보안 대응의 핵심 과제이다

【0003】보안 취약점을 탐지하기 위해 현재 널리 활용되는 접근 방식으로는 국가 취약점 데이터베이스(National Vulnerability Database, NVD)와 같은 공개 데이터베이스에 의존하거나, 패치에서 삭제된 코드 라인을 추적하는 알고리즘 기반 도구가 사용되고 있다. 이들 기술은 비교적 단순한 코드 차이를 기반으로 취약점의 존재 여부를 파악하고, 특정 소프트웨어 버전이 취약한지 여부를 보고하는 역할을 한다. 실제로 이러한 방법은 보안 전문가들이 취약점 관리 및 대응 범위를 설정하는 과정에서 중요한 참고자료로 활용되어 왔다.

【0004】그러나 기존 기술은 몇 가지 한계점을 갖는다. 패치된 코드와 삭제된 코드를 단순히 구문 차이로만 비교하는 경우, 코드의 문법적 표현이 변경되었을 때 취약점 여부를 올바르게 식별하지 못한다. 또한 함수의 진화 과정을 추적하지 못하거나 코드 내 의미적 의존 관계를 고려하지 못하기 때문에, 특정 버전에서 취약점이 여전히 존재하는지 정확하게 판별하기 어렵다. 이로 인해 취약 버전의 시작 시점을 과잉 보고하거나, 반대로 종료 버전을 잘못 판단하여 안전하지 않은 버전을 안전한 것으로 오인하는 문제가 발생한다. 나아가 중간 버전에만 패치가 적용된 특수한 경우 역시 반영하지 못해 실제 보안 상태와 데이터베이스 간 괴리가 생기는 한계도 존재한다.

【0005】이와 같은 문제는 보안 대응 과정에서 치명적인 결과를 초래할 수 있다. 실제로 잘못된 버전 정보는 관리자가 취약한 시스템을 안전하다고 착각하게 하여 공격 표면을 방치하거나, 반대로 안전한 시스템에 불필요한 보안 조치를 반복하게 만드는 비효율을 초래한다.

【0006】 따라서 코드의 구문적 차이만이 아니라 취약점의 원인이 되는 핵심 라인과 그와 의미적으로 연결된 코드 라인을 함께 분석하여 보다 정밀하게 취약 버전을 특정할 수 있는 새로운 기술적 접근이 필요하다.

【선행기술문헌】

【특허문헌】

【0007】 **(특허문헌 0001)** 대한민국 등록특허공보 제10-2159299호

【발명의 내용】

【해결하고자 하는 과제】

【0008】 본 발명이 해결하고자 하는 과제는 소프트웨어 보안 취약점에 대하여 영향을 받는 버전의 범위를 보다 정확하고 신뢰성 있게 특정할 수 있는 기술을 제공하는 데 있다. 이를 위해, 본 발명은 단순한 구문 차이에 의존하지 않고, 취약점의 원인이 되는 핵심 코드 라인과 이와 의미적으로 연결된 라인을 함께 분석하여 취약 여부를 판별할 수 있도록 함으로써, 코드의 표현이 일부 변형되더라도 취약성을 일관되게 식별할 수 있는 기술을 제안하고자 한다.

【0009】 또한, 본 발명은 함수의 계보를 추적하여 각 버전별 코드의 진화 과정을 분석함으로써, 특정 함수가 어느 시점에서 취약하고 어느 시점에서 패치가 적용되어 안전해지는지를 정밀하게 구분할 수 있도록 하는 것을 목표로 한다. 이를 통해 각 취약점에 대응하는 시작 버전과 종료 버전을 체계적으로 도출하고, 보안

데이터베이스의 정보와 실제 소프트웨어의 상태 간 괴리를 줄여 보안 대응의 효율성과 신뢰성을 향상시키고자 한다.

【0010】 한편, 본 발명의 기술적 과제들은 이상에서 언급한 기술적 과제들로 제한되지 않으며, 언급되지 않은 또 다른 기술적 과제들은 아래의 기재로부터 통상의 기술자에게 명확하게 이해될 수 있을 것이다.

【과제의 해결 수단】

【0011】 일 실시예에 따른 프로세서에 의해 동작하는 소프트웨어 취약 버전 식별 장치가 수행하는 방법에 있어서, 분석 대상이 되는 취약점 정보를 입력 받아, 상기 취약점 정보에 대응하는 보안 패치 정보 및 소스코드를 획득하는 동작; 상기 보안 패치 정보를 기초로 상기 소스코드에서 보안 패치로 인해 삭제된 코드 라인을 포함하는 함수를 취약 함수로, 보안 패치로 인해 추가된 코드 라인을 포함하는 함수를 패치 함수로 분류하는 동작; 상기 취약 함수 및 상기 패치 함수 각각을 보안과 연관된 필수 라인으로 지정하고, 어느 하나의 제1 필수 라인에서 사용된 변수를 선언하는 라인, 상기 변수에 값을 할당하는 라인, 또는, 상기 제1 필수 라인의 실행 여부를 결정하는 라인을 제1 의존 라인으로 지정하는 동작; 상기 제1 필수 라인과 상기 제1 의존 라인을 매핑하여 취약점이 의미적으로 연결되어 있는 의미적 쌍을 생성하는 동작; 상기 취약 함수 및 상기 패치 함수 중 분석 대상으로 선택된 적어도 하나 이상의 제1 함수에 대한 모든 버전별 함수를 수집하는 동작; 및 각 버전의 함수가 상기 의미적 쌍을 포함하는 비율을 기초로 각 버전의 취약 여부를 판별

하는 동작을 포함할 수 있다.

【0012】 또한, 상기 획득하는 동작은 상기 취약점 정보를 포함하는 NVD 데이터베이스의 CVE 상세 페이지로부터 커밋 URL을 수집하여 상기 커밋 URL에 대응하는 보안 패치 정보를 획득하는 동작; 및 상기 커밋 URL에 포함된 소프트웨어 저장소 주소로부터 소프트웨어 저장소 전체를 로컬 환경으로 복제하여 상기 소스코드 정보를 획득하는 동작을 포함할 수 있다.

【0013】 또한, 상기 분류하는 동작은 상기 소프트웨어 저장소로부터 패치 적용 전 버전의 소스코드 파일과 패치 적용 후 버전의 소스코드 파일을 추출하는 동작; Universal Ctags를 이용하여 각 소스코드 파일에서 보안 패치로 인해 + 또는 - 표시가 부여된 코드 라인을 포함하는 함수를 추출하는 동작; 및 상기 추출된 함수 중 - 표시가 부여된 코드 라인을 포함하는 함수를 취약 함수로, + 표시가 부여된 코드 라인을 포함하는 함수를 패치 함수로 분류하는 동작을 포함할 수 있다.

【0014】 또한, 상기 의미적 쌍을 생성하는 동작은 취약 함수에 해당하는 필수 라인과 이에 매핑된 의존 라인을 취약 의미적 쌍으로 생성하는 동작; 및 패치 함수에 해당하는 필수 라인과 이에 매핑된 의존 라인을 패치 의미적 쌍으로 생성하는 동작을 포함할 수 있다.

【0015】 또한, 상기 의미적 쌍을 생성하는 동작은 패치에 의해 추가된 코드와 패치에 의해 삭제된 코드가 구문적으로 동일한 경우의 라인을 의미적 쌍에서 제거하는 동작; 및 상기 취약 의미적 쌍과 상기 패치 의미적 쌍 양쪽에 모두 포함되어 있는 의존 라인이 있는 경우, 취약점의 증거를 우선하도록 해당 의존 라인을 패

치 의미적 쌍에서 제거하는 동작을 포함할 수 있다.

【0016】 또한, 상기 판별하는 동작은 상기 제1 함수에 대한 제1 버전에 대하여, 상기 제1 버전이 전체 취약 의미적 쌍을 포함하는 비율로서 취약점 유사도를 계산하는 동작; 상기 제1 버전이 전체 패치 의미적 쌍을 포함하는 비율로서 패치 유사도를 계산하는 동작; 및 상기 취약점 유사도가 미리 설정된 취약 임계값보다 크고, 상기 패치 유사도가 미리 설정된 패치 임계값보다 작은 경우, 상기 제1 버전을 취약 버전으로 판별하는 동작을 포함할 수 있다.

【0017】 또한, 상기 모든 버전별 함수를 수집하는 동작은 상기 소프트웨어 저장소에 대해 "git log --follow -L:<function>:<file>" 명령을 실행하여, 상기 제1 함수의 변경 이력을 추적함으로써 상기 제1 함수에 대한 모든 버전별 함수를 수집하는 동작을 포함할 수 있다.

【0018】 또한, 상기 판별하는 동작은 각 버전의 함수에 대하여 의미적 쌍을 기준으로 비교하기 전에, 상기 제1 함수의 각 버전에 대응하는 함수의 코드 라인을 정규화하기 위해, 상기 코드 라인에 포함된 주석 및 공백을 제거하는 동작; 상기 코드 라인의 문자를 모두 소문자로 변환하는 동작; 및 상기 코드 라인에 포함된 기설정된 구문을 분석 대상에서 제외하는 동작을 포함할 수 있다.

