(19) 대한민국특허청(KR)
(12) 등록특허공보(B1)

(45) 공고일자 2021년10월28일
(11) 등록번호 10-2318714
(24) 등록일자 2021년10월22일

(51) 국제특허분류(Int. Cl.)
G06F 21/57 (2013.01) *G06F 11/36* (2006.01)
G06F 21/56 (2013.01) *G06F 8/75* (2018.01)
(52) CPC특허분류
G06F 21/577 (2013.01)
G06F 11/3604 (2013.01)
(21) 출원번호 10-2020-0034311
(22) 출원일자 2020년03월20일
심사청구일자 2020년03월20일
(65) 공개번호 10-2021-0098297
(43) 공개일자 2021년08월10일
(30) 우선권주장
1020200011611 2020년01월31일 대한민국(KR)
(56) 선행기술조사문헌
KR101568224 B1*
KR101780233 B1*
KR1020190112524 A*
KR1020190113408 A*
*는 심사관에 의하여 인용된 문헌

(73) 특허권자
고려대학교 산학협력단
서울특별시 성북구 안암로 145, 고려대학교 (안암동5가)
(72) 발명자
이희조
장하진
양경석
(74) 대리인
이대호, 박건홍

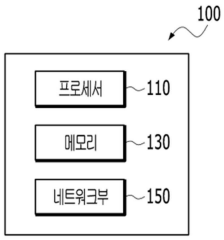
전체 청구항 수 : 총 13 항 심사관 : 구대성

(54) 발명의 명칭 **바이너리 코드 클론 기반 소프트웨어 취약점 탐지를 위한 컴퓨터 프로그램**

(57) 요 약

전술한 과제를 해결하기 위한 본 개시의 일 실시예에서, 하나 이상의 프로세서들에 의해 실행 가능한 컴퓨턴 판독가능 저장 매체에 저장된 컴퓨터 프로그램으로서, 상기 컴퓨터 프로그램은 상기 하나 이상의 프로세서로 하여금 소프트웨어의 바이너리 코드에 기반하여 취약점을 판별하기 위한 이하의 동작들을 수행하도록 하며, 상기 동작들은: 소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처(Vulnerability Signature)를 생성하는 동작, 타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문(Target Function Fingerprint)을 생성하는 동작 및 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작을 포함할 수 있다.

대 표 도



(52) CPC특허분류
G06F 21/563 (2013.01)
G06F 8/75 (2013.01)

이 발명을 지원한 국가연구개발사업

과제고유번호	R1914831
부처명	과학기술정보통신부
과제관리(전문)기관명	정보통신기획평가원
연구사업명	정보보호핵심원천기술개발
연구과제명	블록체인 플랫폼 보안취약점 자동분석 기술개발
기 여 율	1/1
과제수행기관명	고려대학교 산학협력단
연구기간	2019.06.01 ~ 2020.02.29

명 세 서

청구범위

청구항 1

컴퓨터 판독가능 저장 매체에 저장된 컴퓨터 프로그램으로서, 상기 컴퓨터 프로그램은 하나 이상의 프로세서에서 실행되는 경우, 상기 하나 이상의 프로세서로 하여금 소프트웨어의 바이너리 코드에 기반하여 취약점을 판별하기 위한 이하의 동작들을 수행하도록 하며, 상기 동작들은,

소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처(Vulnerability Signature)를 생성하는 동작;

타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문(Target Function Fingerprint)을 생성하는 동작; 및

상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작;

을 포함하고,

상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작은,

상기 취약점 시그니처 및 상기 타겟 함수 지문에 대한 1차 필터링을 수행하여 후보 취약점 시그니처 및 후보 타겟 함수 지문을 선별하는 동작;

을 포함하고, 그리고

상기 1차 필터링은,

상기 취약점 시그니처 및 상기 타겟 함수 지문 각각에서 취약점에 관련한 코드를 식별하기 위한 필터링인,

컴퓨터 판독가능 저장 매체에 저장된 컴퓨터 프로그램.

청구항 2

제 1 항에 있어서,

상기 취약점 정보는,

소스 코드 형태의 취약점 패치 정보, 취약점에 관련한 바이너리 코드 및 취약점이 패치된 바이너리 코드를 포함하는,

컴퓨터 판독가능 저장 매체에 저장된 컴퓨터 프로그램.

청구항 3

제 1 항에 있어서,

상기 소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성하는 동작은,

상기 취약점 정보에 기초하여 변경에 관련한 바이너리 코드를 식별하는 동작;

상기 식별된 변경에 관련한 바이너리 코드에 기초하여 취약 함수 지문(Vulnerability Function Fingerprint) 및 패치 함수 지문(Patched Function Fingerprint)을 생성하는 동작; 및

상기 취약 함수 지문 및 상기 패치 함수 지문을 포함하는 취약점 시그니처를 생성하는 동작;

을 포함하는,

컴퓨터 판독가능 저장 매체에 저장된 컴퓨터 프로그램.

청구항 4

제 3 항에 있어서,

상기 식별된 변경에 관련한 바이너리 코드에 기초하여 취약 함수 지문 및 패치 함수 지문을 생성하는 동작은,

상기 식별된 변경에 관련한 바이너리 코드를 기초하여 실행 가능한 함수 및 바이너리 코드의 속성 정보를 획득하는 동작;

상기 실행 가능한 함수에 기초하여 함수의 속성 정보를 획득하는 동작;

상기 함수 내의 독립적인 코드의 흐름에 기초하여 상기 함수를 하나 이상의 스트랜드(Strand)로 분할하는 동작; 및

상기 하나 이상의 스트랜드 각각에 상기 함수의 속성 정보 및 상기 바이너리의 속성 정보를 맵핑하여 상기 취약 함수 지문 및 상기 패치 함수 지문을 생성하는 동작;

을 포함하는,

컴퓨터 판독가능 저장 매체에 저장되는 컴퓨터 프로그램.

청구항 5

제 4 항에 있어서,

상기 스트랜드는,

하나의 변수의 값을 산출하기 위해 필요한 명령어들의 집합으로, 상기 유사도 비교에 기준이 되는 단위인,

컴퓨터 판독가능 저장 매체에 저장된 컴퓨터 프로그램.

청구항 6

제 1 항에 있어서,

상기 타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성하는 동작은,

상기 타겟 바이너리 코드를 기초하여 실행 가능한 타겟 함수 및 상기 타겟 바이너리 코드의 속성 정보를 획득하는 동작;

상기 타겟 함수에 기초하여 타겟 함수의 속성 정보를 획득하는 동작;

상기 타겟 함수 내의 독립적인 코드의 흐름에 기초하여 상기 타겟 함수를 하나 이상의 스트랜드로 분할하는 동작; 및

상기 하나 이상의 스트랜드 각각에 상기 타겟 함수의 속성 정보 및 상기 타겟 바이너리의 속성 정보를 맵핑하여 상기 타겟 함수 지문을 생성하는 동작;

을 포함하는,

컴퓨터 판독가능 저장 매체에 저장된 컴퓨터 프로그램.

청구항 7

제 1 항에 있어서,

상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교는,

상기 취약점 시그니처에 대응하는 함수 지문 및 상기 타겟 함수 지문 각각을 사전 결정된 단위로 분할하고, 그리고 분할된 모든 단위 중 얼마나 많은 단위가 유사한지 여부에 대한 비교를 수행하는 N-Gram 유사도 비교인,

컴퓨터 판독가능 저장 매체에 저장된 컴퓨터 프로그램.

청구항 8

제 1 항에 있어서,

상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작은,

상기 취약점 시그니처에 포함된 취약 함수 지문과 상기 타겟 함수 지문 각각에 포함된 함수 간의 제 1 유사도를 산출하는 동작;

상기 취약점 시그니처에 포함된 패치 함수 지문과 상기 타겟 함수 지문 각각에 포함된 함수 간의 제 2 유사도를 산출하는 동작; 및

상기 제 1 유사도 및 상기 제 2 유사도의 차이에 기초하여 상기 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작;

을 포함하는,

컴퓨터 판독가능 저장 매체에 저장된 컴퓨터 프로그램.

청구항 9

제 8 항에 있어서,

상기 제 1 유사도는 상기 소프트웨어에서 취약점이 판별될 확률과 양의 상관 관계를 가지며, 상기 제 2 유사도는 상기 소프트웨어에서 취약점이 판별될 확률과 음의 상관 관계를 가지는,

컴퓨터 판독가능 저장 매체에 저장된 컴퓨터 프로그램.

청구항 10

제 8 항에 있어서,

상기 제 1 유사도 및 상기 제 2 유사도의 차이에 기초하여 상기 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작은,

상기 제 1 유사도와 제 2 유사도의 차이가 사전 결정된 기준을 초과하는 경우, 상기 타겟 소프트웨어에 취약점이 존재하는 것으로 판별하는 동작;

을 포함하는,

컴퓨터 판독가능 저장 매체에 저장된 컴퓨터 프로그램.

청구항 11

삭제

청구항 12

제 1 항에 있어서,

상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작은,

상기 후보 취약점 시그니처 및 후보 타겟 함수 지문에 대한 2차 필터링을 수행하여 유사 스트랜드를 식별하는 동작;

을 포함하는,

컴퓨터 판독가능 저장 매체에 저장된 컴퓨터 프로그램.

청구항 13

컴퓨팅 장치의 프로세서에서 수행되는 소프트웨어의 타겟 바이너리 코드에 기반하여 취약점을 판별하기 위한 방법에 있어서,

컴퓨팅 장치에 포함된 프로세서가 소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성하는 단계;

상기 프로세서가 타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성하는 단계; 및

상기 프로세서가 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 단계;

를 포함하고,

상기 프로세서가 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 단계는,

상기 취약점 시그니처 및 상기 타겟 함수 지문에 대한 1차 필터링을 수행하여 후보 취약점 시그니처 및 후보 타겟 함수 지문을 선별하는 단계;

를 포함하고, 그리고

상기 1차 필터링은,

상기 취약점 시그니처 및 상기 타겟 함수 지문 각각에서 취약점에 관련한 코드를 식별하기 위한 필터링인,

컴퓨팅 장치의 프로세서에서 수행되는 소프트웨어의 바이너리 코드에 기반하여 취약점을 판별하기 위한 방법.

청구항 14

소프트웨어의 바이너리 코드에 기반하여 취약점을 판별하기 위한 컴퓨팅 장치로,

하나 이상의 코어를 포함하는 프로세서;

상기 프로세서에서 실행가능한 프로그램 코드들은 포함하는 메모리; 및

클라이언트와 데이터를 송수신하는 네트워크부;

를 포함하고, 그리고

상기 프로세서는,

소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성하고,

타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성하고, 그리고

상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하고,

상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 경우에, 상기 취약점 시그니처 및 상기 타겟 함수 지문에 대한 1차 필터링을 수행하여 후보

취약점 시그니처 및 후보 타겟 함수 지문을 선별하고, 그리고

상기 1차 필터링은,

상기 취약점 시그니처 및 상기 타겟 함수 지문 각각에서 취약점에 관련한 코드를 식별하기 위한 필터링인,

소프트웨어의 바이너리 코드에 기반하여 취약점을 판별하기 위한 컴퓨팅 장치.

발명의 설명

기 술 분 야

[0001] 본 개시는 소프트웨어의 취약점을 탐지하는 방법에 관한 것으로, 보다 구체적으로 소프트웨어의 바이너리에서 알려진 취약 코드의 코드 클론을 빠르고 정확하게 탐지하여 취약점의 유무를 판별하기 위한 컴퓨터 프로그램에 관한 것이다.

배 경 기 술

[0003] 라이브러리는 코드 작성에 자주 사용되는 로직을 재활용 가능하도록 만든 소프트웨어로, 특정 기능의 구현이 가능하며 타 소프트웨어와 결합할 수 있는 인터페이스를 가진 소프트웨어이다. 또한, 라이브러리는 사용자 자신이 사용하기 위해 만들 수 있고, 불특정 다수가 사용할 수 있도록 공개할 수도 있다. 라이브러리를 사용하면 필요한 로직을 직접 구현할 필요가 없이 라이브러리에서 제공하는 API (Application Program Interface)를 호출하는 것을 통해서 원하는 기능을 사용할 수 있게 된다.

[0004] 오픈소스 소프트웨어(Open Source Software, OSS)란 소스코드가 공개되어 있으면서 오프소스 라이선스를 준수하는 한 누구나 특별한 제한없이 재사용, 수정 및 재배포가 가능한 소프트웨어를 말한다.

[0005] 상술한 바와 같은 라이브러리 및 오픈소스 소프트웨어를 활용하는 경우, 개발 시간 및 비용을 단축시킬 수 있는 장점이 존재하나, 코드 클론의 발생 우려가 존재한다. 여기서, 코드 클론이란, 동일하거나, 또는 유사한 코드가 여러 소프트웨어 또는 하나의 소프트웨어 내에서 중복되는 현상으로, 주로 외부 코드의 재사용이 주 발생 원인이다. 이러한 코드 클론은 소프트웨어에 문제를 야기시킬 수 있다. 예를 들어, 특정 코드로부터 취약점이 발견되는 경우, 개발자는 해당 코드에 대응하는 모든 코드 클론을 식별하여 수정하여야 한다.

[0006] 다시 말해, 라이브러리 및 오픈소스 소프트웨어는 널리 사용되는 것을 목적으로 하기 때문에 취약점을 내포한 채 배포되는 경우, 수많은 소프트웨어에 취약점을 전파하는 창구로 작용할 수 있다. 또한, 특정 소프트웨어의 취약점을 패치한 후에도 해당 소프트웨어에 의존하는 타 소프트웨어에 패치가 적용되기까지 오랜 시간이 소요되는 문제가 발생할 우려가 있다. 또한, 하나의 소스코드는 컴파일 환경에 따라 다양한 형태의 바이너리로 컴파일이 가능(즉, 다양한 기능적 패치 가능)하나, 소프트웨어의 취약점 패치는 기능 패치에 비해 소규모인 경우가 많다. 이러한 환경에서, 바이너리에서 발생한 소규모의 코드 변화의 원인이 컴파일 환경 변화인지 또는 소프트웨어의 취약점 패치인지 여부를 판별하기 어려울 수 있다. 뿐만 아니라, 코드 클로닝된 소스코드를 컴파일한 바이너리에서 취약한 부분이 존재하는지 여부에 대한 판별이 어려울 수도 있다.

[0007] 이에 따라, 두 바이너리 함수의 유사도를 계산하여 코드 클론을 탐지하는 종래의 기술이 존재한다. 구체적으로, 종래의 바이너리 코드 클론 탐지 방법은, 원본 바이너리 코드 및 코드 클론 의심 바이너리 코드의 전부 또는 일부를 비교하거나, 구문 분석에 기초하여 추출된 주요 구문 또는 토큰을 비교함으로써, 코드 클론을 탐지할 수 있다.

[0008] 다만, 종래의 코드 클론 탐지 방법은, 바이너리 함수 쌍의 유사도를 계산하여 코드 클론의 존재 여부를 탐지할 뿐, 해당 코드에 취약점이 존재하는지 여부를 판별하기 어려우며, 규모 가변성의 한계가 존재할 수 있다. 또한, 종래의 방법은, 바이너리 코드에 대한 문자열 비교를 수행하므로, 비교적 많은 컴퓨팅 자원을 소모하며, 방대한 작업 시간이 요구될 수 있다.

[0009] 따라서, 소프트웨어 바이너리에서 알려진 취약점을 규모 가변성 있게 탐지하는 방법에 대한 수요가 당 업계에 존재할 수 있다.

선행기술문헌

특허문헌

[0010] (특허문헌 0001) 한국등록특허 2014-0001951

발명의 내용

해결하려는 과제

[0011] 본 개시는 전술한 배경기술에 대응하여 안출된 것으로, 소프트웨어의 바이너리에서 알려진 취약 코드의 코드 클론을 빠르고 정확하게 탐지하여 취약점의 유무를 판별하는 컴퓨터 프로그램을 제공하기 위한 것이다.

과제의 해결 수단

[0013] 전술한 과제를 해결하기 위한 본 개시의 일 실시예에 소프트웨어의 바이너리 코드에 기반하여 취약점을 판별하기 위한 컴퓨터 프로그램이 개시된다. 상기 컴퓨터 프로그램은 하나 이상의 프로세서에서 실행되는 경우, 상기 하나 이상의 프로세서들로 하여금 소프트웨어의 바이너리 코드에 기반하여 취약점을 판별하기 위한 이하의 동작들을 수행하도록 하며, 상기 동작들은, 소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성하는 동작, 타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성하는 동작 및 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작을 포함할 수 있다.

[0014] 대안적으로, 상기 취약점 정보는, 소스 코드 형태의 취약점 패치 정보, 취약점에 관련한 바이너리 코드 및 취약점이 패치된 바이너리 코드를 포함할 수 있다.