【0019】 또한, 상기 제1 함수는 복수의 함수를 포함하고, 상기 판별하는 동작은 특정 소프트웨어 버전에 포함된 상기 복수의 함수를 대상으로 각 함수에 대한 취약 여부 판정 결과를 도출하는 동작; 상기 복수의 함수 중 적어도 하나가 취약한 것으로 판정된 경우, 상기 소프트웨어 버전을 취약 버전으로 분류하는 동작; 및 상

기 각 함수에 대한 취약 여부 판정 및 상기 취약 버전 분류 동작을 소프트웨어의 모든 버전에 대해 수행하여, 소프트웨어의 시작 버전과 종료 버전의 범위 형태로 최종적인 취약 버전 정보를 생성하는 동작을 포함할 수 있다.

【0020】 일 실시예에 따른 소프트웨어 취약 버전 식별 장치는 명령어를 포함하는 메모리; 및 상기 명령어를 기초로 소정의 동작을 수행하는 프로세서를 포함하고, 상기 프로세서의 동작은 분석 대상이 되는 취약점 정보를 입력 받아, 상기 취약점 정보에 대응하는 보안 패치 정보 및 소스코드를 획득하는 동작; 상기 보안 패치 정보를 기초로 상기 소스코드에서 보안 패치로 인해 삭제된 코드 라인을 포함하는 함수를 취약 함수로, 보안 패치로 인해 추가된 코드 라인을 포함하는 함수를 패치 함수로 분류하는 동작; 상기 취약 함수 및 상기 패치 함수 각각을 보안과 연관된 필수 라인으로 지정하고, 어느 하나의 제1 필수 라인에서 사용된 변수를 선언하는 라인, 상기 변수에 값을 할당하는 라인, 또는, 상기 제1 필수 라인의 실행 여부를 결정하는 라인을 제1 의존 라인으로 지정하는 동작; 상기 제1 필수 라인과 상기 제1 의존 라인을 매핑하여 취약점이 의미적으로 연결되어 있는 의미적 쌍을 생성하는 동작; 상기 취약 함수 및 상기 패치 함수 중 분석 대상으로 선택된 적어도 하나 이상의 제1 함수에 대한 모든 버전별 함수를 수집하는 동작; 및 각 버전의 함수가 상기 의미적 쌍을 포함하는 비율을 기초로 각 버전의 취약 여부를 판별하는 동작을 포함할 수 있다.

【발명의 효과】

【0021】 본 발명에 따르면 보안 패치의 적용 여부를 단순한 코드 차이가 아

닌 의미적 연관성을 중심으로 분석함으로써, 소프트웨어의 취약 여부를 보다 정밀하게 판정할 수 있다. 구체적으로, 취약점과 직접적으로 연결된 필수 라인과 그 실행 조건이나 데이터 흐름을 구성하는 의존 라인을 함께 고려하므로, 코드 표현이 일부 변형되거나 함수명이 변경되더라도 일관된 취약성 검출이 가능하다. 이를 통해 취약 버전의 시작 시점과 종료 시점을 명확히 특정할 수 있으며, 기존 데이터베이스에서 발생하던 과잉 보고나 누락 문제를 효과적으로 줄일 수 있다.

【0022】 또한, 본 발명은 함수 계보를 추적하여 각 소프트웨어 버전별 코드의 진화를 반영함으로써, 실제 보안 상태를 정확하게 재현할 수 있으며, 이를 기반으로 취약버전 범위를 자동화된 방식으로 신속하게 도출할 수 있다. 이러한 효과는 보안 담당자가 즉각적으로 대응 범위를 설정하고 패치 우선순위를 정하는 데 활용될 수 있어, 보안 운영의 효율성과 신뢰성을 동시에 향상시키는 결과를 제공한다.

【0023】 한편, 본 발명에 의한 효과는 이상에서 언급한 것들로 제한되지 않으며, 언급되지 않은 또 다른 기술적 효과들은 아래의 기재로부터 통상의 기술자에게 명확하게 이해될 수 있을 것이다.

【도면의 간단한 설명】

【0024】 도 1은 일 실시예에 따른 소프트웨어 취약 버전 식별 장치의 구성도이다.

도 2는 일 실시예에 따른 소프트웨어 취약 버전 식별 장치가 수행하는 동작의 단계를 나타낸 흐름도이다.

도 3은 일 실시예에 따른 소프트웨어 취약 버전 식별 장치가 도 2의 동작에 따라 처리하는 데이터의 흐름을 나타낸 개념도이다.

도 4는 일 실시예에 따른 소스코드에서 취약 함수 또는 패치 함수를 분류하는 동작의 예시도이다.

도 5는 일 실시예에 따라 필수 라인과 이에 대응하는 의존 라인을 탐색하는 동작의 예시도이다.

도 6은 기존 기법과 본 발명의 기법을 사용한 취약 버전 판별 과정에서 다양한 지표의 성능을 평가한 실험 결과에 대한 도면이다.

도 7은 NVD와 본 발명의 기법을 이용하여 취약 버전 범위 산정의 차이를 유형별로 나타낸 도면이다.

도 8은 NVD와 본 발명의 기법을 이용한 취약 버전 정보 간 차이를 정량적으로 분석한 결과를 나타낸 표이다.

도 9는 본 발명의 기법을 통해 각 CVE를 대상으로 취약 버전 범위를 분석하는 데 소요된 시간을 나타낸 그래프이다.

【발명을 실시하기 위한 구체적인 내용】

【0025】본 발명의 목적과 기술적 구성 및 그에 따른 작용 효과에 관한 자세한 사항은 본 발명의 명세서에 첨부된 도면에 의거한 이하의 상세한 설명에 의해 보다 명확하게 이해될 것이다. 첨부된 도면을 참조하여 본 발명에 따른 실시예를 상세하게 설명한다.

【0026】본 명세서에서 개시되는 실시예들은 본 발명의 범위를 한정하는 것으로 해석되거나 이용되지 않아야 할 것이다. 이 분야의 통상의 기술자에게 본 명세서의 실시예를 포함한 설명은 다양한 응용을 갖는다는 것이 당연하다. 따라서, 본 발명의 상세한 설명에 기재된 임의의 실시예들은 본 발명을 보다 잘 설명하기 위한 예시적인 것이며 본 발명의 범위가 실시예들로 한정되는 것을 의도하지 않는다.

【0027】도면에 표시되고 아래에 설명되는 기능 블록들은 가능한 구현의 예들일 뿐이다. 다른 구현들에서는 상세한 설명의 사상 및 범위를 벗어나지 않는 범위에서 다른 기능 블록들이 사용될 수 있다. 또한, 본 발명의 하나 이상의 기능 블록이 개별 블록들로 표시되지만, 본 발명의 기능 블록들 중 하나 이상은 동일 기능을 실행하는 다양한 하드웨어 및 소프트웨어 구성들의 조합일 수 있다.

【0028】또한, 어떤 구성요소들을 포함한다는 표현은 “개방형”의 표현으로서 해당 구성요소들이 존재하는 것을 단순히 지칭할 뿐이며, 추가적인 구성요소들을 배제하는 것으로 이해되어서는 안 된다.

【0029】나아가 어떤 구성요소가 다른 구성요소에 “연결되어” 있다거나 “접속되어” 있다고 언급될 때에는, 그 다른 구성요소에 직접적으로 연결 또는 접속되어 있을 수도 있지만, 중간에 다른 구성요소가 존재할 수도 있다고 이해되어야 한다.

【0030】이하, 본 발명의 다양한 실시예가 첨부된 도면을 참조하여 기재된다. 그러나, 이는 본 발명을 특정한 실시 형태에 대해 한정하려는 것이 아니며, 본 발명의 실시예의 다양한 변경(modification), 균등물(equivalent), 및/또는 대체물(alternative)을 포함하는 것으로 이해되어야 한다.

【0031】본 발명은 보안 패치 정보를 이용하여 취약점 발생에 직접적으로 관련된 코드 라인과, 이와 데이터 또는 제어 의존성에 의해 의미적으로 연결된 코드 라인을 의미적 쌍으로 매핑함으로써 소프트웨어의 취약 버전을 효과적으로 식별하는 기술을 구현하는 소프트웨어 취약 버전 식별 장치(100)를 제안한다.

【0032】이하, 본 발명의 소프트웨어 취약 버전 식별 장치(100)에 대한 구성과, 각 구성의 동작을 살펴보도록 한다.

【0033】도 1은 일 실시예에 따른 소프트웨어 취약 버전 식별 장치(100)(이하, '장치(100)'로 지칭)의 구성도이다.

【0034】도 1을 참조하면, 일 실시예에 따른 장치(100)는 각각 메모리(110), 프로세서(120), 입출력 인터페이스(130) 및 통신 인터페이스(140)를 포함할 수 있다.

【0035】메모리(110)는 외부 장치로부터 획득한 데이터 또는 스스로 생성한 데이터를 저장할 수 있다. 메모리(110)는 프로세서(120)의 동작을 수행시킬 수 있는 명령어들을 저장할 수 있다. 예를 들어, 메모리(110)는 취약점 정보, 보안 패치 정보, 소스코드, 의미적 쌍, 함수의 버전별 정보 등을 저장할 수 있다.