[0015] 대안적으로, 상기 소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성하는 동작은, 상기 취약점 정보에 기초하여 변경에 관련한 바이너리 코드를 식별하는 동작, 상기 식별된 변경에 관련한 바이너리 코드에 기초하여 취약 함수 지문(Vulnerability Function Fingerprint) 및 패치 함수 지문(Patched Function Fingerprint)을 생성하는 동작 및 상기 취약 함수 지문 및 상기 패치 함수 지문을 포함하는 취약점 시그니처를 생성하는 동작을 포함할 수 있다.

[0016] 대안적으로, 상기 식별된 변경에 관련한 바이너리 코드에 기초하여 취약 함수 지문 및 패치 함수 지문을 생성하는 동작은, 상기 식별된 변경에 관련한 바이너리 코드를 기초하여 실행 가능한 함수 및 바이너리 코드의 속성 정보를 획득하는 동작, 상기 실행 가능한 함수에 기초하여 함수의 속성 정보를 획득하는 동작, 상기 함수 내의 독립적인 코드의 흐름에 기초하여 상기 함수를 하나 이상의 스트랜드(Strand)로 분할하는 동작 및 상기 하나 이상의 스트랜드 각각에 상기 함수의 속성 정보 및 상기 바이너리의 속성 정보를 맵핑하여 상기 취약 함수 지문 및 상기 패치 함수 지문을 생성하는 동작을 포함할 수 있다.

[0017] 대안적으로, 상기 스트랜드는, 하나의 변수의 값을 산출하기 위해 필요한 명령어들의 집합으로, 상기 유사도 비교에 기준이 되는 단위일 수 있다.

[0018] 대안적으로, 상기 타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성하는 동작은, 상기 타겟 바이너리 코드를 기초하여 실행 가능한 타겟 함수 및 상기 타겟 바이너리 코드의 속성 정보를 획득하는 동작, 상기 타겟 함수에 기초하여 타겟 함수의 속성 정보를 획득하는 동작, 상기 타겟 함수 내의 독립적인 코드의 흐름에 기초하여 상기 타겟 함수를 하나 이상의 스트랜드로 분할하는 동작 및 상기 하나 이상의 스트랜드 각각에 상기 타겟 함수의 속성 정보 및 상기 타겟 바이너리의 속성 정보를 맵핑하여 상기 타겟 함수 지문을 생성하는 동작을 포함할 수 있다.

[0019] 대안적으로, 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교는, 상기 취약점 시그니처에 대응하는 함수 지문 및 상기 타겟 함수 지문 각각을 사전 결정된 단위로 분할하고, 그리고 분할된 모든 단위 중 얼마나 많은 단위가 유사한지 여부에 대한 비교를 수행하는 N-Gram 유사도 비교일 수 있다.

[0020] 대안적으로, 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작은, 상기 취약점 시그니처에 포함된 취약 함수 지문과 상기 타겟 함수 지문 각각에 포함된 함수 간의 제 1 유사도를 산출하는 동작, 상기 취약점 시그니처에 포함된 패치 함수 지문과 상기 타겟 함수 지문 각각에 포함된 함수 간의 제 2 유사도를 산출하는 동작 및 상기 제 1 유사도 및 상기 제 2 유사

도의 차이에 기초하여 상기 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작을 포함할 수 있다.

[0021] 대안적으로, 상기 제 1 유사도는 상기 소프트웨어에서 취약점이 판별될 확률과 양의 상관 관계를 가지며, 상기 제 2 유사도는 상기 소프트웨어에서 취약점이 판별될 확률과 음의 상관 관계를 가질 수 있다.

[0022] 대안적으로, 상기 제 1 유사도 및 상기 제 2 유사도의 차이에 기초하여 상기 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작은, 상기 제 1 유사도와 제 2 유사도의 차이가 사전 결정된 기준을 초과하는 경우, 상기 타겟 소프트웨어에 취약점이 존재하는 것으로 판별하는 동작을 포함할 수 있다.

[0023] 대안적으로, 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작은, 상기 취약점 시그니처 및 상기 타겟 함수 지문에 대한 1차 필터링을 수행하여 후보 취약점 시그니처 및 후보 타겟 함수 지문을 선별하는 동작을 포함하고, 그리고 상기 1차 필터링은, 상기 취약점 시그니처 및 상기 타겟 함수 지문 각각에서 취약점에 관련한 코드를 식별하기 위한 필터링일 수 있다.

[0024] 대안적으로, 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 동작은, 상기 후보 취약점 시그니처 및 후보 타겟 함수 지문에 대한 2차 필터링을 수행하여 유사 스트랜드를 식별하는 동작을 더 포함할 수 있다.

[0025] 본 개시의 다른 실시예에서 컴퓨팅 장치의 프로세서에서 수행되는 소프트웨어의 타겟 바이너리 코드에 기반하여 취약점을 판별하기 위한 방법이 개시된다. 상기 방법은, 컴퓨팅 장치에 포함된 프로세서가 소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성하는 단계, 상기 프로세서가 타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성하는 단계 및 상기 프로세서가 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하는 단계를 포함할 수 있다.

[0026] 본 개시의 또 다른 일 실시예에 소프트웨어의 바이너리 코드에 기반하여 취약점을 판별하기 위한 컴퓨팅 장치가 개시된다. 상기 컴퓨팅 장치는 하나 이상의 코어를 포함하는 프로세서, 상기 프로세서에서 실행가능한 프로그램 코드들은 포함하는 메모리 및 클라이언트와 데이터를 송수신하는 네트워크부를 포함하고, 그리고 상기 프로세서는, 소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성하고, 타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성하고, 그리고 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별할 수 있다.

발명의 효과

[0028] 본 개시는 소프트웨어의 바이너리에서 알려진 취약 코드의 코드 클론을 빠르고 정확하게 탐지하여 취약점의 유무를 판별하는 컴퓨터 프로그램을 제공할 수 있다.

도면의 간단한 설명

[0030] 다양한 양상들이 이제 도면들을 참조로 기재되며, 여기서 유사한 참조 번호들은 총괄적으로 유사한 구성요소들을 지칭하는데 이용된다. 이하의 실시예에서, 설명 목적을 위해, 다수의 특정 세부사항들이 하나 이상의 양상들의 총체적 이해를 제공하기 위해 제시된다. 그러나, 그러한 양상(들)이 이러한 구체적인 세부사항들 없이 실시될 수 있음은 명백할 것이다.

도 1은 본 개시의 일 실시예와 관련된 컴퓨팅 장치의 블록 구성도를 도시한다.

도 2는 본 개시의 일 실시예와 관련된 바이너리 코드 클론 기반 소프트웨어의 취약점 판별하기 위한 시스템에 대한 개략도를 도시한다.

도 3은 본 개시의 일 실시예와 관련된 취약점 시그니처를 전처리하는 과정에 대한 예시적인 순서도를 도시한다.

도 4는 본 개시의 일 실시예와 관련된 타겟 함수 지문을 전처리하는 과정에 대한 예시적인 순서도를 도시한다.

도 5는 본 개시의 일 실시예와 관련된 코드 클론 탐지 및 취약점 판별하는 과정에 대한 예시적인 순서도를 도시한다.

도 6은 본 개시의 일 실시예와 관련된 코드 흐름 쌍의 유사도 산출 방법을 설명하기 위한 예시도를 도시한다.

도 7은 본 개시의 일 실시예와 관련된 바이너리 코드 클론 기반 소프트웨어의 취약점 판별하기 위한 시스템의

처리 과정을 예시적으로 나타낸 모식도이다.

도 8은 본 개시의 일 실시예와 관련된 취약점 정보 전처리 및 바이너리 코드 전처리 과정을 예시적으로 나타낸 모식도이다.

도 9는 본 개시의 일 실시예와 관련된 코드 클론 탐지 및 취약점 판별 과정을 예시적으로 나타낸 모식도이다.

도 10은 본 개시의 일 실시예와 관련된 바이너리 코드 클론 기반 소프트웨어의 취약점 판별 방법에 대한 예시적인 순서도를 도시한다.

도 11은 바이너리 코드 클론 기반 소프트웨어의 취약점 판별 방법을 구현하기 위한 로직의 예시적인 순서도를 도시한다.

도 12는 본 개시의 일 실시예들이 구현될 수 있는 예시적인 컴퓨팅 환경에 대한 간략하고 일반적인 개략도를 도시한다.

발명을 실시하기 위한 구체적인 내용

[0031] 다양한 실시예들이 이제 도면을 참조하여 설명된다. 본 명세서에서, 다양한 설명들이 본 개시의 이해를 제공하기 위해서 제시된다. 그러나, 이러한 실시예들은 이러한 구체적인 설명 없이도 실행될 수 있음이 명백하다.

[0032] 본 명세서에서 사용되는 용어 "컴포넌트", "모듈", "시스템" 등은 컴퓨터-관련 엔티티, 하드웨어, 펌웨어, 소프트웨어, 소프트웨어 및 하드웨어의 조합, 또는 소프트웨어의 실행을 지칭한다. 예를 들어, 컴포넌트는 프로세서 상에서 실행되는 처리과정(procedure), 프로세서, 객체, 실행 스레드, 프로그램, 및/또는 컴퓨터일 수 있지만, 이들로 제한되는 것은 아니다. 예를 들어, 컴퓨팅 장치에서 실행되는 애플리케이션 및 컴퓨팅 장치 모두 컴포넌트일 수 있다. 하나 이상의 컴포넌트는 프로세서 및/또는 실행 스레드 내에 상주할 수 있다. 일 컴포넌트는 하나의 컴퓨터 내에 로컬화 될 수 있다. 일 컴포넌트는 2개 이상의 컴퓨터들 사이에 분배될 수 있다. 또한, 이러한 컴포넌트들은 그 내부에 저장된 다양한 데이터 구조들을 갖는 다양한 컴퓨터 판독가능한 매체로부터 실행할 수 있다. 컴포넌트들은 예를 들어 하나 이상의 데이터 패킷들을 갖는 신호(예를 들면, 로컬 시스템, 분산 시스템에서 다른 컴포넌트와 상호작용하는 하나의 컴포넌트로부터의 데이터 및/또는 신호를 통해 다른 시스템과 인터넷과 같은 네트워크를 통해 전송되는 데이터)에 따라 로컬 및/또는 원격 처리들을 통해 통신할 수 있다.

[0033] 더불어, 용어 "또는"은 배타적 "또는"이 아니라 내포적 "또는"을 의미하는 것으로 의도된다. 즉, 달리 특정되지 않거나 문맥상 명확하지 않은 경우에, "X는 A 또는 B를 이용한다"는 자연적인 내포적 치환 중 하나를 의미하는 것으로 의도된다. 즉, X가 A를 이용하거나; X가 B를 이용하거나; 또는 X가 A 및 B 모두를 이용하는 경우, "X는 A 또는 B를 이용한다"가 이들 경우들 어느 것으로도 적용될 수 있다. 또한, 본 명세서에 사용된 "및/또는"이라는 용어는 열거된 관련 아이템들 중 하나 이상의 아이템의 가능한 모든 조합을 지칭하고 포함하는 것으로 이해되어야 한다.

[0034] 또한, "포함한다" 및/또는 "포함하는"이라는 용어는, 해당 특징 및/또는 구성요소가 존재함을 의미하는 것으로 이해되어야 한다. 다만, "포함한다" 및/또는 "포함하는"이라는 용어는, 하나 이상의 다른 특징, 구성요소 및/또는 이들의 그룹의 존재 또는 추가를 배제하지 않는 것으로 이해되어야 한다. 또한, 달리 특정되지 않거나 단수 형태를 지시하는 것으로 문맥상 명확하지 않은 경우에, 본 명세서와 청구범위에서 단수는 일반적으로 "하나 또는 그 이상"을 의미하는 것으로 해석되어야 한다.

[0035] 당업자들은 추가적으로 여기서 개시된 실시예들과 관련되어 설명된 다양한 예시 적 논리적 블록들, 구성들, 모듈들, 회로들, 수단들, 로직들, 및 알고리즘 단계들이 전자 하드웨어, 컴퓨터 소프트웨어, 또는 양쪽 모두의 조합들로 구현될 수 있음을 인식해야 한다. 하드웨어 및 소프트웨어의 상호교환성을 명백하게 예시하기 위해, 다양한 예시 적 컴포넌트들, 블록들, 구성들, 수단들, 로직들, 모듈들, 회로들, 및 단계들은 그들의 기능성 측면에서 일반적으로 위에서 설명되었다. 그러한 기능성이 하드웨어로 또는 소프트웨어로서 구현되는지 여부는 전반적인 시스템에 부과된 특정 어플리케이션(application) 및 설계 제한들에 달려 있다. 숙련된 기술자들은 각각의 특정 어플리케이션들을 위해 다양한 방법들로 설명된 기능성을 구현할 수 있다. 다만, 그러한 구현의 결정들이 본 개시내용의 영역을 벗어나게 하는 것으로 해석되어서는 안된다.

[0037] 라이브러리 또는 오픈소스 소프트웨어는 널리 사용되는 것을 목적으로 하기 때문에 취약점을 내포한 채 타 사용자에게 배포되는 경우, 수많은 소프트웨어에 취약점을 전파하는 창구로 작용할 수 있다. 취약점 전파 문제를 완화하기 위하여 코드 콜론의 탐지하는 것은 가장 효과적인 방법 중 하나일 수 있다. 이에 따라, 취약점에 관련한

코드 클론을 탐지하여 소프트웨어의 취약점을 판별하기 위한 다양한 종래 기술들이 존재한다.

[0038] 일반적으로 코드 클론을 통한 취약점 탐지 방법으로는, 소프트웨어의 소스 코드에 기반한 탐지 방법 및 소프트웨어의 바이너리 코드에 기반한 탐지 방법 등이 존재할 수 있다.

[0039] 먼저, 소스 코드에 기반한 취약점 탐지 방법은, 사전 정의한 취약점에 관련한 소스 코드와 대상 소프트웨어의 소스 코드 간 유사도 비교 통해 수행될 수 있다. 즉, 소프트웨어의 소스 코드를 활용하여 해당 소프트웨어의 코드 클론 탐지 및 취약점을 판별할 수 있다. 상술한 방법은 효과적이고 효율적이나, 일반적으로 다수의 프로그램들은 상업용 운영 체제 및 독점 소프트웨어와 같은 바이너리 코드의 형태로 배포되고 있다. 즉, 프로그램 자체에서 소스 코드를 제공하지 않는 경우, 해당 방법을 통한 코드 클론 탐지 및 취약점 판별이 어려울 수 있다.

[0040] 추가적으로, 프로그램의 바이너리는 동일한 소스 코드에서 컴파일 되더라도 배포자에 따라 다른 기능과 취약성을 가질 수 있다. 또한, 프로그램 자체의 소스 코드에 취약점이 없더라도, 종속 라이브러리가 프로그램 자체의 보안 수준에 관계없이 취약점을 일으킬 수 있어 소스 코드를 활용한 취약점 탐지에 불완전성이 존재할 수 있다.

[0041] 이에 따라, 소프트웨어의 바이너리 코드에 기반하여 코드 클론을 탐지하는 종래의 기술들이 존재한다. 바이너리 코드에 기반하여 코드 클론 탐지는 별도의 소스 코드를 요구하지 않기 때문에 높은 활용도를 가질 수 있다.

[0042] 다만, 종래의 바이너리 코드에 기반한 코드 클론 탐지 방법들은, 단일 소스에서 여러 바이너리 표현이 컴파일 되는 등 컴파일 환경의 다양성으로 인해 확장성에 문제가 있을 수 있다. 또한, 바이너리 코드를 통해 두 대상 함수의 유사성을 계산할 수는 있으나, 함수가 실제 대상 바이너리에 존재하는지 여부를 판단하거나, 또는 취약점에 관련한 바이너리인지 여부를 판단하기는 어려울 수 있다.

[0043] 추가적으로, 종래의 바이너리 코드에 기반한 코드 클론 탐지 또는 취약점 판별 방법들은, 취약점 판별에 대한 높은 정확도와 빠른 처리 속도 모두를 확보하기 어려운 측면이 있다. 예를 들어, 종래의 방법들은, 바이너리 코드의 정적 속성에 의존하여 처리 속도를 향상시키는 반면, 컴파일 환경의 변경에 취약하여 정확도가 저하될 수 있다. 다른 예를 들어, 컴파일 환경으로 인한 변경 처리에 중점을 두어, 정확도를 향상시킬 수 있으나, 처리 속도의 저하를 수반할 우려가 존재한다.

[0044] 상술한 바와 같이, 바이너리 코드에 기반하여 코드 클론을 검출하고, 취약점을 판별하는 종래의 기술들은 취약점 판별 속도, 또는 정확성이 저하되는 우려가 존재한다.

[0045] 따라서, 본 개시는 바이너리 코드에 기반한 코드 클론 탐지 및 취약점 판별 과정에서, 보다 높은 정확도와 빠른 처리 속도를 제공하는 컴퓨팅 장치(100)를 제공할 수 있다. 본 개시의 컴퓨팅 장치(100)가 바이너리 코드에 기반하여 코드 클론 및 취약점을 판별하는 구체적인 방법들은 이하에서 후술하도록 한다.