【0036】 프로세서(120)는 전반적인 동작을 제어하는 연산 장치이다. 프로세서(120)는 메모리(110)에 저장된 명령어들을 실행할 수 있다. 본 문서의 실시예에 따른 장치(100)의 동작은 프로세서(120)에 의해 수행되는 동작으로 이해될 수 있다.

【0037】 입출력 인터페이스(130)는 정보를 입력하거나 출력하는 하드웨어 인터페이스 또는 소프트웨어 인터페이스를 포함할 수 있다.

【0038】 통신 인터페이스(140)는 통신망을 통해 정보를 송수신 할 수 있게 한다. 이를 위해, 통신 인터페이스(140)는 무선 통신모듈 또는 유선 통신모듈을 포함할 수 있다.

【0039】 장치(100)는 프로세서(120)를 통해 연산을 수행하고 네트워크를 통해 정보를 송수신할 수 있는 다양한 형태의 장치로 구현될 수 있다. 예를 들면, 서버, 컴퓨터 장치, 휴대용 통신 장치, 스마트 폰, 휴대용 멀티미디어 장치, 노트북, 태블릿 PC 등의 형태로 구현될 수 있으나, 이러한 예시에 한정되는 것은 아니다.

【0040】 도 2는 일 실시예에 따른 장치(100)가 수행하는 동작의 단계를 나타낸 흐름도이다. 도 3은 일 실시예에 따른 장치(100)가 도 2의 동작에 따라 처리하는 데이터의 흐름을 나타낸 개념도이다. 도 2 및 도 3의 실시예에 따른 장치(100)의 동작은 프로세서(120)에 의해 수행되는 동작으로 이해될 수 있다.

【0041】 한편, 도 2 및 도 3에 개시된 각 단계는 본 발명의 목적을 달성함에 있어서 바람직한 실시예일 뿐이며, 필요에 따라 일부 단계가 추가 또는 삭제될 수

있음은 물론이고, 어느 한 단계가 다른 단계에 포함되어 수행될 수도 있다. 도 2 및 도 3에 개시된 각 동작의 순서는 이해의 편의를 위해 배치된 순서일 뿐, 이러한 순서가 시계열적인 순서로 한정되는 것이 아니며, 설계자의 선택에 따라 순서가 다르게 변경되어 동작될 수 있다.

【0042】 도 2 및 도 3을 함께 참조하면, S1010 단계에서, 장치(100)는 분석 대상이 되는 취약점 정보를 입력 받아, 취약점 정보에 대응하는 보안 패치 정보 및 소스코드를 획득할 수 있다.

【0043】 여기서 취약점 정보란 소프트웨어의 보안 결함을 특정하기 위해 부여되는 고유 식별자로, 예컨대 CVE(Common Vulnerabilities and Exposures) ID와 같은 형태일 수 있다.

【0044】 보안 패치 정보란 취약점 정보를 해결하기 위하여 개발자 또는 공급자가 공개한 코드 변경에 관한 정보를 말하며, 예컨대 패치에 대응하는 커밋 식별자(예: commit hash), 커밋이 게시된 URL(예: GitHub 커밋 URL), 패치에 포함된 변경 파일 목록 및 각 파일에서 추가(+)되거나 삭제(-)된 코드 라인을 나타내는 diff 정보 등을 포함할 수 있다.

【0045】 일 예로, 장치(100)는 취약점 정보를 이용하여 NVD(National Vulnerability Database)와 같은 취약점 데이터베이스의 CVE(Common Vulnerabilities and Exposures) 상세 페이지를 조회하고, CVE 상세 페이지에 기재된 커밋 URL을 수집함으로써 보안 패치 정보를 확보할 수 있다. 또한, 장치(100)는 커밋에 기재된 소프트웨어 저장소 주소를 이용하여 해당 저장소를 로컬 환경으로

복제(예: git clone)하여 소스코드를 획득할 수 있다.

【0046】 소스코드란 보안 패치 정보에 대응하는 소프트웨어 저장소에 저장된 코드 일체를 의미하며, 예컨대 저장소의 전체 복제본(예: git clone으로 획득한 작업복사본), 소스 파일들(예: .c, .cpp, .java, .py 등), 각 소스 파일의 특정 시점 버전(예: 커밋 해시로 식별되는 스냅샷) 및 버전 관리 히스토리(예: 커밋 로그, 브랜치 정보)를 포함한다.

【0047】 일 예로, 장치(100)는 특정 CVE ID(CVE-2025-XXXX 등)를 입력받은 경우, 해당 CVE ID를 키워드로 NVD 데이터베이스를 검색하고, 검색 결과에 포함된 GitHub 커밋 URL을 추출한 뒤, 이를 통해 해당 소프트웨어 저장소 전체를 로컬 환경에 복제하여 분석 대상 소스코드를 확보할 수 있다. 본 발명에서는 소스코드를 이용하여 패치 적용 전, 후의 함수 및 변경된 코드 라인을 추출하고, 해당 함수의 변경 이력을 따라 모든 버전의 함수를 수집하여 이후의 의미적 쌍 생성 및 취약 여부 판정에 활용하게 된다.

【0048】 S1020 단계에서, 장치(100)는 보안 패치 정보를 기초로 소스코드에서 보안 패치로 인해 삭제된 코드 라인을 포함하는 함수를 취약 함수로, 보안 패치로 인해 추가된 코드 라인을 포함하는 함수를 패치 함수로 분류할 수 있다.

【0049】 도 4는 일 실시예에 따른 소스코드에서 취약 함수 또는 패치 함수를 분류하는 동작의 예시도이다.

【0050】도 4를 참조하면, 장치(100)는 보안 패치가 적용되기 전, 후의 소스 코드 파일을 비교하여, 삭제(-) 표시가 부여된 코드 라인을 포함하는 함수와 추가(+) 표시가 부여된 코드 라인을 포함하는 함수를 각각 추출할 수 있다. 예컨대 도 4의 예시에서, nstrips64 = TIFFhowmany_64(...)와 같이 - 표시가 붙어 삭제된 코드 라인을 포함하는 함수는 취약 함수로 분류되며, nstrips = TIFFhowmany_32(...) 및 if (nstrips == 0)와 같이 + 표시가 붙어 추가된 코드 라인을 포함하는 함수는 패치 함수로 분류된다.

【0051】이를 위해, 장치(100)는 소프트웨어 저장소로부터 패치 적용 전 버전의 소스코드 파일과 패치 적용 후 버전의 소스코드 파일을 추출하고, Universal Ctags와 같은 파싱 도구를 이용하여 각 파일에서 + 또는 - 표시가 부여된 코드 라인을 포함하는 함수를 추출함으로써, 추출된 함수 중 - 표시가 부여된 함수를 취약 함수로, + 표시가 부여된 함수를 패치 함수로 분류할 수 있다.

【0052】S1030 단계에서, 장치(100)는 취약 함수 및 패치 함수 각각을 보안과 연관된 필수 라인으로 지정하고, 어느 하나의 제1 필수 라인에서 사용된 변수를 선언하는 라인, 어느 하나의 제1 필수 라인에서 사용된 변수에 특정 값을 할당하는 라인, 또는, 제1 필수 라인의 실행 여부를 결정하는 라인 중 어느 하나에 속하는 라인을 제1 의존 라인으로 지정할 수 있다.

【0053】즉, 제1 필수 라인에서 참조되는 변수가 함수 내에서 처음 정의되는 구문이 존재하는 경우, 해당 변수 선언 라인은 제1 필수 라인의 정상적인 실행을 위해 반드시 필요하므로 제1 의존 라인으로 지정될 수 있다.

【0054】또한 제1 필수 라인에서 사용되는 변수에 대하여 특정한 값이 대입되는 구문이 존재하는 경우, 해당 대입 구문은 제1 필수 라인의 동작 결과에 직접적인 영향을 주므로 제1 의존 라인으로 지정될 수 있다.

【0055】또한, 제1 필수 라인의 실행이 조건문(if, while 등)이나 분기문에 의해 제어되는 경우, 해당 조건문 또는 분기문 라인은 제1 필수 라인의 실행 여부를 결정하는 중요한 맥락 정보를 제공하므로 제1 의존 라인으로 지정될 수 있다.

【0056】여기서, '제1'이라는 용어는 복수의 필수 라인 중 어느 하나를 특정하기 위해 사용된 식별 기호에 불과하며, 순서의 의미를 한정하는 것은 아니다. 따라서 제1 필수 라인 및 제1 의존 라인이라는 용어는 모든 필수 라인 중 어느 하나를 예시적으로 지칭하기 위한 편의상의 표기에 불과하다.

【0057】이를 위해, 장치(100)는 Joern과 같은 코드 분석 도구를 이용하여 C/C++ 소스코드를 분석하고, 코드 속성 그래프(Code Property Graph, CPG)를 생성할 수 있다. 코드 속성 그래프는 데이터 흐름, 제어 흐름, 추상 구문 트리 정보를 통합적으로 표현하는 그래프 구조로서, 장치(100)는 이를 통해 코드 라인 간의 데이터 의존성 및 제어 의존성을 식별할 수 있다. 이러한 의존성 분석 결과는 필수 라인과 의존 라인 간의 연결 관계를 체계적으로 파악하는 데 활용되며, 결과적으로 의미적 쌍을 보다 정확하게 구성할 수 있도록 지원한다.

【0058】도 5는 일 실시예에 따라 필수 라인과 이에 대응하는 의존 라인을 탐색하는 동작의 예시도이다.

【0059】도 5를 참조하면, 코드 내에서 붉은색으로 표시된 부분은 보안과 직접적으로 관련된 필수 라인(예: 취약 함수 내 취약 코드 라인)을 나타내며, 청색으로 표시된 부분은 해당 필수 라인과 데이터 또는 제어 의존 관계를 가지는 의존 라인이다. 예컨대, 변수 stripbytes를 선언하는 라인과 rowblockbytes 값을 할당하는 라인은 필수 라인에서 사용된 변수를 정의하거나 갱신하므로 데이터 의존 라인으로 탐색된다. 또한, 조건문에서 rowsperstrip을 비교하는 라인은 필수 라인의 실행 여부를 제어하는 구문이므로 제어 의존 라인으로 탐색된다.

【0060】따라서 장치(100)는 특정 필수 라인을 기준으로, 데이터 흐름과 제어 흐름을 분석하여 해당 필수 라인과 의미적으로 연결된 모든 의존 라인을 식별할 수 있고, 다음의 동작을 통해 취약점과 직접 또는 간접적으로 관련된 의미적 쌍을 구성할 수 있다.

【0061】S1040 단계에서, 장치(100)는 제1 필수 라인과 제1 필수 라인에 대응하는 제1 의존 라인을 매핑하여 취약점이 의미적으로 연결되어 있는 의미적 쌍을 생성할 수 있다. 즉, 장치(100)는 특정 필수 라인 하나에 대해서만 의미적 쌍을 생성하는 것이 아니라, 모든 취약 함수 및 패치 함수 각각의 필수 라인을 대상으로, 각각의 필수 라인과 그에 대응하는 의존 라인을 탐색하고, 이를 매핑하여 다수의 의미적 쌍을 구성한다.

【0062】여기서 의미적 쌍이란, 단순히 코드 라인의 물리적 인접성을 기준으로 한 관계가 아니라, 변수 참조, 값 할당, 제어 흐름 분기와 같이 프로그램 실행 논리 상 서로 영향을 주고받는 관계를 반영한 쌍을 말한다. 예컨대, 필수 라인에서

사용되는 변수가 특정 의존 라인에서 선언되거나 초기화되는 경우, 또는 필수 라인의 실행 여부가 조건문 의존 라인에 의해 제어되는 경우, 두 라인은 의미적으로 연결된 것으로 간주되어 하나의 의미적 쌍을 형성한다. 따라서 이러한 의미적 쌍은 보안 취약점이 코드 실행 경로와 데이터 흐름을 통해 어떻게 전파되는지를 설명할 수 있는 단위가 된다.

【0063】 이때, 장치(100)는 취약 함수에 해당하는 필수 라인과 이에 매핑된 의존 라인을 취약 의미적 쌍으로 생성하고, 패치 함수에 해당하는 필수 라인과 이에 매핑된 의존 라인을 패치 의미적 쌍으로 생성하여, 의미적 쌍을 두 가지 범주로 구분할 수 있다. 이를 통해 장치(100)는 동일한 코드 구조 내에서도 보안 취약점의 발생 원인과 이를 보완하기 위한 수정 사항을 명확히 구별할 수 있으며, 이후 단계에서 각 함수 버전이 취약 의미적 쌍 또는 패치 의미적 쌍을 어느 정도 포함하는지 비교함으로써 해당 버전의 취약 여부를 보다 정밀하게 판별할 수 있다.

【0064】 추가적으로, 장치(100)는 의미적 쌍의 신뢰성과 정확도를 높이기 위해, 의미적 쌍을 생성하는 과정에서 불필요하거나 중복되는 관계를 정제할 수 있다.

【0065】 예를 들어, 장치(100)는 패치에 의해 추가된 코드와 패치에 의해 삭제된 코드가 구문적으로 동일한 경우의 라인이 있다면, 해당 라인을 의미적 쌍에서 제거할 수 있다. 이로써 단순한 코드 위치 변경이나 표현 차이로 인해 발생하는 불필요한 중복 쌍이 제거될 수 있다.

【0066】 또한, 취약 의미적 쌍과 패치 의미적 쌍 양쪽에 동일한 의존 라인이

포함되어 있는 경우, 장치(100)는 해당 의존 라인을 취약 의미적 쌍에만 남기고, 패치 의미적 쌍에서는 제거하여 취약점의 증거를 우선하도록 할 수 있다. 이를 통해 취약 여부를 판별하는 과정에서 모호성을 줄이고, 취약 코드의 근거성을 강화할 수 있다.

【0067】 S1050 단계에서, 장치(100)는 취약 함수 및 패치 함수 중 분석 대상으로 선택된 적어도 하나 이상의 제1 함수에 대한 모든 버전별 함수를 수집할 수 있다.

【0068】 여기서, 제1 함수란 보안 패치로 인해 변경된 함수들 중에서 사용자가 관심을 가지고 분석 대상으로 지정한 함수로서, 취약 함수 또는 패치 함수 중 어느 하나일 수 있다. 즉, 제1 함수는 사용자의 분석 목적에 따라 선택된 핵심 함수로, 해당 함수의 모든 버전을 수집함으로써 취약점이 발생하고 수정되는 과정을 추적하고 이후 의미적 쌍 생성 및 취약 여부 판정에 활용할 수 있다.

【0069】 일 예로, 장치(100)는 소프트웨어 저장소에 대해 "git log --follow -L:<function>:<file>" 명령을 실행하여, 제1 함수의 변경 이력을 추적함으로써 제1 함수에 대한 모든 버전별 함수를 수집할 수 있다. 이때, 제1 함수의 변경 이력은 함수 단위의 코드 추가, 삭제, 수정 내역을 포함할 수 있으며, 장치(100)는 해당 이력을 기반으로 소스코드 저장소의 각 커밋 시점에서 존재하는 제1 함수의 스냅샷을 획득할 수 있다. 따라서 장치(100)는 제1 함수가 초기 버전에서 어떠한 코드 라인을 포함하고 있었는지, 이후 버전에서 어떠한 방식으로 변경되었는지를 추적할 수 있으며, 이러한 일련의 버전별 함수들을 수집함으로써 특정 취약점의 발생 시점

과 패치 적용 시점을 연속적으로 분석할 수 있게 도힌다.

【0070】 S1060 단계에서, 장치(100)는 제1 함수에 대한 각 버전의 함수가 의미적 쌍을 포함하는 비율을 기초로 각 버전의 취약 여부를 판별할 수 있다.

【0071】 이때, 장치(100)는 의미적 쌍 기반의 비교가 코드 스타일이나 불필요한 기호 차이에 의해 왜곡되지 않도록, 의미적 쌍을 기준으로 비교하기 이전에, 각 버전에 대응하는 함수의 코드 라인을 정규화할 수 있다. 구체적으로, 장치(100)는 제1 함수의 각 버전에 대응하는 함수의 코드 라인에 포함된 주석 및 공백을 제거하여 불필요한 비의미적 요소를 배제하고, 코드 라인의 문자를 모두 소문자로 변환하여 대소문자 차이에 따른 불일치를 해소하며, 사전에 설정된 특정 구문(예: import문, 단순 로그 출력문 등)을 분석 대상에서 제외함으로써 본질적인 코드 동작에 관련된 부분만을 비교 대상으로 설정할 수 있다.

【0072】 이에 따라, 장치(100)는 제1 함수에 대한 각 버전의 함수가 취약 버전인지 여부를 정량적으로 판별하기 위해, 특정 버전인 제1 버전의 함수가 의미적 쌍을 포함하는 비율을 기초로 제1 버전의 취약 여부를 판별할 수 있다. 여기서, '제1'이라는 용어는 복수의 버전 중 어느 하나를 특정하기 위해 사용된 식별 기호에 불과하며, 순서의 의미를 한정하는 것은 아니다. 따라서 제1 버전이라는 용어는 모든 버전 중 어느 하나를 예시적으로 지칭하기 위한 편의상의 표기에 불과하다.

【0073】 구체적으로, 장치(100)는 제1 버전이 전체 취약 의미적 쌍을 포함하는 비율로서 취약점 유사도를 계산할 수 있다. 즉, 장치(100)는 제1 버전에 존재하는 취약 의미적 쌍의 개수를 전체 취약 의미적 쌍의 개수로 나누어, 해당 버전이

보안 패치 이전의 취약 동작을 얼마나 많이 보존하고 있는지를 정량적으로 산출한다. 이를 통해 취약점 유사도는 값이 1에 가까울수록 패치가 적용되지 않은 상태와 유사함을 의미하고, 값이 0에 가까울수록 패치로 인한 취약점 제거 효과가 반영된 상태임을 나타낸다.

【0074】 또한, 장치(100)는 제1 버전이 전체 패치 의미적 쌍을 포함하는 비율로서 패치 유사도를 계산할 수 있다. 즉, 장치(100)는 제1 버전에 존재하는 패치 의미적 쌍의 개수를 전체 패치 의미적 쌍의 개수로 나누어, 해당 버전이 보안 패치 이후의 수정 동작을 얼마나 반영하고 있는지를 정량적으로 산출한다. 이에 따라 패치 유사도는 값이 1에 가까울수록 패치가 충실히 적용된 상태임을 의미하고, 값이 0에 가까울수록 패치 적용의 흔적이 거의 없는 상태임을 나타낸다.