[0047] 도 1은 본 개시의 일 실시예와 관련된 컴퓨팅 장치의 블록 구성도를 도시한다.

[0048] 본 개시의 일 실시예에 따르면, 컴퓨팅 장치(100)는 소프트웨어의 바이너리 코드에 기초하여 코드 클론을 탐지함으로써, 소프트웨어의 취약점 유무를 판별할 수 있다. 컴퓨팅 장치(100)는 도 1에 도시된 바와 같이, 프로세서(110), 메모리(130) 및 네트워크부(150)를 포함할 수 있다. 전술한 컴포넌트들은 예시적인 것으로서, 본 개시 내용의 권리범위가 전술한 컴포넌트들로 제한되지 않는다. 즉, 본 개시의 실시예들에 대한 구현 양태에 따라서 추가적인 컴포넌트들이 포함되거나 또는 전술한 컴포넌트들 중 일부가 생략될 수 있다.

[0049] 본 개시의 일 실시예에 따르면, 컴퓨팅 장치(100)는 클라이언트와 데이터를 송수신하는 네트워크부(150)를 포함할 수 있다. 네트워크부(150)는 컴퓨팅 장치(100)와 클라이언트와의 통신 기능을 제공할 수 있다. 예를 들어, 네트워크부(150)는 클라이언트로부터 특정 소프트웨어의 바이너리 코드에 대한 코드 클론 탐지 또는, 해당 소프트웨어의 취약점 판별에 관련된 요청을 수신할 수 있다. 본 개시의 일 실시예에 따른 네트워크부(150)는 공중전화 교환망(PSTN: Public Switched Telephone Network), xDSL(x Digital Subscriber Line), RADSL(Rate Adaptive DSL), MDSL(Multi Rate DSL), VDSL(Very High Speed DSL), UADSL(Universal Asymmetric DSL), HDSL(High Bit Rate DSL) 및 근거리 통신망(LAN) 등과 같은 다양한 유선 통신 시스템들을 사용할 수 있다.

[0050] 또한, 본 명세서에서 제시되는 네트워크부(150)는 CDMA(Code Division Multi Access), TDMA(Time Division Multi Access), FDMA(Frequency Division Multi Access), OFDMA(Orthogonal Frequency Division Multi Access), SC-FDMA(Single Carrier-FDMA) 및 다른 시스템들과 같은 다양한 무선 통신 시스템들을 사용할 수 있다. 본 명세서에서 설명된 기술들은 위에서 언급된 네트워크들로 제한되는 것은 아니며, 다른 네트워크들에서도 사용될 수도 있다.

[0051] 본 개시의 일 실시예에 따르면, 클라이언트로부터 발행된 쿼리에 따라서, 컴퓨팅 장치(100)의 후술될 동작들이 수행될 수 있다. 클라이언트는 컴퓨팅 장치(100)와 통신을 위한 메커니즘을 갖는 시스템에서의 임의의 형태의 노드(들)를 의미할 수 있다. 예를 들어, 이러한 클라이언트는 PC, 랩탑 컴퓨터, 워크스테이션, 단말 및/또는 네트워크 접속성을 갖는 임의의 전자 디바이스를 포함할 수 있다. 또한, 클라이언트는 에이전트, API(Application Programming Interface) 및 플러그-인(Plug-in) 중 적어도 하나에 의해 구현되는 임의의 서버를 포함할 수도 있다.

[0052] 본 개시의 일 실시예에 따르면, 메모리(130)는 프로세서(110)가 생성하거나 결정한 임의의 형태의 정보 및 네트워크부(150)가 수신한 임의의 형태를 저장할 수 있다. 메모리(130)는 본 개시의 일 실시예에 따른 소프트웨어의 취약점 판별을 수행하기 위한 컴퓨터 프로그램을 저장할 수 있으며, 저장된 컴퓨터 프로그램은 프로세서(110)에 의하여 판독되어 구동될 수 있다. 예를 들어, 메모리(130)는 코드 클론을 탐지하기 위한 취약 코드 클론의 집합에 관한 정보를 저장할 수 있다. 다른 예를 들어, 메모리(130)는 취약점에 관련한 코드와 취약점 판별에 대상이 되는 소프트웨어의 코드 각각에 대한 전처리 방법에 관한 정보를 저장할 수 있다. 전술한 메모리에 저장된 구체적인 정보들에 대한 기재는 예시일 뿐, 본 개시는 이에 제한되지 않는다.

[0053] 본 개시의 일 실시예에 따르면, 메모리(130)는 플래시 메모리 타입(flash memory type), 하드디스크 타입(hard disk type), 멀티미디어 카드 마이크로 타입(multimedia card micro type), 카드 타입의 메모리(예를 들어 SD 또는 XD 메모리 등), 램(Random Access Memory, RAM), SRAM(Static Random Access Memory), 롬(Read-Only Memory, ROM), EEPROM(Electrically Erasable Programmable Read-Only Memory), PROM(Programmable Read-Only Memory), 자기 메모리, 자기 디스크, 광디스크 중 적어도 하나의 타입의 저장매체를 포함할 수 있다. 컴퓨팅 장치(100)는 인터넷 상에서 컴퓨팅 장치(100)의 저장 기능을 수행하는 웹 스토리지(web storage)와 관련되어 동작할 수도 있다. 전술한 메모리에 대한 기재는 예시일 뿐, 본 개시는 이에 제한되지 않는다.

[0054] 본 개시의 일 실시예에 따르면, 프로세서(110)는 통상적으로 컴퓨팅 장치(100)의 전반적인 동작을 처리할 수 있다. 프로세서(110)는 위에서 살펴본 구성요소들을 통해 입력 또는 출력되는 신호, 데이터, 정보 등을 처리하거나 메모리(130)에 저장된 응용 프로그램을 구동함으로써, 클라이언트에게 적절한 정보(예컨대, 타겟 소프트웨어에 관련한 코드 클론 탐지, 또는 취약점 유무에 관한 정보) 또는 기능을 제공하거나 처리할 수 있다.

[0055] 본 개시의 일 실시예에 따르면, 프로세서(110)는 소프트웨어의 바이너리 코드에 기반하여 취약점을 탐지할 수 있다. 구체적으로, 프로세서(110)는 패치와 연관이 있는 코드의 흐름에 대한 정보를 포함하는 취약점 시그니처(650)를 대상 소프트웨어의 바이너리 코드(즉, 타겟 바이너리 코드(710))와 비교함으로써, 소프트웨어의 코드 클론 여부를 탐지할 수 있다. 또한, 바이너리 코드에 기반하여 탐지한 코드 클론을 통해 해당 소프트웨어에 취약점이 내포되어 있는지 여부를 판별할 수 있다. 즉, 프로세서(110)는 도 2에 도시된 바와 같이, 임의의 소프트웨어에 대한 취약점 존재 유무를 판별하기 위하여, 취약점 정보 전처리(200), 타겟 바이너리 전처리(300) 및 코드 클론 탐지 및 취약점 판별(400) 과정을 수행할 수 있다. 프로세서(110)가 수행하는 취약점 정보 전처리(200), 타겟 바이너리 전처리(300) 및 코드 클론 탐지 및 취약점 판별(400) 각각의 구체적인 동작은 도 3 내지 도 6을 참조하여 이하에서 서술하도록 한다.

[0056] 본 개시의 일 실시예에 따르면, 프로세서(110)는 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처(650)를 생성할 수 있다. 구체적으로, 프로세서(110)는 소프트웨어 내에서 알려진 취약점에 관련한 취약점 정보에 대한 전처리를 수행함으로써 소프트웨어의 취약점 판별에 기준이 되는 취약점 시그니처(650)를 생성할 수 있다. 취약점 정보는, 소스 코드 형태의 취약점 패치 정보, 취약에 관련한 바이너리 코드(즉, 취약 바이너리 코드(610)) 및 취약점이 패치된 바이너리 코드(즉, 패치 바이너리 코드(620))를 포함할 수 있다. 취약점 패치 정보는, 소프트웨어의 알려진 취약점을 식별하기 위하여 표준화된 정보로, 예를 들어, 보안 취약점 표준 코드(Common Vulnerabilities and Exposures)에 관련한 정보일 수 있다. 즉, 프로세서(110)는 공격에 활용가능한 취약점을 가지고 있는 코드 클론을 판별하기 위한 기준이 되는 취약점 정보를 전처리하여 취약점 시그니처(650)를 생성할 수 있다.

[0057] 프로세서(110)는 취약점 패치 정보, 취약 바이너리 코드(610) 및 패치 바이너리 코드(620)에 대한 전처리 과정을 통해 취약점 시그니처(650)를 생성할 수 있다. 취약점 시그니처(650)는, 소프트웨어에 알려진 취약점에 관한 정보를 포함할 수 있으며, 코드 클론 탐지 및 취약점 판별 과정에서 대상 소프트웨어의 타겟 바이너리와 매칭 가능한 형태로 전처리된 것일 수 있다. 구체적으로, 프로세서(110)는 취약점 정보에 기초하여 변경에 관련한 바이너리 코드를 식별할 수 있다. 이 경우, 변경에 관련한 바이너리 코드는, 소프트웨어의 보안 패치에 따라 명령어가 추가되거나 삭제됨에 따라 변경되는 코드를 의미할 수 있다. 프로세서(110)는 변경에 관련한 바이너리 코

드에 기초하여 취약 함수 지문(643) 및 패치 함수 지문을 생성할 수 있다. 또한, 프로세서(110)는 취약 함수 지문(643) 및 패치 함수 지문을 포함하는 취약점 시그니처(650)를 생성할 수 있다.

[0058] 보다 구체적으로, 도 3을 참조하면, 프로세서(110)는 취약점 패치 정보에 기초하여 취약 바이너리 코드(610)에 제거 표시(631)(Removal Mark)를 수행할 수 있다(211). 제거 표시(631)는 보안 패치에 의해 취약한 바이너리에서 삭제된 명령어와 관련한 것일 수 있다. 구체적으로, 프로세서(110)는 소스 코드 형태의 취약점 패치 정보에 기초하여 취약점에 관련하여 제거된 명령어를 식별할 수 있으며, 취약 바이너리 코드(610)에서 제거된 명령어에 대응하는 코드에 제거 표시(631)를 수행할 수 있다. 즉, 프로세서(110)는 취약 바이너리 코드(610)에 취약점 패치에 따라 제거된 소스 코드에 대응하는 바이너리 코드 부분에 관련하여 제거 표시(631)를 부여함으로써, 취약점 시그니처(650) 생성 과정에서 전체 바이너리에 대한 처리가 아닌, 취약점에 관련한 부분만을 식별하여 해당 명령어에 관련한 전처리만을 수행할 수 있다. 즉, 해당 제거 표시(631)를 통해 보다 효율적으로 취약점 시그니처(650)(즉, 취약 함수 지문(643))를 생성할 수 있다.

[0059] 프로세서(110)는 취약 바이너리 코드(610)에 기초하여 취약 함수 및 취약 바이너리 속성(Property) 정보를 획득할 수 있다(213). 취약 함수는, 취약 바이너리 코드(610)에 대응하는 명령어에 관련한 함수를 의미할 수 있다. 취약 바이너리 속성 정보는, 취약점을 가진 바이너리에 대응하는 프로퍼티(property)를 의미할 수 있다. 또한, 프로세서(110)는 취약 바이너리를 중간 언어(Intermediate Representation, IR)로 번역하고, 취약 함수를 분석하여 취약 함수의 속성 정보를 획득할 수 있다(215). 취약 함수의 속성 정보는, 해당 함수가 컴파일 환경 변화에 의해 영향을 받는지 여부에 관련된 안정성 정보일 수 있다. 즉, 취약 함수의 속성 정보는, 해당 취약 함수가 다른 컴파일 환경에서 안정적인지(즉, 컴파일 환경 변화에 영향을 받지 않음) 또는 불안정적인지(즉, 컴파일 환경 변화에 영향을 받음) 여부에 관한 정보를 포함할 수 있다. 이러한 취약 함수의 속성 정보는, 예를 들어, 컴파일 환경에서 변경되지 않는, 독립된 코드 흐름의 수 또는 함수의 호출 횟수 중 적어도 하나에 기반한 것일 수 있다.

[0060] 취약 바이너리를 중간 언어로 번역하는 것은, 여러 아키텍처의 바이너리를 사용하는 코드 클론 탐지 알고리즘을 고속화하기 위한 것일 수 있다. 즉, 중간 언어를 통해 여러 아키텍처에서 하나의 코드와 알고리즘을 활용할 수 있다. 또한, 하나의 코드와 알고리즘을 여러 아키텍처에서 활용할 수 있으므로, 함수를 스트랜드(strand)로 분할하기 위한 알고리즘을 보다 용이하게 작성할 수 있다.

[0061] 또한, 프로세서(110)는 취약 함수 내의 독립적인 코드의 흐름에 기초하여 취약 함수를 하나 이상의 스트랜드로 분할하고, 그리고 분할된 하나 이상의 스트랜드 각각에 대응하는 취약 함수의 속성 정보 및 취약 바이너리 속성 정보를 맵핑하여 취약 함수 지문(643)을 생성할 수 있다(217). 스트랜드는, 하나의 변수 값을 산출하기 위해 필요한 명령어들의 집합으로, 유사도 비교에 기준이 되는 단위일 수 있다. 즉, 스트랜드의 집합은 소프트웨어 시맨틱의 일부 또는 전체를 대표할 수 있다.

[0062] 즉, 프로세서(110)에 의해 생성된 취약 함수 지문(643)은, 소프트웨어의 취약점에 관련한 바이너리 코드가 스트랜드 단위로 구분된 집합을 의미할 수 있다.

[0063] 또한, 프로세서(110)는 취약점 패치 정보에 기초하여 패치 바이너리 코드(620)에 추가 표시(632)(Additional Mark)를 수행할 수 있다(221). 추가 표시(632)는 보안 패치에 의해 패치 바이너리에 추가된 명령어와 관련한 것일 수 있다. 구체적으로, 프로세서(110)는 소스 코드 형태의 취약점 패치 정보에 기초하여 취약점의 패치와 관련하여 추가된 명령어를 식별할 수 있으며, 패치 바이너리 코드(620)에서 추가된 명령어에 대응하는 코드에 추가 표시(632)를 수행할 수 있다. 즉, 프로세서(110)는 추가 바이너리 코드에 취약점 패치에 따라 추가된 소스 코드에 대응하는 바이너리 코드 부분에 관련하여 추가 표시(632)를 부여함으로써, 취약점 시그니처(650) 생성 과정에서 전체 바이너리에 대한 처리가 아닌, 패치된 취약점에 관련한 부분만을 식별하여 해당 명령어에 관련한 전처리만을 수행할 수 있다. 즉, 해당 추가 표시(632)를 통해 보다 효율적으로 취약점 시그니처(650)(즉, 패치 함수 지문)를 생성할 수 있다.

[0064] 프로세서(110)는 패치 바이너리 코드(620)에 기초하여 패치 함수 및 패치 바이너리 속성 정보를 획득할 수 있다(223). 패치 함수는, 패치 바이너리 코드(620)에 대응하는 명령어에 관련한 함수를 의미할 수 있다. 패치 바이너리 속성 정보는 취약점이 패치된 바이너리에 대응하는 프로퍼티를 의미할 수 있다. 또한, 프로세서(110)는 패치 바이너리를 중간 언어로 번역하고, 패치 함수를 분석하여 패치 함수의 속성 정보를 획득할 수 있다(225). 패치 함수의 속성 정보는, 해당 패치 함수가 다른 컴파일 환경에서 안정적인지(즉, 컴파일 환경 변화에 영향을 받지 않음) 또는 불안정적인지(즉, 컴파일 환경 변화에 영향을 받음) 여부에 관한 정보를 포함할 수 있다. 이러한 패치 함수의 속성 정보는, 예를 들어, 컴파일 환경에서 변경되지 않는, 독립된 코드 흐름의 수 또는 함수의 호

출 횟수 중 적어도 하나에 기반한 것일 수 있다.

[0065] 또한, 프로세서(110)는 패치 함수 내의 독립적인 코드의 흐름에 기초하여 패치 함수를 하나 이상의 스트랜드로 분할하고, 그리고 분할된 하나 이상의 스트랜드 각각에 대응하는 패치 함수의 속성 정보 및 패치 바이너리 속성 정보를 맵핑하여 패치 함수 지문을 생성할 수 있다.

[0066] 즉, 프로세서(110)에 의해 생성된 패치 함수 지문은, 소프트웨어의 취약점에 대한 패치가 완료된 바이너리 코드가 스트랜드 단위로 구분된 집합을 의미할 수 있다. 다시 말해, 패치 함수 지문은 소프트웨어에서 취약점을 야기시키지 않는 바이너리 코드의 집합일 수 있다.