【0075】 이에 따라, 장치(100)는 취약점 유사도가 미리 설정된 취약 임계값(예: 0.5)보다 크고, 패치 유사도가 미리 설정된 패치 임계값(예: 0.5)보다 작은 경우, 제1 버전을 취약 버전으로 판별할 수 있다. 예를 들어, 취약 임계값과 패치 임계값이 각각 0.5로 설정된 경우, 특정 제1 버전에 대해 취약점 유사도가 0.7로 계산되고 패치 유사도가 0.3으로 계산되었다면, 해당 제1 버전은 취약점 관련 코드 패턴을 다수 포함하면서도 보안 패치의 효과는 충분히 반영하지 못한 상태로 판단되므로, 장치(100)는 이를 취약 버전으로 분류할 수 있다.

【0076】 이와 같은 판별 과정은 제1 함수의 모든 버전에 반복 적용되어, 각 버전에 대한 취약 여부가 체계적으로 도출되며, 이를 통해 최종적으로 소프트웨어의 취약 버전 범위를 확정할 수 있다.

【0077】 한편, 분석 대상이 되는 제1 함수는 하나만 지정되는 것이 아니라, 사용자의 관심이나 분석 목적에 따라 복수 개가 동시에 지정될 수도 있다.

【0078】 이에 따라, 장치(100)는 특정 소프트웨어 버전에 포함된 복수의 제1 함수를 대상으로 각각의 함수에 대해 취약 여부 판정 결과를 도출할 수 있다. 이때 복수의 함수 중 적어도 하나라도 취약한 것으로 판정된 경우, 장치(100)는 해당 소프트웨어 버전을 전체적으로 취약 버전으로 분류할 수 있다. 또한, 이러한 판정 절차는 소프트웨어의 모든 버전에 반복 적용되어, 각 버전별로 취약 여부가 체계적으로 도출될 수 있다. 장치(100)는 이 결과를 종합하여 소프트웨어의 시작 버전과 종료 버전을 포함하는 범위 형태로 최종적인 취약 버전 정보를 생성할 수 있다.

【0079】 도 6은 기존 기법과 본 발명의 기법을 사용한 취약 버전 판별 과정에서 다양한 지표의 성능을 평가한 실험 결과에 대한 도면이다.

【0080】 도 6을 참조하면, 본 발명의 기법(CLOVERY)은 기존의 기법(NVD, VOFinder, V-SZZ) 대비 현저히 높은 정확도를 보였으며, Macro-F1 점수 기준으로 97.43%를 달성하였다. 이때 Precision은 96.27%, Recall은 98.62%로 측정되었으며, 이는 보안 취약점이 반영된 버전을 식별하는 데 있어 높은 정밀도와 재현율을 동시에 확보하였음을 확인할 수 있다. 따라서 도 6은 본 발명의 기법이 기존 기법들에 비해 취약 버전 판별의 정확성과 신뢰성을 크게 향상시킨다는 점을 실험적으로 나타낸다.

【0081】도 7은 NVD와 본 발명의 기법을 이용하여 취약 버전 범위 산정의 차이를 유형별로 나타낸 도면이다.

【0082】도 7을 참조하면, 본 발명의 기법(CLOVERY)은 취약 버전의 시작점과 종료점을 의미적 쌍 기반으로 정밀하게 판별함으로써 NVD와 차이를 보인다. T1과 T2는 취약 버전의 시작점 차이를 나타내며, T1의 경우 NVD가 CLOVERY보다 이른 버전을 시작점으로 지정하였고, T2의 경우 CLOVERY가 NVD보다 이른 시작 버전을 판별하였다. T3와 T4는 취약 버전의 종료점 차이를 보여주며, T3의 경우 CLOVERY가 더 빠른 종료점을, T4의 경우 NVD가 더 빠른 종료점을 제시하였다. 마지막으로 T5는 취약 버전의 중간 구간에서 차이가 발생하는 경우를 의미하며, NVD는 연속된 취약 버전 범위를 산출한 반면, 본 발명은 의미적 쌍 단위 분석을 통해 실제 취약 코드가 포함된 특정 버전만을 식별함으로써 불필요하게 확장된 범위를 줄일 수 있음을 보여준다. 따라서 도 7은 본 발명의 방법이 단순한 구문적 비교가 아닌 의미적 분석을 통해 취약 버전의 시작점, 종료점, 중간 구간을 보다 정밀하게 특정할 수 있음을 나타낸다.

【0083】도 8은 NVD와 본 발명의 기법을 이용한 취약 버전 정보 간 차이를 정량적으로 분석한 결과를 나타낸 표이다.

【0084】도 8을 참조하면, 두 방법의 취약 버전 범위 산정 과정에서 발생하는 차이는 T1~T5의 다섯 가지 유형으로 구분될 수 있다. 도 8의 결과는 본 발명의 기법(CLOVERY)이 NVD에 비해 취약 버전의 시작점과 종료점뿐 아니라 중간 구간에 이르기까지 다양한 차이를 보이면서도, 실제 코드 기반 의미적 분석을 통해 보다

정밀하고 근거 있는 취약 버전 범위를 산출함을 나타낸다.

【0085】 도 9는 본 발명의 기법을 통해 각 CVE를 대상으로 취약 버전 범위를 분석하는 데 소요된 시간을 나타낸 그래프이다.

【0086】 도 9를 참조하면, 초기 일부 CVE의 경우 보안 패치 내역이 복잡하거나 코드 변경 이력이 방대하여 수천 초(예: 약 6000초 이상)가 소요되는 사례도 존재한다. 그러나 대부분의 CVE에서는 시간이 급격히 감소하여 수백 초 이하의 범위에서 분석이 완료되며, 이후 약 2000개 이상의 CVE에 대해서는 안정적으로 짧은 시간 내에 처리되는 것을 확인할 수 있다. 따라서 도 9는 본 발명이 복잡한 코드 변경 이력에도 불구하고 대규모 CVE 집합을 효율적으로 분석할 수 있음을 나타낸다.

【0087】 상술한 실시예에 따르면, 본 발명은 보안 패치의 적용 여부를 단순한 코드 차이가 아닌 의미적 연관성을 중심으로 분석함으로써, 소프트웨어의 취약 여부를 보다 정밀하게 판정할 수 있다. 구체적으로, 취약점과 직접적으로 연결된 필수 라인과 그 실행 조건이나 데이터 흐름을 구성하는 의존 라인을 함께 고려하므로, 코드 표현이 일부 변형되거나 함수명이 변경되더라도 일관된 취약성 검출이 가능하다. 이를 통해 취약 버전의 시작 시점과 종료 시점을 명확히 특정할 수 있으며, 기존 데이터베이스에서 발생하던 과잉 보고나 누락 문제를 효과적으로 줄일 수 있다.

【0088】 또한, 본 발명은 함수 계보를 추적하여 각 소프트웨어 버전별 코드의 진화를 반영함으로써, 실제 보안 상태를 정확하게 재현할 수 있으며, 이를 기반으로 취약버전 범위를 자동화된 방식으로 신속하게 도출할 수 있다. 이러한 효과는

보안 담당자가 즉각적으로 대응 범위를 설정하고 패치 우선순위를 정하는 데 활용될 수 있어, 보안 운영의 효율성과 신뢰성을 동시에 향상시키는 결과를 제공한다.

【0089】 본 문서의 다양한 실시예들 및 이에 사용된 용어들은 본 문서에 기재된 기술적 특징들을 특정한 실시예들로 한정하려는 것이 아니며, 해당 실시예의 다양한 변경, 균등물, 또는 대체물을 포함하는 것으로 이해되어야 한다. 도면의 설명과 관련하여, 유사한 또는 관련된 구성요소에 대해서는 유사한 참조 부호가 사용될 수 있다. 아이템에 대응하는 명사의 단수 형은 관련된 문맥상 명백하게 다르게 지시하지 않는 한, 아이템 한 개 또는 복수 개를 포함할 수 있다.

【0090】 본 문서에서, "A 또는 B", "A 및 B 중 적어도 하나", "A 또는 B 중 적어도 하나,""A, B 또는 C," "A, B 및 C 중 적어도 하나,"및 "A, B, 또는 C 중 적어도 하나"와 같은 문구들 각각은 그 문구들 중 해당하는 문구에 함께 나열된 항목들의 모든 가능한 조합을 포함할 수 있다. " 1", "제2", 또는 "첫째" 또는 "둘째"와 같은 용어들은 단순히 해당 구성요소를 다른 해당 구성요소와 구분하기 위해 사용될 수 있으며, 해당 구성요소들을 다른 측면(예: 중요성 또는 순서)에서 한정하지 않는다. 어떤(예: 제1) 구성요소가 다른(예: 제2) 구성요소에, "기능적으로" 또는 "통신적으로"라는 용어와 함께 또는 이런 용어 없이, "커플드" 또는 "커넥티드"라고 언급된 경우, 그것은 어떤 구성요소가 다른 구성요소에 직접적으로(예: 유선으로), 무선으로, 또는 제3 구성요소를 통하여 연결될 수 있다는 것을 의미한다.