[0067] 따라서, 프로세서(110)는 전술한 전처리 과정을 통해 취약 함수 지문(643) 및 패치 함수 지문을 포함하는 취약점 시그니처(650)를 생성할 수 있다(230). 다시 말해, 프로세서(110)는 패치의 영향을 받은 코드의 흐름을 스트랜드 단위로 저장하여 취약 함수 지문(643) 및 패치 함수 지문을 포함하는 취약점 시그니처(650)를 생성할 수 있다. 즉, 취약점 시그니처(650)는 취약점과 연관된 코드 흐름만을 스트랜드 단위로 포함하여 생성됨에 따라, 타겟 바이너리 코드(710)와의 유사도 비교 과정에서 모든 코드 흐름을 비교할 필요가 없이 유사 스트랜드 단위의 비교가 가능하도록 하여 코드 클론 탐지 시간을 획기적으로 감소시킬 수 있다.

[0068] 본 개시의 일 실시예에 따르면, 프로세서(110)는 타겟 바이너리 코드(710)에 대한 전처리를 수행하여 타겟 함수 지문(730)을 생성할 수 있다. 타겟 바이너리 코드(710)에 대한 전처리는, 전술한 취약 바이너리 코드(610) 및 패치 바이너리 코드(620)에 대한 전처리와 대응될 수 있다.

[0069] 구체적으로, 도 4를 참조하면, 프로세서(110)는 타겟 소프트웨어(즉, 취약점 판별의 대상이 되는 소프트웨어)의 타겟 바이너리 코드(710)에 기초하여 타겟 함수 및 타겟 바이너리 속성 정보를 획득할 수 있다(310). 타겟 함수는 타겟 바이너리 코드(710)에 대응하는 명령어에 관련한 함수를 의미할 수 있다. 타겟 바이너리 속성 정보는 타겟 소프트웨어의 타겟 바이너리에 대응하는 프로퍼티를 의미할 수 있다.

[0070] 추가적인 실시예에서, 타겟 함수 및 타겟 바이너리 속성 정보 획득에 기반이 되는 타겟 바이너리 코드(710)는, 전체 타겟 바이너리에 대한 처리가 아닌, 클라이언트의 요청에 관련한 명령어만을 처리하기 위한 임의의 표시(Mark)가 부여된 것일 수 있다. 즉, 프로세서(110)는 타겟 바이너리 코드(710)에 부여된 임의의 표시를 통해 타겟 함수 지문(730) 생성 과정에서 전체 타겟 바이너리에 대한 처리가 아닌, 클라이언트의 요청에 관련한 부분만을 식별하여 해당 명령어에 관련한 전처리만을 수행할 수 있다. 즉, 클라이언트의 요청에 관련한 임의의 표시를 통해 효율적으로 타겟 함수 지문(730)을 생성할 수 있다. 또한, 프로세서(110)는 타겟 바이너리를 중간 언어로 번역하고, 타겟 함수를 분석하여 타겟 함수의 속성 정보를 획득할 수 있다(320). 타겟 함수의 속성 정보는, 해당 타겟 함수가 다른 컴파일 환경에서 안정적인지(즉, 컴파일 환경 변화에 영향을 받지 않음) 또는 불안정적인지(즉, 컴파일 환경 변화에 영향을 받음) 여부에 관한 정보를 포함할 수 있다. 이러한 타겟 함수의 속성 정보는, 예를 들어, 컴파일 환경에서 변경되지 않는, 독립된 코드 흐름의 수 또는 함수의 호출 횟수 중 적어도 하나에 기반한 것일 수 있다.

[0071] 또한, 프로세서(110)는 타겟 함수 내의 독립적인 코드의 흐름에 기초하여 타겟 함수를 하나 이상의 스트랜드로 분할하고, 그리고 분할된 하나 이상의 스트랜드 각각에 대응하는 타겟 함수의 속성 정보 및 타겟 바이너리 속성 정보를 맵핑하여 타겟 함수 지문(730)을 생성할 수 있다. 즉, 프로세서(110)에 의해 생성된 타겟 함수 지문(730)은, 타겟 소프트웨어에 관련한 바이너리 코드가 스트랜드 단위로 구분된 집합을 의미할 수 있다.

[0072] 따라서, 프로세서(110)는 전술한 바와 같이, 패치의 영향을 받은 코드의 흐름을 스트랜드 단위로 저장하여 취약 함수 지문(643) 및 패치 함수 지문을 포함하는 취약점 시그니처(650)를 사전 생성할 수 있다. 또한, 프로세서(110)는 타겟 소프트웨어의 타겟 바이너리를 취약점 시그니처(650)와 비교 가능한 형태로 전처리하여 타겟 함수 지문(730)을 생성할 수 있다. 이 경우, 타겟 함수 지문(730) 또한, 코드의 흐름이 스트랜드 단위로 저장되어 있으므로, 취약점 시그니처(650)와 스트랜드 단위의 비교가 수행될 수 있어, 코드 클론 탐지 및 취약점 판별에 시간이 획기적으로 감소될 수 있다.

[0073] 본 개시의 일 실시예에 따르면, 프로세서(110)는 소프트웨어의 바이너리 코드에 기초하여 타겟 소프트웨어에서의 코드 클론 탐지 및 취약점 판별(400)을 수행할 수 있다. 프로세서(110)는 취약점 정보 전처리(200)를 통해 생성된 취약점 시그니처(650)와 타겟 소프트웨어의 타겟 바이너리 전처리(300)를 통해 생성된 타겟 함수 지문(730) 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별할 수 있다. 취약점 시그니처(650) 및 타겟 함수 지문(730) 간의 유사도 비교는, 취약점 시그니처(650)에 대응하는 함수 지문(즉, 취약 함수 지문(643) 및 타겟 함수 지문(730)) 및 타겟 함수 지문(730) 각각을 사전 결정된 단위로 분할하고, 그리고

분할된 모든 단위 중 얼마나 많은 단위가 유사한지 여부에 대한 비교를 수행하는 N-Gram 유사도 비교일 수 있다. 프로세서(110)는, 두 시퀀스를 N-Gram으로 분할한 뒤, 생성된 모든 N-Gram 중 얼마나 많은 N-Gram이 동일한가를 기준으로 각 함수 지문 간 유사도를 산출할 수 있다. 타 시퀀스 유사도 비교 방법과 달리 N-Gram 유사도는 시퀀스의 sub-ordering을 보존한다. 즉, N-Gram 유사도에 기반한 유사도 비교를 통해 바이너리의 형태가 변해도 함수 시맨틱의 sub-ordering이 보존되는 경향이 강해질 수 있다.

[0074] 예를 들어, 도 6에 도시된 바와 같이, 시퀀스x가 ABCDE를 포함하고 시퀀스y가 ABCZE를 포함하는 경우, 각각의 시퀀스는 Gx = {AB, ABC, BCD, CDE, DE} 및 Gy = {AB, ABC, BCZ, CZE, ZE}와 같이 N-Gram으로 분할될 수 있다. 이 경우, N-Gram으로 분할된 함수의 합집합은 8개(즉, AB, ABC, BCD, CDE, DE, BCZ, CZE, ZE)일 수 있으며, 함수의 교집합은 2개(AB, ABC)일 수 있다. 즉, 각 시퀀스의 N-Gram 유사도는 2/8(즉, 0.25)로 산출될 수 있다. 다시 말해, N-Gram 유사도 비교는 분할되어 생성된 모든 N-Gram 중 얼마나 많은 N-gram이 동일한가를 기준으로 하는 각 함수의 유사도 비교 방법일 수 있다. 즉, N-Gram 유사도 비교를 통해 각 함수 지문 간의 함수의 종류와 빈도를 적절히 비교하여 타겟 함수 지문(730) 중 취약점 시그니처(650)와 유사한 코드 흐름이 얼마나 존재하는지를 유사도로 산출할 수 있다. 전술한 시퀀스 각각의 함수, 분할 기준 및 유사도에 대한 구체적인 기재는 예시일 뿐, 본 개시는 이에 제한되지 않는다.

[0075] 또한, 프로세서(110)는 각 함수(즉, 취약점 시그니처(650)에 포함된 함수 지문 및 타겟 함수 지문(730)) 별 코드 흐름 집합 사이의 유사도에 기초하여 한 함수 쌍의 유사도를 산출할 수 있다.

$$Score(F_x, F_y) = \frac{1}{n}\sum_{i=1}^{n} \max_{j=[1,m]} \left(Sim(s_{x_i}, s_{y_j})\right)$$

[0076]

[0077] **<수식 1>**

[0078] 수식 1은 함수 쌍의 유사도를 어떻게 코드 흐름의 유사도에서 계산하는지 표현한다. F_x과 F_y는 함수를 나타내며, 각각 n개와 m개의 스트랜드를 가질 수 있다. F_x의 스트랜드는 s_x, F_y의 스트랜드는 s_y로 표현될 수 있다.

[0079] 프로세서(110)는 취약점 시그니처(650)에 포함된 취약 함수 지문(643)과 타겟 함수 지문(730) 각각에 포함된 함수 간의 제 1 유사도를 산출할 수 있다. 제 1 유사도는, 타겟 함수 지문(730)에 포함된 함수와 취약점에 관련한 바이너리 코드의 집합인 취약 함수 지문(643)에 포함된 함수 간의 유사도이므로, 타겟 소프트웨어에서 취약점이 판별된 확률과 양의 상관 관계(즉, 비례 관계)를 가질 수 있다.

[0080] 또한, 프로세서(110)는 취약점 시그니처(650)에 포함된 패치 함수 지문과 타겟 함수 지문(730) 각각에 포함된 함수 간의 제 2 유사도를 산출할 수 있다. 제 2 유사도는, 타겟 함수 지문(730)에 포함된 함수와 취약점을 야기시키지 않는(즉, 패치가 완료된) 바이너리 코드의 집합인 패치 함수 지문에 포함된 함수 간의 유사도이므로, 타겟 소프트웨어서 취약점이 판별된 확률과 음의 상관 관계(즉, 반비례 관계)를 가질 수 있다.

[0081] 또한, 프로세서(110)는 제 1 유사도 및 제 2 유사도의 차이에 기초하여 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별할 수 있다. 프로세서(110)는 제 1 유사도와 제 2 유사도의 차이가 사전 결정된 기준을 초과하는 경우, 타겟 소프트웨어에 취약점이 존재하는 것으로 판별할 수 있다.

[0082] 구체적으로, 프로세서(110)는 타겟 함수 지문(730)의 함수가 취약 함수 지문(643)의 함수와 유사할 경우 취약할 확률이 높은 것으로 판별할 수 있고, 그리고 타겟 함수 지문(730)의 함수가 패치 함수 지문의 함수와 유사할 경우 취약하지 않을 확률이 높은 것으로 판별할 수 있다. 이 경우, 컴퓨팅 장치(100)는 취약점 시그니처(650)가 정확하게 생성되지 못한 경우, 패치된 함수 지문과의 유사도 산출 과정에서 오탐이 발생할 가능성을 최소화시키기 위해 1보다 작은 계수를 곱해 확률을 낮게 반영할 수 있다. 프로세서(110)는 취약할 확률(즉, 제 1 유사도)에서 취약하지 않은 확률(즉, 제 2 유사도)을 뺀 값이 사전 결정된 기준값 보다 큰 경우, 취약점이 존재하는 것으로 판별할 수 있다.

[0083] 본 개시의 일 실시예에 따르면, 프로세서(110)는 취약점 시그니처(650) 및 타겟 함수 지문(730)에 대한 필터링을 수행하여 유사도 판별 과정을 보다 효율적으로 수행할 수 있다. 유사도 판별 과정을 효율적으로 수행하기 위한 구체적인 필터링 과정에 대한 구체적인 설명은 도 5를 참조하여 이하에서 후술하도록 하며, 중복 설명은 생

략하도록 한다.

[0084] 프로세서(110)는 함수의 속성 정보에 기초하여 취약점 시그니처(650) 및 타겟 함수 지문(730)에 대한 1차 필터링을 수행하여 후보 취약점 시그니처(651) 및 후보 타겟 함수 지문(731)을 선별할 수 있다(410). 1차 필터링은, 취약점 시그니처(650) 및 타겟 함수 지문(730) 각각에서 취약점에 관련한 코드를 식별하기 위한 필터링일 수 있다.

[0085] 1차 필터링은, 예를 들어, 취약점 시그니처(650)에 포함된 함수 지문 및 타겟 함수 지문(730) 각각에 대응하는 함수의 속성 정보에 기초하여 수행될 수 있다. 구체적으로, 취약점 시그니처(650)는, 취약 함수 지문(643) 및 패치 함수 지문(644)으로 구성될 수 있다. 또한, 취약 함수 지문(643), 패치 함수 지문(644) 및 타겟 함수 지문(730) 각각에는, 프로세서(110)에 의한 전처리 과정을 통해 함수의 속성 정보가 맵핑되어 있을 수 있다. 함수의 속성 정보는, 함수 각각이 컴파일 환경에서 안정적인지 여부에 관련한 정보일 수 있다. 즉, 프로세서(110)는 취약 함수 지문(643), 패치 함수 지문(644) 및 타겟 함수 지문(730) 각각에 맵핑된 함수의 속성 정보에 기초하여 취약 함수 지문(643), 패치 함수 지문(644) 및 타겟 함수 지문(730)에 포함된 함수 중 다양한 컴파일 환경에서 안정적인 함수를 제외시키는 1차 필터링(즉, 취약점과 관련 없는 함수 지문을 필터링)을 수행할 수 있다. 전술한, 함수의 속성에 관련한 1차 필터링에 대한 구체적인 기재는 예시일 뿐이며, 본 개시에서의 1차 필터링은, 각 함수 지문에서 취약점에 관련한 바이너리 코드들을 식별하기 위한 다양한 필터 조건을 포함할 수 있다.

[0086] 다시 말해, 프로세서(110)는 각 지문에 맵핑된 함수 속성 정보에 기초하여 취약점에 관련한 바이너리 코드에 대응하는 함수 지문만을 로드함으로써, 비교 과정에서의 전체 프로세서의 속도를 향상시키고, 그리고 리소스의 사용량을 저감시킬 수 있다. 즉, 프로세서(110)는 유사도 비교의 대상이 되는 함수 지문들을 축소시켜 성능을 향상시킬 수 있다.

[0087] 또한, 프로세서(110)는 후보 취약점 시그니처(651) 및 후보 타겟 함수 지문(731)에 대한 2차 필터링을 수행하여 각 함수 지문에서 유사 스트랜드를 식별할 수 있다(420). 후보 취약점 시그니처(651)에 포함된 후보 취약 함수 지문 및 후보 패치 함수 지문과 후보 타겟 함수 지문(731)은 프로세서(110)의 전처리 과정을 통해 스트랜드 단위로 구분되어 있을 수 있다. 이 경우, 각 함수 지문에 포함되는 하나 이상의 스트랜드 각각은, 하나의 변수 값을 계산하는데 필요한 명령 세트로써, 하나의 독립적인 데이터 흐름을 나타내는 것일 수 있다. 구체적으로, 프로세서(110)는 후보 취약 함수 지문에 포함된 하나 이상의 스트랜드와 후보 타겟 함수 지문(731)에 포함된 하나 이상의 스트랜드 중 유사한 스트랜드를 식별할 수 있다. 또한, 프로세서(110)는 후보 패치 함수 지문에 포함된 하나 이상의 스트랜드와 후보 타겟 함수 지문(731)에 포함된 하나 이상의 스트랜드 중 유사한 스트랜드를 식별할 수 있다. 즉, 각 후보 함수 지문에 포함된 하나 이상의 스트랜드 중 관련성을 가진 유사 스트랜드만을 유사성 비교를 위한 함수로써 식별할 수 있다. 다시 말해, 함수들의 대표성을 나타내는 스트랜드 단위의 비교를 통해 각 함수 지문에서 유사도 비교 대상이 되는 함수를 보다 효율적으로 식별할 수 있으므로, 처리 속도가 향상될 수 있다.

[0088] 또한, 프로세서(110)는 후보 취약점 시그니처(651) 및 후보 타겟 함수 지문(731) 각각에서 식별된 유사 스트랜드 간의 유사도를 산출할 수 있다(430). 이 경우, 각 스트랜드 간의 유사도 비교는 N-gram 유사도 비교일 수 있다. 또한, 프로세서(110)는 각 함수(즉, 후보 취약점 시그니처(651)에 포함된 함수 지문 및 후보 타겟 함수 지문(731)) 별 코드 흐름 집합 사이의 유사도에 기초하여 한 함수 쌍의 유사도를 산출할 수 있다. 전술한 유사도 비교 방법은 예시일 뿐이며 본 개시는 이에 제한되지 않는다.

[0089] 또한, 프로세서(110)는 유사도 비교 결과에 기초하여 코드 클론 및 취약점 유무를 판별하여 클라이언트에게 알림을 제공할 수 있다(440). 구체적으로, 프로세서(110)는 제 1 유사도 및 제 2 유사도의 차이가 사전 결정된 기준을 초과하는 경우, 타겟 소프트웨어에 취약점이 존재하는 것으로 판별할 수 있다. 또한, 프로세서(110)는 타겟 소프트웨어에 취약점이 존재하는 것으로 판별한 경우, 클라이언트에게 취약점에 관련한 알림을 제공할 수 있다. 취약점에 관련한 알림이 클라이언트에게 제공됨에 따라, 클라이언트는 취약점 판별의 대상이 되는 타겟 소프트웨어가 취약점을 포함하는지 여부를 인지할 수 있다.

[0090] 따라서, 프로세서(110)는 취약점 정보에 관련한 전처리를 통해 소프트웨어의 바이너리에 기반한 코드 클론 및 취약점 판별에 기준이 되는 취약점 시그니처(650)를 사전 생성할 수 있다. 또한, 프로세서(110)는 임의의 소프트웨어의 타겟 바이너리를 취약점 시그니처(650)와 비교 가능한 형태로 전처리하여 타겟 함수 지문(730)을 생성할 수 있다. 또한, 프로세서(110)는 취약점 시그니처(650)와 타겟 함수 지문(730)의 유사도 비교에 기초하여 타겟 소프트웨어의 코드 클론 및 취약점을 판별할 수 있다. 이 경우, 프로세서(110)에 의해 전처리된 각 함수 지문(즉, 취약 함수 지문(643), 패치 함수 지문(644) 및 타겟 함수 지문(730))은 스트랜드 단위로 저장되어 있으

며, 각각의 함수의 속성 정보가 맵핑되어 있으므로, 각 함수 지문의 유사도 비교 과정에 1차 필터링 및 2차 필터링이 수행될 수 있다. 이에 따라, 코드 클론 탐지 및 취약점 판별의 효율 및 처리 속도가 향상될 수 있다.

[0092] 도 7은 본 개시의 일 실시예와 관련된 바이너리 코드 클론 기반 소프트웨어의 취약점 판별하기 위한 시스템의 처리 과정을 예시적으로 나타낸 모식도이다. 도 7에서 도시되는 내용에 대한 특징 중 도 2 내지 5와 관련하여 앞서 설명된 특징과 중복되는 특징에 대해서는 도 2 내지 5에 기재된 내용을 참고하고 여기에서는 그 설명을 생략하기로 한다.

[0093] 도 7에 도시된 바와 같이, 프로세서(110)는 취약점 정보 전처리(200), 타겟 바이너리 전처리(300) 및 코드 클론 탐지 및 취약점 판별(400) 과정을 통해 임의의 소프트웨어를 대상으로 코드 클론 및 취약점 유무를 판별할 수 있다.

[0094] 자세히 설명하면, 프로세서(110)는 취약점 정보 전처리(200)를 수행할 수 있다. 취약점 정보는, 소스 코드 형태의 취약점 패치 정보, 취약 바이너리 코드(610) 및 패치 바이너리 코드(620)를 포함할 수 있다. 취약점 패치 정보는, 소프트웨어의 알려진 취약점을 식별하기 위하여 표준화된 정보로, 패치 전 또는, 패치 후의 코드 변화에 대한 정보를 포함할 수 있다. 프로세서(110)는 취약점 패치 정보에 기초하여 취약 바이너리 코드(610)에서 제거와 관련한 코드를 식별할 수 있고, 그리고 패치 바이너리 코드(620)에서 추가와 관련한 코드를 식별할 수 있다. 즉, 프로세서(110)는 취약점 패치 정보에 기초하여 취약 바이너리 코드(610) 및 패치 바이너리 코드(620) 각각에서 변경에 관련한 코드(즉, 제거 또는, 추가된 코드)를 식별할 수 있다. 이에 따라, 프로세서(110)는 취약 바이너리 코드(610) 및 패치 바이너리 코드(620) 각각에서 변경에 관련한 코드에 변경 표시(630)를 수행할 수 있다. 변경 표시(630)는 취약점 정보의 전처리 과정에서 전체 바이너리 코드에 대한 처리가 아닌, 취약점에 관련한 부분만을 식별하여 해당 명령어에 관련한 전처리만을 수행하기 위한 것일 수 있다. 프로세서(110)는 취약 바이너리 코드(610) 및 패치 바이너리 코드(620)에 대한 전처리(640)를 수행하여 취약점 시그니처(650)를 생성할 수 있다. 또한, 프로세서(110)는 취약 바이너리 코드(610) 및 패치 바이너리 코드(620)의 전처리를 통해 생성된 취약점 시그니처(650)를 타겟 함수 지문(730)과의 매칭을 위해 데이터베이스(660)에 저장할 수 있다.

[0095] 또한, 프로세서(110)는 타겟 바이너리에 대한 전처리(300)를 수행할 수 있다. 타겟 바이너리 코드(710)는 타겟 소프트웨어에 관한 바이너리 코드일 수 있으며, 타겟 바이너리 코드(710)에 대한 전처리는, 타겟 바이너리 코드를 취약점 시그니처와 매칭 가능한 형태로 가공하기 위한 것일 수 있다. 구체적으로, 프로세서(110)는 타겟 바이너리 코드(710)에 대한 전처리(720)를 통해 타겟 함수 지문(730)을 생성할 수 있다. 또한, 프로세서(110)는 생성된 타겟 함수 지문을 캐시 메모리(740)에 저장할 수 있다. 즉, 프로세서(110)는 코드 클론 및 취약점 판별에 대상이 되는 임의의 소프트웨어(즉, 타겟 소프트웨어)의 바이너리 코드에 대한 전처리(720)를 통해 타겟 함수 지문(730)을 생성할 수 있으며, 생성된 타겟 함수 지문(730)을 캐시 메모리(740)에 저장할 수 있다. 이러한 타겟 바이너리 코드에 대한 전처리는 클라이언트로부터 수신하는 타겟 소프트웨어의 코드 클론 및 취약점 판별에 관련한 쿼리 요청에 기초하여 수행되는 것일 수 있다. 즉, 프로세서(110)는 클라이언트의 쿼리 요청에 대응하는 최초 시점에 타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성할 수 있다.

[0096] 또한, 프로세서(110)는 임의의 소프트웨어에 대한 코드 클론 탐지 및 취약점 유무를 판별(400)할 수 있다. 구체적으로, 프로세서(110)는 취약점 정보의 전처리를 통해 생성된 취약점 시그니처(650)와 타겟 바이너리 코드(710)의 전처리를 통해 생성된 타겟 함수 지문(730)과의 유사도 비교를 통해 코드 클론을 탐지할 수 있다. 이 경우, 취약점 시그니처(650)는 소프트웨어에서 취약점에 관련한 코드들의 집합이므로, 타겟 바이너리 코드에 대응하는 타겟 함수 지문에서 코드 클론이 탐지되는 경우, 타겟 소프트웨어가 취약점을 내포하고 있는 것일 수 있다. 즉, 프로세서(110)는 취약점 시그니처(650)와 타겟 함수 지문(730)의 코드 흐름의 유사도를 통해 코드 클론 및 취약점을 판별할 수 있다. 또한, 프로세서(110)는 취약점 시그니처(650)와 타겟 함수 지문(730) 간의 코드 클론 탐지를 위한 매칭(810) 과정에서의 고속화를 위해 취약점 시그니처(650)와 타겟 함수 지문(730) 각각에 대한 필터링을 수행할 수 있다. 자세히 설명하면, 프로세서(110)는 취약점 시그니처(650) 및 타겟 함수 지문(730) 각각에 대한 지문 필터링(820)을 수행할 수 있다. 또한, 프로세서(110)는 지문 필터링(820) 된 취약점 시그니처(650) 및 타겟 함수 지문(730) 각각에 대한 스트랜드 필터링(830)을 수행할 수 있다. 이 경우, 지문 필터링(820) 및 스트랜드 필터링(830)은 도 5를 참조하여 설명한 1차 필터링 및 2차 필터링 각각에 대응하는 필터링일 수 있다. 즉, 프로세서(110)는 취약점 시그니처(650) 및 타겟 함수 지문(730) 각각에 대한 지문 필터링(820) 및 스트랜드 필터링(820)을 통해 코드 클론을 탐지하기 위한 지문 매칭(840)을 최소화시킬 수 있다. 이에 따라, 코드 클론 및 취약점 판별 과정이 고속화될 수 있다.

[0098] 도 8은 본 개시의 일 실시예와 관련된 취약점 정보 전처리 및 바이너리 코드 전처리 과정을 예시적으로 나타낸

모식도이다. 도 8에서 도시되는 내용에 대한 특징 중 도 2 내지 5와 관련하여 앞서 설명된 특징과 중복되는 특징에 대해서는 도 2 내지 5에 기재된 내용을 참고하고 여기에서는 그 설명을 생략하기로 한다.

[0099] 본 개시의 일 실시예에 따르면, 프로세서(110) 취약점 정보에 기초하여 취약점 시그니처(650)를 생성할 수 있다. 구체적으로, 프로세서(110)는 취약 바이너리 코드(610) 및 패치 바이너리 코드(620) 각각에 변경 표시(630)를 수행할 수 있다. 변경 표시(630)는 취약점 시그니처 생성 과정에서 전체 바이너리 코드에 대한 처리가 아닌, 취약점에 관련한 부분만을 식별하여 해당 명령어에 관련한 전처리만을 수행하기 위한 것일 수 있다. 프로세서(110)는 취약 바이너리 코드에 추가 표시(632)를 수행할 수 있다. 추가 표시(632)는 보안 패치에 의해 취약한 바이너리에서 제거된 명령어와 관련한 것일 수 있다. 프로세서(110)는 취약 바이너리 코드에 제거 표시(631)를 수행할 수 있다. 제거 표시(631)는 보안 패치에 의해 취약한 바이너리 코드에 추가된 명령어와 관련한 것일 수 있다.

[0100] 또한, 프로세서(110)는 취약 바이너리 코드(610) 및 제거 표시(631)에 기초하여 취약 바이너리 코드 전처리(641)를 수행할 수 있다. 또한, 프로세서(110)는 패치 바이너리 코드(620) 및 추가 표시(632)에 기초하여 패치 바이너리 코드 전처리(642)를 수행할 수 있다. 본 개시에서, 바이너리 코드들(즉, 취약 바이너리 코드, 패치 바이너리 코드 및 타겟 바이너리 코드)은 코드 클론 탐지를 위한 유사도 비교를 위해 동일한 형태로 가공되어야 하므로, 대응되는 전처리 과정을 포함할 수 있다. 구체적으로, 프로세서(110)는 취약 바이너리 코드(610) 및 제거 표시(631)에 기초하여 취약 바이너리 코드 전처리(641)를 수행할 수 있으며, 패치 바이너리 코드 및 추가 표시(632)에 기초하여 패치 바이너리 코드 전처리(642)를 수행할 수 있다. 또한, 프로세서(110)는 타겟 바이너리 코드 및 임의의 표시(920)에 기초하여 타겟 바이너리 코드 전처리를 수행할 수 있다. 임의의 표시(920)는 타겟 바이너리 코드에서 클라이언트의 요청에 관련한 코드 흐름만을 대상으로 전처리를 수행하기 위한 것일 수 있다. 임의의 표시는 클라이언트의 요청에 기반한 것으로 선택적인 것일 수 있다. 임의의 표시가 타겟 바이너리 코드에 부여된 경우, 프로세서(110)는 해당 부분의 코드 흐름만을 기초로 전처리를 수행하여 타겟 지문 함수를 생성할 수 있으므로, 전처리 과정의 효율성이 증대될 수 있다.

[0101] 상술한 바와 같이, 프로세서(110)에 의해 수행되는 바이너리 코드들(즉, 취약 바이너리 코드, 패치 바이너리 코드 및 타겟 바이너리 코드)에 대한 전처리는 대응되는 수행 과정을 포함하므로, 바이너리 코드 전처리(640)로 통합하여 구체적으로 후술하도록 한다.

[0102] 프로세서(110)는 바이너리 코드(910)를 분석하여 바이너리에 대응하는 함수 및 바이너리 속성 정보를 획득(930, 940)할 수 있다. 또한, 프로세서(110)는 바이너리 코드에 대응하는 함수를 중간 언어로 번역(950)할 수 있다. 바이너리 코드에 대응하는 함수를 중간 언어로 번역함으로써, 여러 아키텍처의 바이너리를 사용하는 코드 클론 탐지 알고리즘을 고속화할 수 있다. 즉, 중간 언어를 통해 여러 아키텍처에서 하나의 코드와 알고리즘을 활용할 수 있다. 또한, 하나의 코드와 알고리즘을 여러 아키텍처에서 활용할 수 있으므로, 함수를 스트랜드(strand)로 분할하기 위한 알고리즘을 보다 용이하게 작성할 수 있다. 또한, 프로세서(110)는 함수를 스트랜드로 분할(960)하고, 그리고 함수의 속성 정보를 추출(970)할 수 있다. 즉, 프로세서(110)는 바이너리 함수를 스트랜드(961)로 분할하고, 그리고 분할된 스트랜드 각각에 대응하는 함수의 속성 정보(971) 및 바이너리 속성 정보(941)를 맵핑하여 함수 지문(980)을 생성할 수 있다.

[0103] 즉, 상술한 과정에서 바이너리 코드(910)가 취약 바이너리 코드(610)인 경우, 프로세서(110)는 취약 함수 지문(643)을 생성할 수 있으며, 패치 바이너리 코드(620)인 경우, 프로세서(110)는 패치 함수 지문(644)을 생성할 수 있다. 또한, 바이너리 코드(910)가 타겟 바이너리 코드(710)인 경우, 프로세서(110)는 타겟 함수 지문을 생성할 수 있다.

[0104] 프로세서(110)는 취약 함수 지문(643) 및 패치 함수 지문(644)을 포함하는 취약점 시그니처를 생성할 수 있으며, 취약 함수 지문(643) 및 패치 함수 지문(644) 각각을 타겟 함수 지문(730)과 비교하여 타겟 소프트웨어에서의 코드 클론 탐지 및 취약점 판별을 수행할 수 있다.

[0106] 도 9는 본 개시의 일 실시예와 관련된 코드 클론 탐지 및 취약점 판별 과정을 예시적으로 나타낸 모식도이다. 도 9에서 도시되는 내용에 대한 특징 중 도 2 내지 5와 관련하여 앞서 설명된 특징과 중복되는 특징에 대해서는 도 2 내지 5에 기재된 내용을 참고하고 여기에서는 그 설명을 생략하기로 한다.

[0107] 본 개시의 일 실시예에 따르면, 프로세서(110)는 취약점 시그니처(650) 및 타겟 함수 지문(730) 간의 유사도 비교를 통해 코드 클론을 탐지하여 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별할 수 있다. 프로세서(110)는 취약점 시그니처(650) 및 타겟 함수 지문(730) 각각에 필터링을 수행하여 각 함수 지문 간의 코드 클론

탐지 속도를 극대화시킬 수 있다. 구체적으로, 프로세서(110)는 취약점 시그니처(650) 및 타겟 함수 지문(730) 각각에 대한 지문 필터링(820)을 수행하여 후보 취약점 시그니처(651) 및 후보 타겟 함수 지문(731)을 선별할 수 있다. 지문 필터링(820)은, 취약점 시그니처(650) 및 타겟 함수 지문(730) 각각에서 취약점에 관련한 코드를 식별하기 위한 필터링일 수 있다.

[0108] 지문 필터링(820)은, 예를 들어, 취약점 시그니처에 포함된 취약 함수 지문(643)과 패치 함수 지문(644) 및 타겟 함수 지문(730) 각각에 대응하는 함수의 속성 정보에 기초하여 수행될 수 있다. 함수의 속성 정보는 함수 각각이 컴파일 환경에서 안정적인지 여부에 관련한 정보일 수 있다. 즉, 프로세서(110)는 지문 필터링(820)을 통해 취약점 시그니처(650) 및 타겟 함수 지문(730) 각각에 포함된 함수 중 다양한 컴파일 환경에서 안정적인 함수를 제외시키는 지문 필터링(820)을 수행함으로써, 취약점에 관련한 바이너리 코드에 대응하는 함수 지문만을 로드할 수 있다. 전술한, 함수의 속성에 관련한 지문 필터링에 대한 구체적인 기재는 예시일 뿐이며, 본 개시에서의 지문 필터링은, 각 함수 지문에서 취약점에 관련한 바이너리 코드들을 식별하기 위한 다양한 필터 조건을 포함할 수 있다.

[0109] 이에 따라, 취약점 시그니처(650)와 타겟 함수 지문(730) 간의 비교 과정에서 전체 프로세스의 속도를 향상시키고, 그리고 리소스 사용량을 저감시킬 수 있다. 즉, 프로세서(110)는 유사도 비교의 대상이 되는 함수 지문들을 축소시켜 성능을 향상시킬 수 있다.