【0091】 본 문서에서 사용된 용어 "모듈"은 하드웨어, 소프트웨어 또는 펌웨어로 구현된 유닛을 포함할 수 있으며, 예를 들면, 로직, 논리 블록, 부품, 또는

회로 등의 용어와 상호 호환적으로 사용될 수 있다. 모듈은, 일체로 구성된 부품 또는 하나 또는 그 이상의 기능을 수행하는, 부품의 최소 단위 또는 그 일부가 될 수 있다. 예를 들면, 일 실시예에 따르면, 모듈은 ASIC(application-specific integrated circuit)의 형태로 구현될 수 있다.

【0092】본 문서의 다양한 실시예들은 기기(예: 전자 장치)에 의해 읽을 수 있는 저장 매체(예: 메모리)에 저장된 하나 이상의 명령어들을 포함하는 소프트웨어(예: 프로그램)로서 구현될 수 있다. 저장 매체는 RAM(random access memory), 메모리 버퍼, 하드 드라이브, 데이터베이스, EPROM(erasable programmable read-only memory), EEPROM(electrically erasable read-only memory), ROM(read-only memory) 및/또는 등등을 포함할 수 있다.

【0093】또한, 본 문서의 실시예들의 프로세서는, 저장 매체로부터 저장된 하나 이상의 명령어들 중 적어도 하나의 명령을 호출하고, 그것을 실행할 수 있다. 이것은 기기가 호출된 적어도 하나의 명령어에 따라 적어도 하나의 기능을 수행하도록 운영되는 것을 가능하게 한다. 이러한 하나 이상의 명령어들은 컴파일러에 의해 생성된 코드 또는 인터프리터에 의해 실행될 수 있는 코드를 포함할 수 있다. 프로세서는 범용 프로세서, FPGA(Field Programmable Gate Array), ASIC(Application Specific Integrated Circuit), DSP(Digital Signal Processor) 및/또는 등등 일 수 있다.

【0094】기기로 읽을 수 있는 저장매체는, 비일시적(non-transitory) 저장매체의 형태로 제공될 수 있다. 여기서, '비일시적'은 저장매체가 실재(tangible)하

는 장치이고, 신호(예: 전자기파)를 포함하지 않는다는 것을 의미할 뿐이며, 이 용어는 데이터가 저장매체에 반영구적으로 저장되는 경우와 임시적으로 저장되는 경우를 구분하지 않는다.

【0095】본 문서에 개시된 다양한 실시예들에 따른 방법은 컴퓨터 프로그램 제품(computer program product)에 포함되어 제공될 수 있다. 컴퓨터 프로그램 제품은 상품으로서 판매자 및 구매자 간에 거래될 수 있다. 컴퓨터 프로그램 제품은 기기로 읽을 수 있는 저장 매체(예: compact disc read only memory (CD-ROM))의 형태로 배포되거나, 또는 어플리케이션 스토어(예: 플레이 스토어)를 통해 또는 두 개의 사용자 장치들(예: 스마트폰들) 간에 직접, 온라인으로 배포(예: 다운로드 또는 업로드)될 수 있다. 온라인 배포의 경우에, 컴퓨터 프로그램 제품의 적어도 일부는 제조사의 서버, 어플리케이션 스토어의 서버, 또는 서버의 메모리와 같은 기기로 읽을 수 있는 저장 매체에 적어도 일시 저장되거나, 임시적으로 생성될 수 있다.

【0096】다양한 실시예들에 따르면, 기술한 구성요소들의 각각의 구성요소(예: 모듈 또는 프로그램)는 단수 또는 복수의 개체를 포함할 수 있다. 다양한 실시예들에 따르면, 전술한 해당 구성요소들 중 하나 이상의 구성요소들 또는 동작들이 생략되거나, 또는 하나 이상의 다른 구성요소들 또는 동작들이 추가될 수 있다. 대체적으로 또는 추가적으로, 복수의 구성요소들(예: 모듈 또는 프로그램)은 하나의 구성요소로 통합될 수 있다. 이런 경우, 통합된 구성요소는 복수의 구성요소들 각각의 구성요소의 하나 이상의 기능들을 통합 이전에 복수의 구성요소들 중 해당

구성요소에 의해 수행되는 것과 동일 또는 유사하게 수행할 수 있다. 다양한 실시예들에 따르면, 모듈, 프로그램 또는 다른 구성요소에 의해 수행되는 동작들은 순차적으로, 병렬적으로, 반복적으로, 또는 휴리스틱하게 실행되거나, 동작들 중 하나 이상이 다른 순서로 실행되거나, 생략되거나, 또는 하나 이상의 다른 동작들이 추가될 수 있다.

【부호의 설명】

【0097】100: 장치

110: 메모리

120: 프로세서

130: 입출력 인터페이스

140: 통신 인터페이스

【청구범위】

【청구항 1】

프로세서에 의해 동작하는 소프트웨어 취약 버전 식별 장치가 수행하는 방법에 있어서,

분석 대상이 되는 취약점 정보를 입력 받아, 상기 취약점 정보에 대응하는 보안 패치 정보 및 소스코드를 획득하는 동작;

상기 보안 패치 정보를 기초로 상기 소스코드에서 보안 패치로 인해 삭제된 코드 라인을 포함하는 함수를 취약 함수로, 보안 패치로 인해 추가된 코드 라인을 포함하는 함수를 패치 함수로 분류하는 동작;

상기 취약 함수 및 상기 패치 함수 각각을 보안과 연관된 필수 라인으로 지정하고, 어느 하나의 제1 필수 라인에서 사용된 변수를 선언하는 라인, 상기 변수에 값을 할당하는 라인, 또는, 상기 제1 필수 라인의 실행 여부를 결정하는 라인을 제1 의존 라인으로 지정하는 동작;

상기 제1 필수 라인과 상기 제1 의존 라인을 매핑하여 취약점이 의미적으로 연결되어 있는 의미적 쌍을 생성하는 동작;

상기 취약 함수 및 상기 패치 함수 중 분석 대상으로 선택된 적어도 하나 이상의 제1 함수에 대한 모든 버전별 함수를 수집하는 동작; 및

각 버전의 함수가 상기 의미적 쌍을 포함하는 비율을 기초로 각 버전의 취약 여부를 판별하는 동작을 포함하는,

방법.

【청구항 2】

제1항에 있어서,

상기 획득하는 동작은

상기 취약점 정보를 포함하는 NVD 데이터베이스의 CVE 상세 페이지로부터 커밋 URL을 수집하여 상기 커밋 URL에 대응하는 보안 패치 정보를 획득하는 동작; 및

상기 커밋 URL에 포함된 소프트웨어 저장소 주소로부터 소프트웨어 저장소 전체를 로컬 환경으로 복제하여 상기 소스코드 정보를 획득하는 동작을 포함하는,

방법.

【청구항 3】

제2항에 있어서,

상기 분류하는 동작은

상기 소프트웨어 저장소로부터 패치 적용 전 버전의 소스코드 파일과 패치 적용 후 버전의 소스코드 파일을 추출하는 동작;

Universal Ctags를 이용하여 각 소스코드 파일에서 보안 패치로 인해 + 또는 - 표시가 부여된 코드 라인을 포함하는 함수를 추출하는 동작; 및

상기 추출된 함수 중 - 표시가 부여된 코드 라인을 포함하는 함수를 취약 함수로, + 표시가 부여된 코드 라인을 포함하는 함수를 패치 함수로 분류하는 동작을 포함하는,

방법.

【청구항 4】

제1항에 있어서,

상기 의미적 쌍을 생성하는 동작은

취약 함수에 해당하는 필수 라인과 이에 매핑된 의존 라인을 취약 의미적 쌍으로 생성하는 동작; 및

패치 함수에 해당하는 필수 라인과 이에 매핑된 의존 라인을 패치 의미적 쌍으로 생성하는 동작을 포함하는,

방법.

【청구항 5】

제4항에 있어서,

상기 의미적 쌍을 생성하는 동작은

패치에 의해 추가된 코드와 패치에 의해 삭제된 코드가 구문적으로 동일한 경우의 라인을 의미적 쌍에서 제거하는 동작; 및

상기 취약 의미적 쌍과 상기 패치 의미적 쌍 양쪽에 모두 포함되어 있는 의존 라인이 있는 경우, 취약점의 증거를 우선하도록 해당 의존 라인을 패치 의미적 쌍에서 제거하는 동작을 포함하는,

방법.

【청구항 6】

제4항에 있어서,

상기 판별하는 동작은

상기 제1 함수에 대한 제1 버전에 대하여,

상기 제1 버전이 전체 취약 의미적 쌍을 포함하는 비율로서 취약점 유사도를 계산하는 동작;

상기 제1 버전이 전체 패치 의미적 쌍을 포함하는 비율로서 패치 유사도를 계산하는 동작; 및

상기 취약점 유사도가 미리 설정된 취약 임계값보다 크고, 상기 패치 유사도가 미리 설정된 패치 임계값보다 작은 경우, 상기 제1 버전을 취약 버전으로 판별하는 동작을 포함하는,

방법.

【청구항 7】

제1항에 있어서,

상기 모든 버전별 함수를 수집하는 동작은

상기 소프트웨어 저장소에 대해 "git log --follow -L:<function>:<file>" 명령을 실행하여, 상기 제1 함수의 변경 이력을 추적함으로써 상기 제1 함수에 대한 모든 버전별 함수를 수집하는 동작을 포함하는,

방법.

【청구항 8】

제1항에 있어서,

상기 판별하는 동작은

각 버전의 함수에 대하여 의미적 쌍을 기준으로 비교하기 전에, 상기 제1 함수의 각 버전에 대응하는 함수의 코드 라인을 정규화하기 위해,

상기 코드 라인에 포함된 주석 및 공백을 제거하는 동작;

상기 코드 라인의 문자를 모두 소문자로 변환하는 동작; 및

상기 코드 라인에 포함된 기 설정된 구문을 분석 대상에서 제외하는 동작을 포함하는,

방법.

【청구항 9】

제1항에 있어서,

상기 제1 함수는

복수의 함수를 포함하고,

상기 판별하는 동작은

특정 소프트웨어 버전에 포함된 상기 복수의 함수를 대상으로 각 함수에 대한 취약 여부 판정 결과를 도출하는 동작;

상기 복수의 함수 중 적어도 하나가 취약한 것으로 판정된 경우, 상기 소프트웨어 버전을 취약 버전으로 분류하는 동작; 및

상기 각 함수에 대한 취약 여부 판정 및 상기 취약 버전 분류 동작을 소프트웨어의 모든 버전에 대해 수행하여, 소프트웨어의 시작 버전과 종료 버전의 범위 형태로 최종적인 취약 버전 정보를 생성하는 동작을 포함하는,

방법.

【청구항 10】

명령어를 포함하는 메모리; 및

상기 명령어를 기초로 소정의 동작을 수행하는 프로세서를 포함하고,

상기 프로세서의 동작은

분석 대상이 되는 취약점 정보를 입력 받아, 상기 취약점 정보에 대응하는 보안 패치 정보 및 소스코드를 획득하는 동작;

상기 보안 패치 정보를 기초로 상기 소스코드에서 보안 패치로 인해 삭제된 코드 라인을 포함하는 함수를 취약 함수로, 보안 패치로 인해 추가된 코드 라인을 포함하는 함수를 패치 함수로 분류하는 동작;

상기 취약 함수 및 상기 패치 함수 각각을 보안과 연관된 필수 라인으로 지정하고, 어느 하나의 제1 필수 라인에서 사용된 변수를 선언하는 라인, 상기 변수에 값을 할당하는 라인, 또는, 상기 제1 필수 라인의 실행 여부를 결정하는 라인을 제1 의존 라인으로 지정하는 동작;

상기 제1 필수 라인과 상기 제1 의존 라인을 매핑하여 취약점이 의미적으로 연결되어 있는 의미적 쌍을 생성하는 동작;

상기 취약 함수 및 상기 패치 함수 중 분석 대상으로 선택된 적어도 하나 이상의 제1 함수에 대한 모든 버전별 함수를 수집하는 동작; 및

각 버전의 함수가 상기 의미적 쌍을 포함하는 비율을 기초로 각 버전의 취약 여부를 판별하는 동작을 포함하는,

소프트웨어 취약 버전 식별 장치.

【요약서】

【요약】

일 실시예에 따른 소프트웨어 취약 버전 식별 장치는 분석 대상이 되는 취약점 정보를 입력 받아, 상기 취약점 정보에 대응하는 보안 패치 정보 및 소스코드를 획득하는 동작; 상기 보안 패치 정보를 기초로 상기 소스코드에서 보안 패치로 인해 삭제된 코드 라인을 포함하는 함수를 취약 함수로, 보안 패치로 인해 추가된 코드 라인을 포함하는 함수를 패치 함수로 분류하는 동작; 상기 취약 함수 및 상기 패치 함수 각각을 보안과 연관된 필수 라인으로 지정하고, 어느 하나의 제1 필수 라인에서 사용된 변수를 선언하는 라인, 상기 변수에 값을 할당하는 라인, 또는, 상기 제1 필수 라인의 실행 여부를 결정하는 라인을 제1 의존 라인으로 지정하는 동작; 상기 제1 필수 라인과 상기 제1 의존 라인을 매핑하여 취약점이 의미적으로 연결되어 있는 의미적 쌍을 생성하는 동작; 상기 취약 함수 및 상기 패치 함수 중 분석 대상으로 선택된 적어도 하나 이상의 제1 함수에 대한 모든 버전별 함수를 수집하는 동작; 및 각 버전의 함수가 상기 의미적 쌍을 포함하는 비율을 기초로 각 버전의 취약 여부를 판별하는 동작을 수행할 수 있다.