[0110] 또한, 프로세서(110)는 후보 취약점 시그니처(651) 및 후보 타겟 함수 지문(731)에 대한 지문 매칭(840)을 통해 타겟 소프트웨어에서의 코드 클론을 탐지하고, 그리고 취약점 유무를 판별할 수 있다. 지문 매칭(840)은 각 함수 지문의 유사도를 비교하기 위한 것일 수 있다. 프로세서(110)는 후보 취약점 시그니처(651) 및 후보 타겟 함수 지문(731)에 대한 스트랜드 필터링(830)을 수행하여 지문 매칭(840)의 대상이 되는 함수들을 식별할 수 있다. 구체적으로, 후보 취약점 시그니처(651)에 포함된 후보 취약 함수 지문 및 후보 패치 함수 지문과 후보 타겟 함수 지문(731)은 프로세서(110)의 전처리 과정을 통해 스트랜드 단위로 구분되어 있을 수 있다. 이 경우, 각 함수 지문의 하나 이상의 스트랜드 각각은, 하나의 변수 값을 계산하는데 필요한 명령어 세트로서, 하나의 독립적인 데이터 흐름을 나타내는 것일 수 있다. 즉, 프로세서(110)는 스트랜드 필터링(830)을 수행하여 각 후보 함수 지문에 포함된 하나 이상의 스트랜드 중 관련성을 가진 유사 스트랜드를 유사성 비교를 위한 함수로써 식별할 수 있다. 다시 말해, 함수들의 대표성을 나타내는 스트랜드 단위의 비교를 통해 각 함수 지문에서 유사도 비교 대상이 되는 함수를 보다 효율적으로 식별할 수 있으므로, 처리 속도가 향상될 수 있다. 프로세서(110)는 전술한 과정을 통해 식별된 함수 간의 유사도 비교를 통해 코드 클론을 탐지하고, 그리고 취약점 유무를 판별할 수 있다. 또한, 프로세서(110)는 타겟 소프트웨어에 대한 코드 클론 및 취약점에 관련한 알림을 클라이언트에게 전송(850)할 수 있다.

[0112] 본 개시의 바이너리 코드 클론 기반 소프트웨어 취약점 탐지법을 이용하여 타겟인 805MB의 데비안 9.0 C/C++ 바이너리에서 6종의 취약점 시그니처의 코드 클론이 존재하는지 검사하였다. 타겟 바이너리는 총 5,420개이며, 3,266,604개의 함수로 구성되어 있다. 이 경우, 취약점 시그니처는 알려진 6종의 취약점을 대표한다.

[0113] 타겟 바이너리 내 존재하는 알려진 취약점의 코드 클론 7개 중 6개를 성공적으로 탐지하여 85.7%의 정확도를 달성하였다. 또한 2종의 취약점 시그니처가 여러 타겟 바이너리에서 탐지되었으며, 동일한 소스 코드에서 컴파일된 바이너리뿐만 아니라 취약점을 가지고 있지만 다른 소스 코드에서 컴파일된 바이너리에서도 취약점을 탐지할 수 있음을 확인하였다.

[0114] 하나의 타겟 바이너리 전처리에 평균 3,450ms가 소요되었으며, 이를 각각의 취약점 시그니처와 비교할 때 평균적으로 2ms가 소요되었다. 병렬 프로그래밍을 적용한 결과 62분만에 모든 타겟 바이너리를 전처리하고 취약점 시그니처와 매칭할 수 있었으며, 전처리된 타겟 바이너리를 캐싱한 경우 코드 클론을 3분만에 매칭할 수 있었다.

[0115] 즉, 본 개시는 바이너리 함수 쌍의 유사도만을 계산하는 종래의 기술과는 달리, 패치 전 함수와 패치 후 함수의 변화 내역에 기초하여 취약점 시그니처를 생성함으로써, 타겟 바이너리에 취약점이 존재하는지 여부를 판별할 수 있다.

[0116] 또한, 본 개시는 타겟 바이너리 내 존재하는 알려진 취약점 중 85.7%의 취약점을 탐지하였으며, 하나의 취약점이 여러 바이너리에 존재하는 경우 모두 탐지가 가능함을 증명하였다.

[0117] 추가적으로, 본 개시에서 하나의 타겟 바이너리 전처리에는 평균 3,450ms가 소요되었으며, 이를 각각의 취약점 시그니처와 비교할 때 평균적으로 2ms가 소요되었다. 동일 환경에서 기존 기술인 Esh 대비 코드 흐름 한 쌍의

유사도를 약 15,000배 빠르게 계산할 수 있다. Esh의 경우 코드 흐름 한 쌍 계산에 평균 $8.70\times10^{6}\mu s$가 소요되었지만, 본 발명의 경우 평균 134μs만에 처리할 수 있었다. 따라서, 본 개시의 컴퓨팅 장치(100)는 종래 기술에 대비하여 임의의 소프트웨어 바이너리가 주어진 경우, 알려진 취약점을 빠르고 정확하게 탐지하는 효과를 제공할 수 있다.

[0119] 도 10은 본 개시의 일 실시예와 관련된 바이너리 코드 클론 기반 소프트웨어의 취약점 판별 방법에 대한 예시적인 순서도를 도시한다.

[0120] 본 개시의 일 실시예에 따르면, 컴퓨팅 장치(100)는 소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성할 수 있다(1010).

[0121] 본 개시의 일 실시예에 따르면, 컴퓨팅 장치(100)는 타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성할 수 있다(1020).

[0122] 본 개시의 일 실시예에 따르면, 컴퓨팅 장치(100)는 취약점 시그니처 및 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별할 수 있다(1030).

[0123] 전술한 도 10의 도시된 단계들은 필요에 의해 순서가 변경될 수 있으며, 적어도 하나 이상의 단계가 생략 또는 추가될 수 있다. 즉, 전술한 단계는 본 개시의 실시예에 불과할 뿐, 본 개시의 권리 범위는 이에 제한되지 않는다.

[0125] 도 11은 바이너리 코드 클론 기반 소프트웨어의 취약점 판별 방법을 구현하기 위한 로직을 도시한다.

[0126] 본 개시의 일 실시예에 따르면, 컴퓨팅 장치(100)는 다음과 같은 로직에 의해 구현될 수 있다.

[0127] 본 개시의 일 실시예에 따르면, 컴퓨팅 장치(100)는 소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성하기 위한 로직(1110), 타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성하기 위한 로직(1120) 및 취약점 시그니처 및 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하기 위한 로직(1130)을 포함할 수 있다.

[0128] 대안적으로, 상기 취약점 정보는, 소스 코드 형태의 취약점 패치 정보, 취약점에 관련한 바이너리 코드 및 취약점이 패치된 바이너리 코드를 포함할 수 있다.

[0129] 대안적으로, 상기 소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성하기 위한 로직은, 상기 취약점 정보에 기초하여 변경에 관련한 바이너리 코드를 식별하기 위한 로직, 상기 식별된 변경에 관련한 바이너리 코드에 기초하여 취약 함수 지문(Vulnerability Function Fingerprint) 및 패치 함수 지문(Patched Function Fingerprint)을 생성하기 위한 로직 및 상기 취약 함수 지문 및 상기 패치 함수 지문을 포함하는 취약점 시그니처를 생성하기 위한 로직을 포함할 수 있다.

[0130] 대안적으로, 상기 식별된 변경에 관련한 바이너리 코드에 기초하여 취약 함수 지문 및 패치 함수 지문을 생성하기 위한 로직은, 상기 식별된 변경에 관련한 바이너리 코드를 기초하여 실행 가능한 함수 및 바이너리 코드의 속성 정보를 획득하기 위한 로직, 상기 실행 가능한 함수에 기초하여 함수의 속성 정보를 획득하기 위한 로직, 상기 함수 내의 독립적인 코드의 흐름에 기초하여 상기 함수를 하나 이상의 스트랜드(Strand)로 분할하기 위한 로직 및 상기 하나 이상의 스트랜드 각각에 상기 함수의 속성 정보 및 상기 바이너리의 속성 정보를 맵핑하여 상기 취약 함수 지문 및 상기 패치 함수 지문을 생성하기 위한 로직을 포함할 수 있다.

[0131] 대안적으로, 상기 스트랜드는, 하나의 변수의 값을 산출하기 위해 필요한 명령어들의 집합으로, 상기 유사도 비교에 기준이 되는 단위일 수 있다.

[0132] 대안적으로, 상기 타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성하기 위한 로직은, 상기 타겟 바이너리 코드를 기초하여 실행 가능한 타겟 함수 및 상기 타겟 바이너리 코드의 속성 정보를 획득하기 위한 로직, 상기 타겟 함수에 기초하여 타겟 함수의 속성 정보를 획득하기 위한 로직, 상기 타겟 함수 내의 독립적인 코드의 흐름에 기초하여 상기 타겟 함수를 하나 이상의 스트랜드로 분할하기 위한 로직 및 상기 하나 이상의 스트랜드 각각에 상기 타겟 함수의 속성 정보 및 상기 타겟 바이너리의 속성 정보를 맵핑하여 상기 타겟 함수 지문을 생성하기 위한 로직을 포함할 수 있다.

[0133] 대안적으로, 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교는, 상기 취약점 시그니처에 대응하는 함수 지문 및 상기 타겟 함수 지문 각각을 사전 결정된 단위로 분할하고, 그리고 분할된 모든 단위 중 얼

마나 많은 단위가 유사한지 여부에 대한 비교를 수행하는 N-Gram 유사도 비교일 수 있다.

[0134] 대안적으로, 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하기 위한 로직은, 상기 취약점 시그니처에 포함된 취약 함수 지문과 상기 타겟 함수 지문 각각에 포함된 함수 간의 제 1 유사도를 산출하기 위한 로직, 상기 취약점 시그니처에 포함된 패치 함수 지문과 상기 타겟 함수 지문 각각에 포함된 함수 간의 제 2 유사도를 산출하기 위한 로직 및 상기 제 1 유사도 및 상기 제 2 유사도의 차이에 기초하여 상기 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하기 위한 로직을 포함할 수 있다.

[0135] 대안적으로, 상기 제 1 유사도는 상기 소프트웨어에서 취약점이 판별될 확률과 양의 상관 관계를 가지며, 상기 제 2 유사도는 상기 소프트웨어에서 취약점이 판별될 확률과 음의 상관 관계를 가질 수 있다.

[0136] 대안적으로, 상기 제 1 유사도 및 상기 제 2 유사도의 차이에 기초하여 상기 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하기 위한 로직은, 상기 제 1 유사도와 제 2 유사도의 차이가 사전 결정된 기준을 초과하는 경우, 상기 타겟 소프트웨어에 취약점이 존재하는 것으로 판별하기 위한 로직을 포함할 수 있다.

[0137] 대안적으로, 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하기 위한 로직은, 상기 취약점 시그니처 및 상기 타겟 함수 지문에 대한 1차 필터링을 수행하여 후보 취약점 시그니처 및 후보 타겟 함수 지문을 선별하기 위한 로직을 포함하고, 그리고 상기 1차 필터링은, 상기 취약점 시그니처 및 상기 타겟 함수 지문 각각에서 취약점에 관련한 코드를 식별하기 위한 필터링일 수 있다.

[0139] 대안적으로, 상기 취약점 시그니처 및 상기 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하기 위한 로직은, 상기 후보 취약점 시그니처 및 후보 타겟 함수 지문에 대한 2차 필터링을 수행하여 유사 스트랜드를 식별하기 위한 로직을 더 포함할 수 있다.

[0140] 본 개시의 일 실시예에 따르면 컴퓨팅 장치(100)를 구현하기 위한 로직은 컴퓨팅 프로그램을 구현하기 위한 수단, 회로 또는 모듈에 의하여 구현될 수도 있다.

[0141] 당업자들은 추가적으로 여기서 개시된 실시예들과 관련되어 설명된 다양한 예시적 논리적 블록들, 구성들, 모듈들, 회로들, 수단들, 로직들 및 알고리즘 단계들이 전자 하드웨어, 컴퓨터 소프트웨어, 또는 양쪽 모두의 조합들로 구현될 수 있음을 인식해야 한다. 하드웨어 및 소프트웨어의 상호교환성을 명백하게 예시하기 위해, 다양한 예시적 컴포넌트들, 블록들, 구성들, 수단들, 로직들, 모듈들, 회로들, 및 단계들은 그들의 기능성 측면에서 일반적으로 위에서 설명되었다. 그러한 기능성이 하드웨어로 또는 소프트웨어로서 구현되는지 여부는 전반적인 시스템에 부과된 특정 어플리케이션(application) 및 설계 제한들에 달려 있다. 숙련된 기술자들은 각각의 특정 어플리케이션들을 위해 다양한 방법들로 설명된 기능성을 구현할 수 있으나, 그러한 구현의 결정들이 본 개시내용의 영역을 벗어나게 하는 것으로 해석되어서는 안된다.

[0143] 도 12는 본 개시의 일 실시예들이 구현될 수 있는 예시적인 컴퓨팅 환경에 대한 간략하고 일반적인 개략도를 도시한다.

[0144] 본 개시가 일반적으로 하나 이상의 컴퓨터 상에서 실행될 수 있는 컴퓨터 실행가능 명령어와 관련하여 전술되었지만, 당업자라면 본 개시가 기타 프로그램 모듈들과 결합되어 및/또는 하드웨어와 소프트웨어의 조합으로 구현될 수 있다는 것을 잘 알 것이다.

[0145] 일반적으로, 프로그램 모듈은 특정의 태스크를 수행하거나 특정의 추상 데이터 유형을 구현하는 루틴, 프로시져, 프로그램, 컴포넌트, 데이터 구조, 기타 등등을 포함한다. 또한, 당업자라면 본 개시의 방법이 단일-프로세서 또는 멀티프로세서 컴퓨터 시스템, 미니컴퓨터, 메인프레임 컴퓨터는 물론 퍼스널 컴퓨터, 핸드헬드 컴퓨팅 장치, 마이크로프로세서-기반 또는 프로그램가능 가전 제품, 기타 등등(이들 각각은 하나 이상의 연관된 장치와 연결되어 동작할 수 있음)을 비롯한 다른 컴퓨터 시스템 구성으로 실시될 수 있다는 것을 잘 알 것이다.

[0146] 본 개시의 설명된 실시예들은 또한 어떤 태스크들이 통신 네트워크를 통해 연결되어 있는 원격 처리 장치들에 의해 수행되는 분산 컴퓨팅 환경에서 실시될 수 있다. 분산 컴퓨팅 환경에서, 프로그램 모듈은 로컬 및 원격 메모리 저장 장치 둘다에 위치할 수 있다.

[0147] 컴퓨터는 통상적으로 다양한 컴퓨터 판독가능 매체를 포함한다. 컴퓨터에 의해 액세스 가능한 매체는 그 어떤 것이든지 컴퓨터 판독가능 매체가 될 수 있고, 컴퓨터 판독가능 매체는 컴퓨터 판독가능 저장 매체 및 컴퓨터 판독가능 전송 매체를 포함할 수 있다. 이러한 컴퓨터 판독가능 저장 매체는 휘발성 및 비휘발성 매체, 이동식

및 비-이동식 매체를 포함한다. 컴퓨터 판독가능 저장 매체는 컴퓨터 판독가능 명령어, 데이터 구조, 프로그램 모듈 또는 기타 데이터와 같은 정보를 저장하는 임의의 방법 또는 기술로 구현되는 휘발성 및 비휘발성 매체, 이동식 및 비이동식 매체를 포함한다. 컴퓨터 판독가능 저장 매체는 RAM, ROM, EEPROM, 플래시 메모리 또는 기타 메모리 기술, CD-ROM, DVD(digital video disk) 또는 기타 광 디스크 저장 장치, 자기 카세트, 자기 테이프, 자기 디스크 저장 장치 또는 기타 자기 저장 장치, 또는 컴퓨터에 의해 액세스될 수 있고 원하는 정보를 저장하는 데 사용될 수 있는 임의의 기타 매체를 포함하지만, 이에 한정되지 않는다.

[0148] 컴퓨터 판독가능 전송 매체는 통상적으로 반송파(carrier wave) 또는 기타 전송 메커니즘(transport mechanism)과 같은 피변조 데이터 신호(modulated data signal)에 컴퓨터 판독가능 명령어, 데이터 구조, 프로그램 모듈 또는 기타 데이터등을 구현하는 정보 전달 매체를 포함한다. 피변조 데이터 신호라는 용어는 신호 내에 정보를 인코딩하도록 그 신호의 특성들 중 하나 이상을 설정 또는 변경시킨 신호를 의미한다. 제한이 아닌 예로서, 컴퓨터 판독가능 전송 매체는 유선 네트워크 또는 직접 배선 접속(direct-wired connection)과 같은 유선 매체, 그리고 음향, RF, 적외선, 기타 무선 매체와 같은 무선 매체를 포함한다. 상술된 매체들 중 임의의 것의 조합도 역시 컴퓨터 판독가능 전송 매체의 범위 안에 포함되는 것으로 한다.

[0149] 컴퓨터(1502)를 포함하는 본 개시의 여러가지 측면들을 구현하는 예시적인 환경(1500)이 나타내어져 있으며, 컴퓨터(1502)는 처리 장치(1504), 시스템 메모리(1506) 및 시스템 버스(1508)를 포함한다. 시스템 버스(1508)는 시스템 메모리(1506)(이에 한정되지 않음)를 비롯한 시스템 컴포넌트들을 처리 장치(1504)에 연결시킨다. 처리 장치(1504)는 다양한 상용 프로세서들 중 임의의 프로세서일 수 있다. 듀얼 프로세서 및 기타 멀티프로세서 아키텍처도 역시 처리 장치(1504)로서 이용될 수 있다.