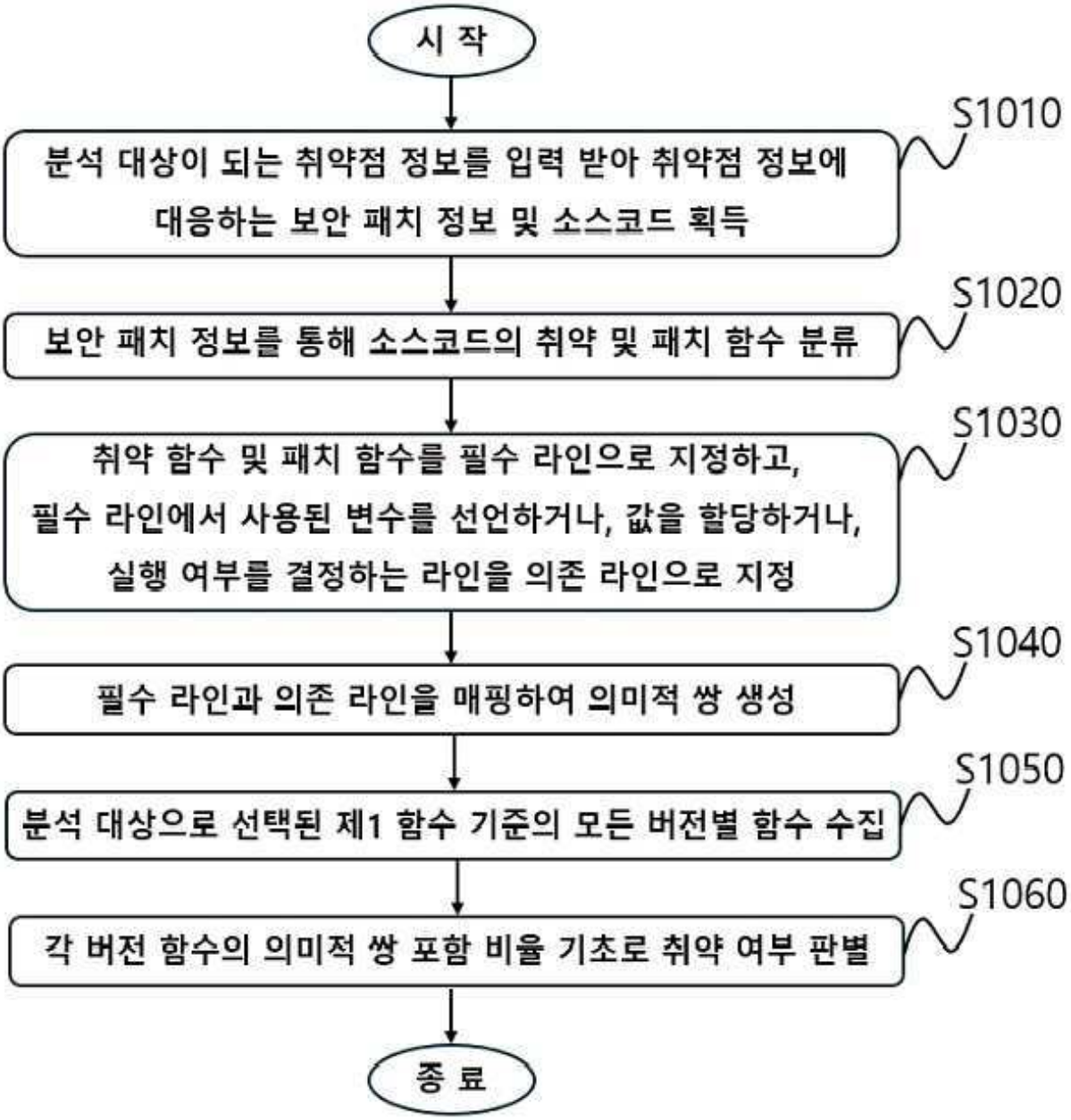
【대표도】

도 3

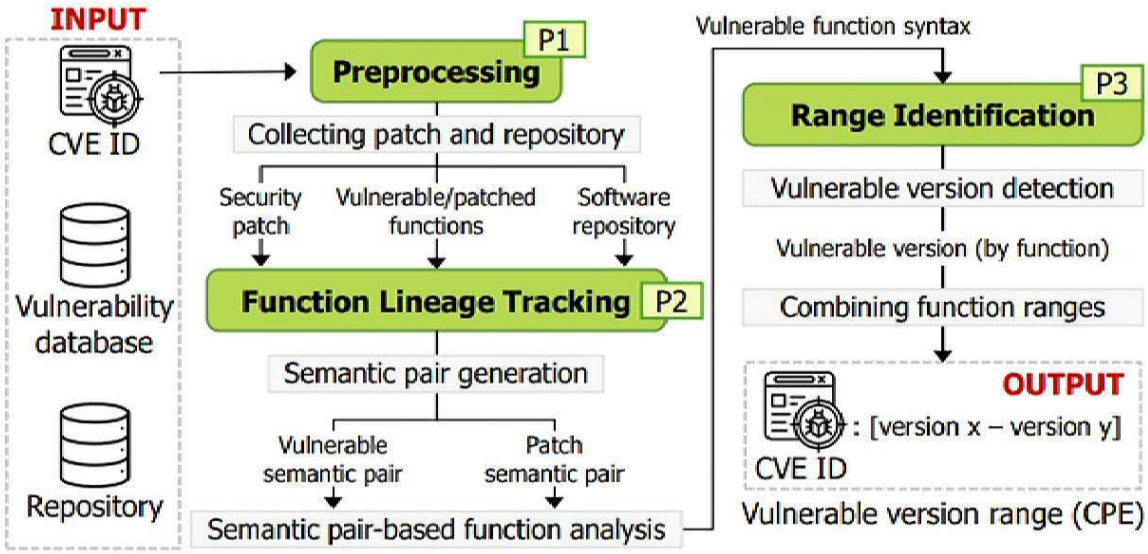
【도면】

【도 1】

【도 2】


시 작
S1010
분석 대상이 되는 취약점 정보를 입력 받아 취약점 정보에 대응하는 보안 패치 정보 및 소스코드 획득
S1020
보안 패치 정보를 통해 소스코드의 취약 및 패치 함수 분류
S1030
취약 함수 및 패치 함수를 필수 라인으로 지정하고, 필수 라인에서 사용된 변수를 선언하거나, 값을 할당하거나, 실행 여부를 결정하는 라인을 의존 라인으로 지정
S1040
필수 라인과 의존 라인을 매핑하여 의미적 쌍 생성
S1050
분석 대상으로 선택된 제1 함수 기준의 모든 버전별 함수 수집
S1060
각 버전 함수의 의미적 쌍 포함 비율 기초로 취약 여부 판별
종 료

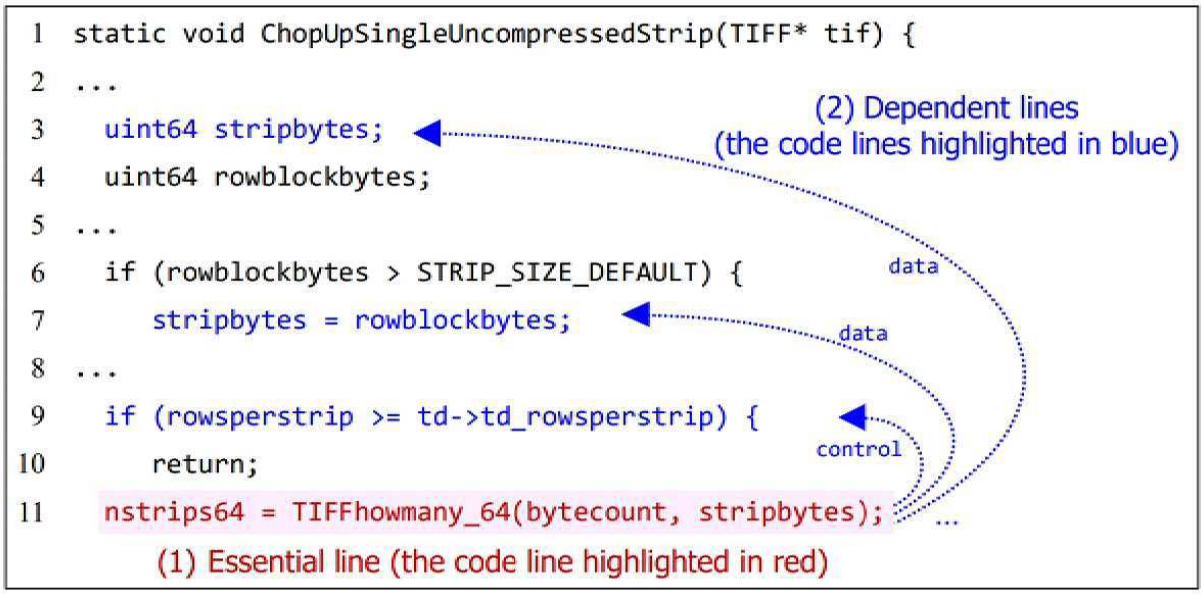
【도 3】


INPUT
CVE ID
Vulnerability database
Repository
P1
Preprocessing
Collecting patch and repository
Security patch
Vulnerable/patched functions
Software repository
Function Lineage Tracking
P2
Semantic pair generation
Vulnerable semantic pair
Patch semantic pair
Semantic pair-based function analysis
Vulnerable function syntax
P3
Range Identification
Vulnerable version detection
Vulnerable version (by function)
Combining function ranges
OUTPUT
: [version x – version y]
CVE ID
Vulnerable version range (CPE)

【도 4】

```
1 // CPE    : cpe:2.3:a:libtiff:libtiff:4.0.7:*:*:*:*:*:*:*
2 // Commit: 9a72a69e035ee70ff5c41541c8c61cd97990d018
3 // Path   : libtiff/tif_dirread.c
4 // Index : 3eec79c9d..570d0c327 100644
5 static void ChopUpSingleUncompressedStrip(TIFF* tif) {
6 ...
7   uint64 stripbytes;
8   uint64 rowblockbytes;
9 ...
10   if (rowblockbytes > STRIP_SIZE_DEFAULT) {
11    stripbytes = rowblockbytes;
12 ...
13   if (rowsperstrip >= td->td_rowsperstrip) {
14    return;
15 - nstrips64 = TIFFhowmany_64(bytecount, stripbytes);
16 - if ((nstrips64==0)||(nstrips64>0xFFFFFFFF))
17 -  return;
18 - nstrips32 = (uint32)nstrips64;
19 + nstrips = TIFFhowmany_32(td->td_imagelength, rowsperstrip);
20 + if( nstrips == 0 )
21 +  return;
```

【도 5】

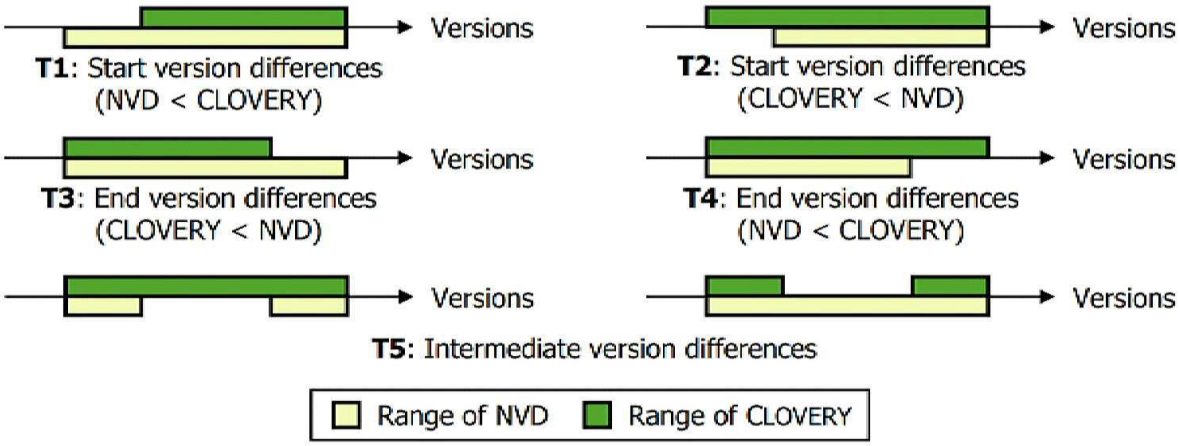


【도 6】

Validation method	#CVEs	NVD			VOFinder [11]		
		Precision	Recall	F1 score	Precision	Recall	F1 score
Reproduction	103	0.8627	0.8712	0.8669	**0.9973**	0.6645	0.7976
Author sites	54	0.8913	0.8612	**0.8759**	0.9129	0.5158	0.6592
Code review	1,345	0.8126	0.8566	0.8340	0.9450	0.5647	0.7070
Total	1,502	0.8189	0.8577	0.8379	0.9474	0.5698	0.7116

Validation method	#CVEs	V-SZZ [10]			CLOVERY		
		Precision	Recall	F1 score	Precision	Recall	F1 score
Reproduction	103	0.9781	0.6753	0.7990	0.8837	**0.9759**	**0.9275**
Author sites	54	**0.9212**	0.7292	0.8140	0.7302	**0.9245**	0.8160
Code review	1,345	0.9182	0.6258	0.7443	**0.9781**	**0.9894**	**0.9837**
Total	1,502	0.9224	0.6329	0.7507	**0.9627**	**0.9862**	**0.9743**

【도 7】



【도 8】

Type	Description	#CVEs	Ratio
T1	Start diff. (NVD < CLOVERY)	578	58.62% (578/986)
T2	Start diff. (CLOVERY < NVD)	325	32.96% (325/986)
T3	End diff. (CLOVERY < NVD)	194	19.68% (194/986)
T4	End diff. (NVD < CLOVERY)	357	36.21% (357/986)
T5	Intermediate diff.	184	18.66% (184/986)

【도 9】