[0150] 시스템 버스(1508)는 메모리 버스, 주변장치 버스, 및 다양한 상용 버스 아키텍처 중 임의의 것을 사용하는 로컬 버스에 추가적으로 상호 연결될 수 있는 몇 가지 유형의 버스 구조 중 임의의 것일 수 있다. 시스템 메모리(1506)는 판독 전용 메모리(ROM)(1510) 및 랜덤 액세스 메모리(RAM)(1512)를 포함한다. 기본 입/출력 시스템(BIOS)은 ROM, EPROM, EEPROM 등의 비휘발성 메모리(1510)에 저장되며, 이 BIOS는 시동 중과 같은 때에 컴퓨터(1502) 내의 구성요소들 간에 정보를 전송하는 일을 돕는 기본적인 루틴을 포함한다. RAM(1512)은 또한 데이터를 캐싱하기 위한 정적 RAM 등의 고속 RAM을 포함할 수 있다.

[0151] 컴퓨터(1502)는 또한 내장형 하드 디스크 드라이브(HDD)(1514)(예를 들어, EIDE, SATA)-이 내장형 하드 디스크 드라이브(1514)는 또한 적당한 섀시(도시 생략) 내에서 외장형 용도로 구성될 수 있음-, 자기 플로피 디스크 드라이브(FDD)(1516)(예를 들어, 이동식 디스켓(1518)으로부터 판독을 하거나 그에 기록을 하기 위한 것임), 및 광 디스크 드라이브(1520)(예를 들어, CD-ROM 디스크(1522)를 판독하거나 DVD 등의 기타 고용량 광 매체로부터 판독을 하거나 그에 기록을 하기 위한 것임)를 포함한다. 하드 디스크 드라이브(1514), 자기 디스크 드라이브(1516) 및 광 디스크 드라이브(1520)는 각각 하드 디스크 드라이브 인터페이스(1524), 자기 디스크 드라이브 인터페이스(1526) 및 광 드라이브 인터페이스(1528)에 의해 시스템 버스(1508)에 연결될 수 있다. 외장형 드라이브 구현을 위한 인터페이스(1524)는 USB(Universal Serial Bus) 및 IEEE 1394 인터페이스 기술 중 적어도 하나 또는 그 둘다를 포함한다.

[0152] 이들 드라이브 및 그와 연관된 컴퓨터 판독가능 매체는 데이터, 데이터 구조, 컴퓨터 실행가능 명령어, 기타 등등의 비휘발성 저장을 제공한다. 컴퓨터(1502)의 경우, 드라이브 및 매체는 임의의 데이터를 적당한 디지털 형식으로 저장하는 것에 대응한다. 상기에서의 컴퓨터 판독가능 매체에 대한 설명이 HDD, 이동식 자기 디스크, 및 CD 또는 DVD 등의 이동식 광 매체를 언급하고 있지만, 당업자라면 집 드라이브(zip drive), 자기 카세트, 플래쉬 메모리 카드, 카트리지, 기타 등등의 컴퓨터에 의해 판독가능한 다른 유형의 매체도 역시 예시적인 운영 환경에서 사용될 수 있으며 또 임의의 이러한 매체가 본 개시의 방법들을 수행하기 위한 컴퓨터 실행가능 명령어를 포함할 수 있다는 것을 잘 알 것이다.

[0153] 운영 체제(1530), 하나 이상의 애플리케이션 프로그램(1532), 기타 프로그램 모듈(1534) 및 프로그램 데이터(1536)를 비롯한 다수의 프로그램 모듈이 드라이브 및 RAM(1512)에 저장될 수 있다. 운영 체제, 애플리케이션, 모듈 및/또는 데이터의 전부 또는 그 일부분이 또한 RAM(1512)에 캐싱될 수 있다. 본 개시가 여러가지 상업적으로 이용가능한 운영 체제 또는 운영 체제들의 조합에서 구현될 수 있다는 것을 잘 알 것이다.

[0154] 사용자는 하나 이상의 유선/무선 입력 장치, 예를 들어, 키보드(1538) 및 마우스(1540) 등의 포인팅 장치를 통해 컴퓨터(1502)에 명령 및 정보를 입력할 수 있다. 기타 입력 장치(도시 생략)로는 마이크, IR 리모콘, 조이스틱, 게임 패드, 스타일러스 펜, 터치 스크린, 기타 등등이 있을 수 있다. 이들 및 기타 입력 장치가 종종 시스템 버스(1508)에 연결되어 있는 입력 장치 인터페이스(1542)를 통해 처리 장치(1504)에 연결되지만, 병렬 포트,

IEEE 1394 직렬 포트, 게임 포트, USB 포트, IR 인터페이스, 기타 등등의 기타 인터페이스에 의해 연결될 수 있다.

[0155] 모니터(1544) 또는 다른 유형의 디스플레이 장치도 역시 비디오 어댑터(1546) 등의 인터페이스를 통해 시스템 버스(1508)에 연결된다. 모니터(1544)에 부가하여, 컴퓨터는 일반적으로 스피커, 프린터, 기타 등등의 기타 주변 출력 장치(도시 생략)를 포함한다.

[0156] 컴퓨터(1502)는 유선 및/또는 무선 통신을 통한 원격 컴퓨터(들)(1548) 등의 하나 이상의 원격 컴퓨터로의 논리적 연결을 사용하여 네트워크화된 환경에서 동작할 수 있다. 원격 컴퓨터(들)(1548)는 워크스테이션, 서버 컴퓨터, 라우터, 퍼스널 컴퓨터, 휴대용 컴퓨터, 마이크로프로세서-기반 오락 기기, 피어 장치 또는 기타 통상의 네트워크 노드일 수 있으며, 일반적으로 컴퓨터(1502)에 대해 기술된 구성요소들 중 다수 또는 그 전부를 포함하지만, 간략함을 위해, 메모리 저장 장치(1550)만이 도시되어 있다. 도시되어 있는 논리적 연결은 근거리 통신망(LAN)(1552) 및/또는 더 큰 네트워크, 예를 들어, 원거리 통신망(WAN)(1554)에의 유선/무선 연결을 포함한다. 이러한 LAN 및 WAN 네트워킹 환경은 사무실 및 회사에서 일반적인 것이며, 인트라넷 등의 전사적 컴퓨터 네트워크(enterprise-wide computer network)를 용이하게 해주며, 이들 모두는 전세계 컴퓨터 네트워크, 예를 들어, 인터넷에 연결될 수 있다.

[0157] LAN 네트워킹 환경에서 사용될 때, 컴퓨터(1502)는 유선 및/또는 무선 통신 네트워크 인터페이스 또는 어댑터(1556)를 통해 로컬 네트워크(1552)에 연결된다. 어댑터(1556)는 LAN(1552)에의 유선 또는 무선 통신을 용이하게 해줄 수 있으며, 이 LAN(1552)은 또한 무선 어댑터(1556)와 통신하기 위해 그에 설치되어 있는 무선 액세스 포인트를 포함하고 있다. WAN 네트워킹 환경에서 사용될 때, 컴퓨터(1502)는 모뎀(1558)을 포함할 수 있거나, WAN(1554) 상의 통신 서버에 연결되거나, 또는 인터넷을 통하는 등, WAN(1554)을 통해 통신을 설정하는 기타 수단을 갖는다. 내장형 또는 외장형 및 유선 또는 무선 장치일 수 있는 모뎀(1558)은 직렬 포트 인터페이스(1542)를 통해 시스템 버스(1508)에 연결된다. 네트워크화된 환경에서, 컴퓨터(1502)에 대해 설명된 프로그램 모듈들 또는 그의 일부분이 원격 메모리/저장 장치(1550)에 저장될 수 있다. 도시된 네트워크 연결이 예시적인 것이며 컴퓨터들 사이에 통신 링크를 설정하는 기타 수단이 사용될 수 있다는 것을 잘 알 것이다.

[0158] 컴퓨터(1502)는 무선 통신으로 배치되어 동작하는 임의의 무선 장치 또는 개체, 예를 들어, 프린터, 스캐너, 데스크톱 및/또는 휴대용 컴퓨터, PDA(portable data assistant), 통신 위성, 무선 검출가능 태그와 연관된 임의의 장비 또는 장소, 및 전화와 통신을 하는 동작을 한다. 이것은 적어도 Wi-Fi 및 블루투스 무선 기술을 포함한다. 따라서, 통신은 종래의 네트워크에서와 같이 미리 정의된 구조이거나 단순하게 적어도 2개의 장치 사이의 애드혹 통신(ad hoc communication)일 수 있다.

[0159] Wi-Fi(Wireless Fidelity)는 유선 없이도 인터넷 등으로의 연결을 가능하게 해준다. Wi-Fi는 이러한 장치, 예를 들어, 컴퓨터가 실내에서 및 실외에서, 즉 기지국의 통화권 내의 아무 곳에서나 데이터를 전송 및 수신할 수 있게 해주는 셀 전화와 같은 무선 기술이다. Wi-Fi 네트워크는 안전하고 신뢰성있으며 고속인 무선 연결을 제공하기 위해 IEEE 802.11(a,b,g, 기타)이라고 하는 무선 기술을 사용한다. 컴퓨터를 서로에, 인터넷에 및 유선 네트워크(IEEE 802.3 또는 이더넷을 사용함)에 연결시키기 위해 Wi-Fi가 사용될 수 있다. Wi-Fi 네트워크는 비인가 2.4 및 5 GHz 무선 대역에서, 예를 들어, 11Mbps(802.11a) 또는 54 Mbps(802.11b) 데이터 레이트로 동작하거나, 양 대역(듀얼 대역)을 포함하는 제품에서 동작할 수 있다.

[0160] 본 개시의 기술 분야에서 통상의 지식을 가진 자는 여기에 개시된 실시예들과 관련하여 설명된 다양한 예시적인 논리 블록들, 모듈들, 프로세서들, 수단들, 회로들 및 알고리즘 단계들이 전자 하드웨어, (편의를 위해, 여기에서 "소프트웨어"로 지칭되는) 다양한 형태들의 프로그램 또는 설계 코드 또는 이들 모두의 결합에 의해 구현될 수 있다는 것을 이해할 것이다. 하드웨어 및 소프트웨어의 이러한 상호 호환성을 명확하게 설명하기 위해, 다양한 예시적인 컴포넌트들, 블록들, 모듈들, 회로들 및 단계들이 이들의 기능과 관련하여 위에서 일반적으로 설명되었다. 이러한 기능이 하드웨어 또는 소프트웨어로서 구현되는지 여부는 특정한 애플리케이션 및 전체 시스템에 대하여 부과되는 설계 제약들에 따라 좌우된다. 본 개시의 기술 분야에서 통상의 지식을 가진 자는 각각의 특정한 애플리케이션에 대하여 다양한 방식들로 설명된 기능을 구현할 수 있으나, 이러한 구현 결정들은 본 개시의 범위를 벗어나는 것으로 해석되어서는 안 될 것이다.

[0161] 여기서 제시된 다양한 실시예들은 방법, 장치, 또는 표준 프로그래밍 및/또는 엔지니어링 기술을 사용한 제조 물품(article)으로 구현될 수 있다. 용어 "제조 물품"은 임의의 컴퓨터-판독가능 장치로부터 액세스 가능한 컴퓨터 프로그램, 캐리어, 또는 매체(media)를 포함한다. 예를 들어, 컴퓨터-판독가능 매체는 자기 저장 장치(예를 들면, 하드 디스크, 플로피 디스크, 자기 스트립, 등), 광학 디스크(예를 들면, CD, DVD, 등), 스마트 카드,

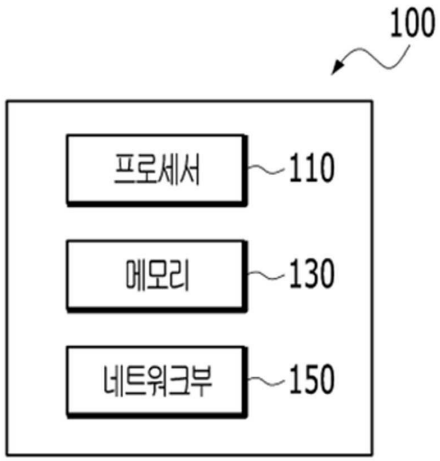
및 플래쉬 메모리 장치(예를 들면, EEPROM, 카드, 스틱, 키 드라이브, 등)를 포함하지만, 이들로 제한되는 것은 아니다. 또한, 여기서 제시되는 다양한 저장 매체는 정보를 저장하기 위한 하나 이상의 장치 및/또는 다른 기계-판독가능한 매체를 포함한다. 용어 "기계-판독가능 매체"는 명령(들) 및/또는 데이터를 저장, 보유, 및/또는 전달할 수 있는 무선 채널 및 다양한 다른 매체를 포함하지만, 이들로 제한되는 것은 아니다.

[0162] 제시된 프로세스들에 있는 단계들의 특정한 순서 또는 계층 구조는 예시적인 접근들의 일례임을 이해하도록 한다. 설계 우선순위들에 기반하여, 본 개시의 범위 내에서 프로세스들에 있는 단계들의 특정한 순서 또는 계층 구조가 재배열될 수 있다는 것을 이해하도록 한다. 첨부된 방법 청구항들은 샘플 순서로 다양한 단계들의 엘리먼트들을 제공하지만 제시된 특정한 순서 또는 계층 구조에 한정되는 것을 의미하지는 않는다.

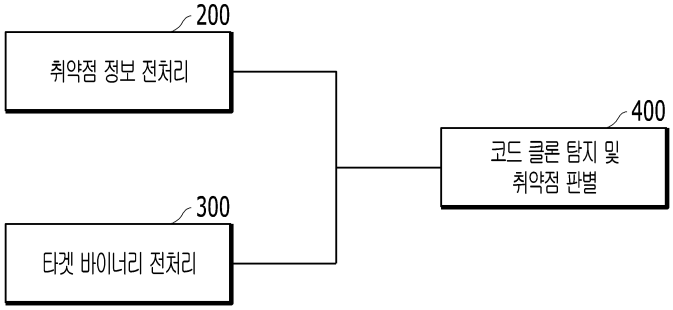
[0163] 제시된 실시예들에 대한 설명은 임의의 본 개시의 기술 분야에서 통상의 지식을 가진 자가 본 개시를 이용하거나 또는 실시할 수 있도록 제공된다. 이러한 실시예들에 대한 다양한 변형들은 본 개시의 기술 분야에서 통상의 지식을 가진 자에게 명백할 것이며, 여기에 정의된 일반적인 원리들은 본 개시의 범위를 벗어남이 없이 다른 실시예들에 적용될 수 있다. 그리하여, 본 개시는 여기에 제시된 실시예들로 한정되는 것이 아니라, 여기에 제시된 원리들 및 신규한 특징들과 일관되는 최광의의 범위에서 해석되어야 할 것이다.

도면

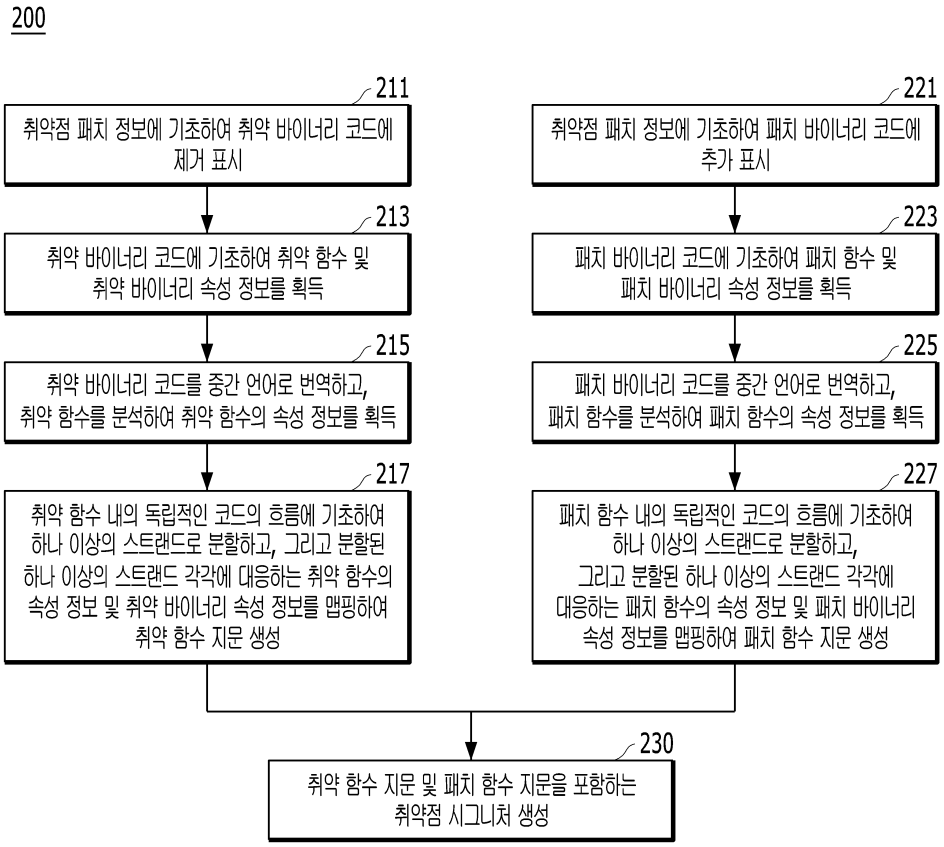
도면1



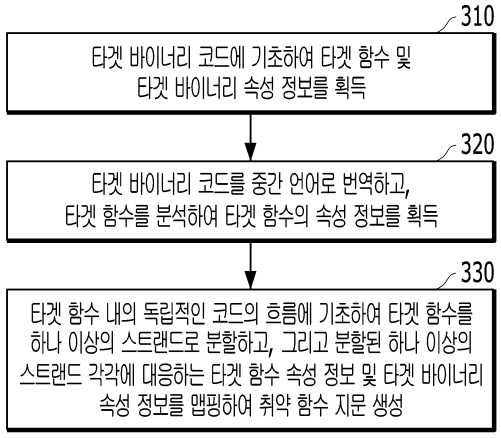
도면2



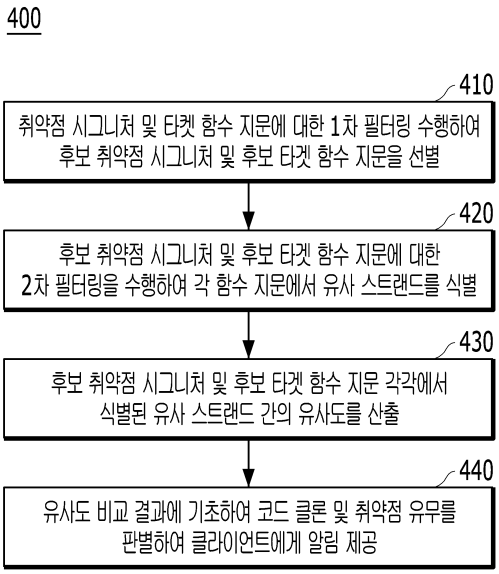
도면3


200
211
취약점 패치 정보에 기초하여 취약 바이너리 코드에 제거 표시
213
취약 바이너리 코드에 기초하여 취약 함수 및 취약 바이너리 속성 정보를 획득
215
취약 바이너리 코드를 중간 언어로 번역하고, 취약 함수를 분석하여 취약 함수의 속성 정보를 획득
217
취약 함수 내의 독립적인 코드의 흐름에 기초하여 하나 이상의 스트랜드로 분할하고, 그리고 분할된 하나 이상의 스트랜드 각각에 대응하는 취약 함수의 속성 정보 및 취약 바이너리 속성 정보를 맵핑하여 취약 함수 지문 생성
221
취약점 패치 정보에 기초하여 패치 바이너리 코드에 추가 표시
223
패치 바이너리 코드에 기초하여 패치 함수 및 패치 바이너리 속성 정보를 획득
225
패치 바이너리 코드를 중간 언어로 번역하고, 패치 함수를 분석하여 패치 함수의 속성 정보를 획득
227
패치 함수 내의 독립적인 코드의 흐름에 기초하여 하나 이상의 스트랜드로 분할하고, 그리고 분할된 하나 이상의 스트랜드 각각에 대응하는 패치 함수의 속성 정보 및 패치 바이너리 속성 정보를 맵핑하여 패치 함수 지문 생성
230
취약 함수 지문 및 패치 함수 지문을 포함하는 취약점 시그니처 생성

도면4


300
310
타겟 바이너리 코드에 기초하여 타겟 함수 및 타겟 바이너리 속성 정보를 획득
320
타겟 바이너리 코드를 중간 언어로 번역하고, 타겟 함수를 분석하여 타겟 함수의 속성 정보를 획득
330
타겟 함수 내의 독립적인 코드의 흐름에 기초하여 타겟 함수를 하나 이상의 스트랜드로 분할하고, 그리고 분할된 하나 이상의 스트랜드 각각에 대응하는 타겟 함수 속성 정보 및 타겟 바이너리 속성 정보를 맵핑하여 취약 함수 지문 생성

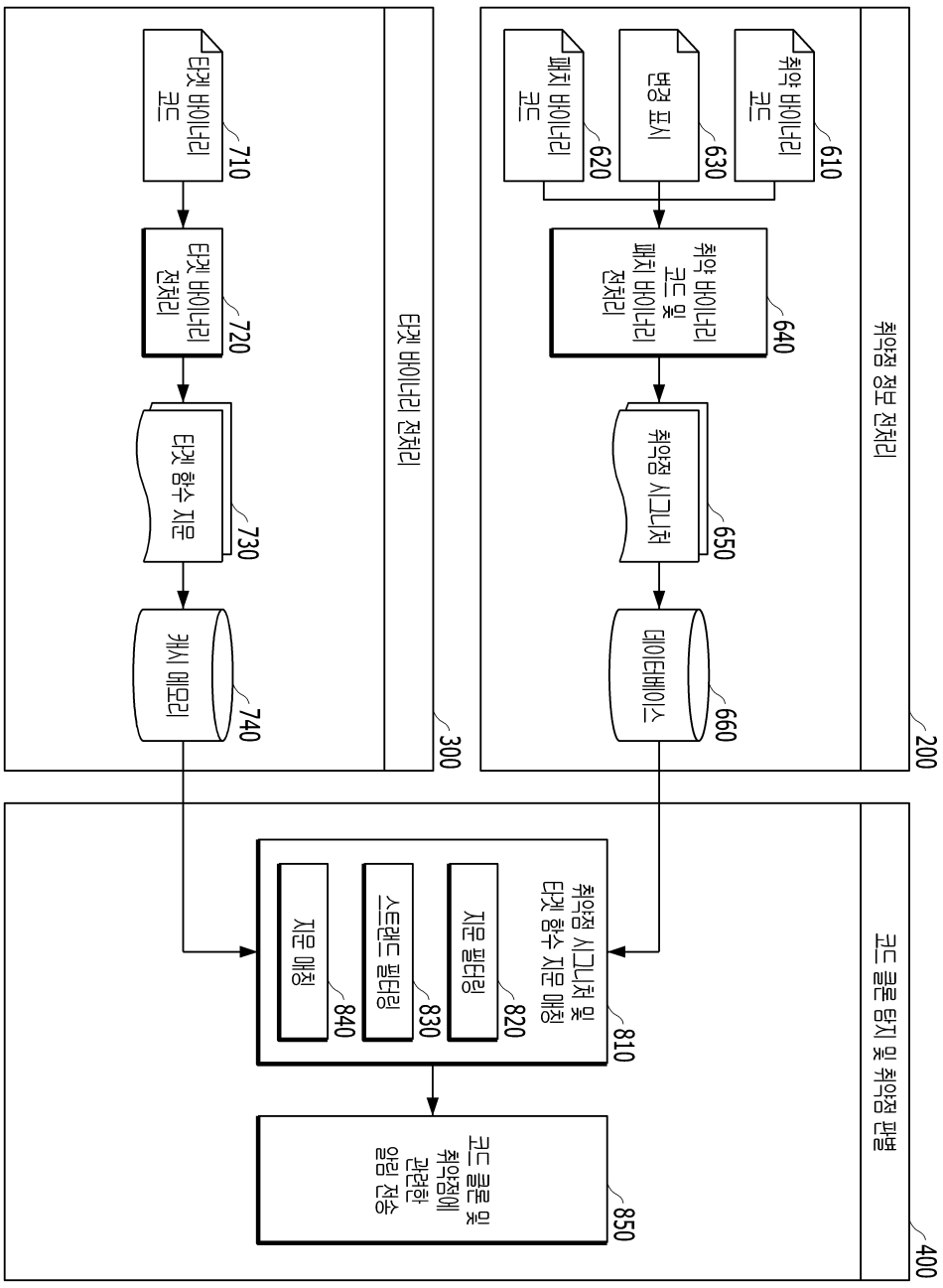
도면5



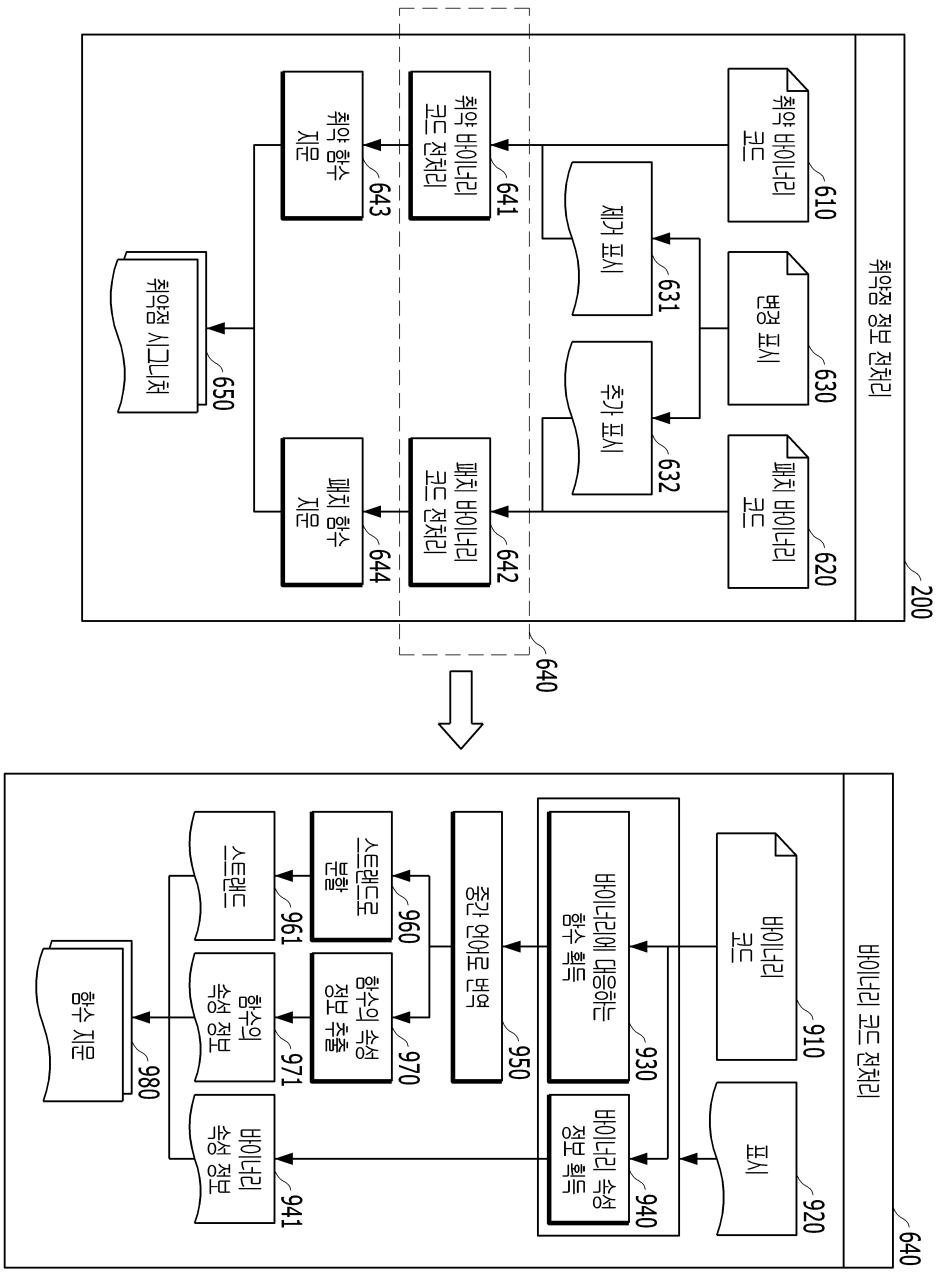
도면6

Seq_X = ABCDE	Seq_Y = ABCZE
G_X = ngram(Seq_X) = { AB, ABC, BCD, CDE, DE }	G_Y = ngram(Seq_Y) = { AB, ABC, BCZ, CZE, ZE }
n-gram similarity (n = 3) $= \frac{N(G_X \cap G_Y)}{N(G_X \cup G_Y)} = \frac{2}{8} = 0.25$	

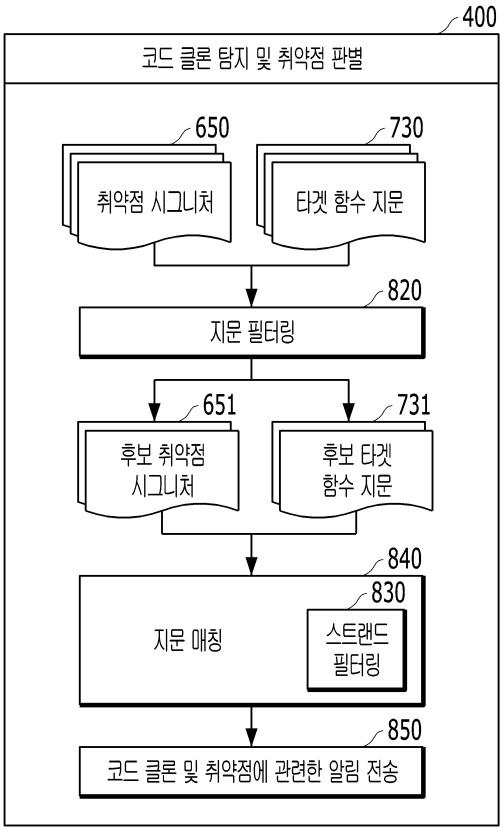
도면7


취약점 정보 전처리
200
취약 바이너리 코드
610
변경 표시
630
패치 바이너리 코드
620
취약 바이너리 코드 및 패치 바이너리 전처리
640
취약점 시그니쳐
650
데이터베이스
660
타겟 바이너리 전처리
300
타겟 바이너리 코드
710
타겟 바이너리 전처리
720
타겟 함수 지문
730
캐시 메모리
740
코드 클론 탐지 및 취약점 판별
400
취약점 시그니쳐 및 타겟 함수 지문 매칭
810
지문 필터링
820
스트랜드 필터링
830
지문 매칭
840
코드 클론 및 취약점에 관련한 알림 전송
850

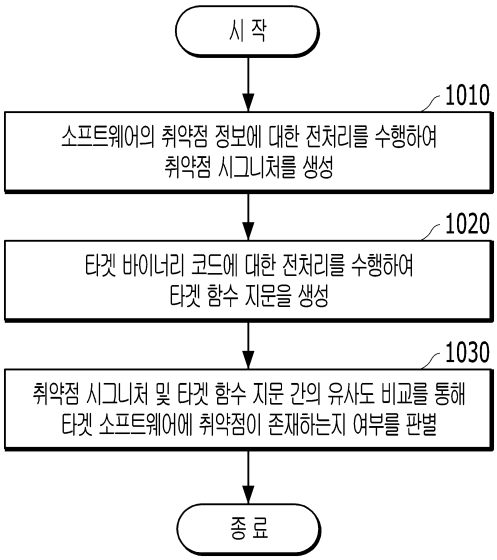
도면8


취약점 정보 전처리
200
취약 바이너리 코드
610
변경 표시
630
패치 바이너리 코드
620
제거 표시
631
추가 표시
632
취약 바이너리 코드 전처리
641
패치 바이너리 코드 전처리
642
640
취약 함수 지문
643
패치 함수 지문
644
취약점 시그니쳐
650
바이너리 코드 전처리
640
바이너리 코드
910
표시
920
바이너리에 대응하는 함수 획득
930
바이너리 속성 정보 획득
940
중간 언어로 번역
950
함수의 속성 정보 추출
970
스트랜드로 분할
960
스트랜드
961
함수의 속성 정보
971
바이너리 속성 정보
941
함수 지문
980

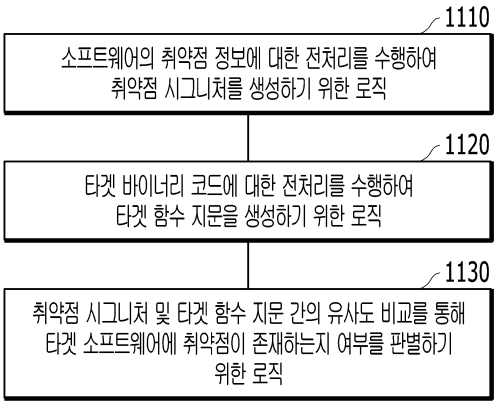
도면9


400
코드 클론 탐지 및 취약점 판별
650
취약점 시그니처
730
타겟 함수 지문
820
지문 필터링
651
후보 취약점 시그니처
731
후보 타겟 함수 지문
840
830
지문 매칭
스트랜드 필터링
850
코드 클론 및 취약점에 관련한 알림 전송

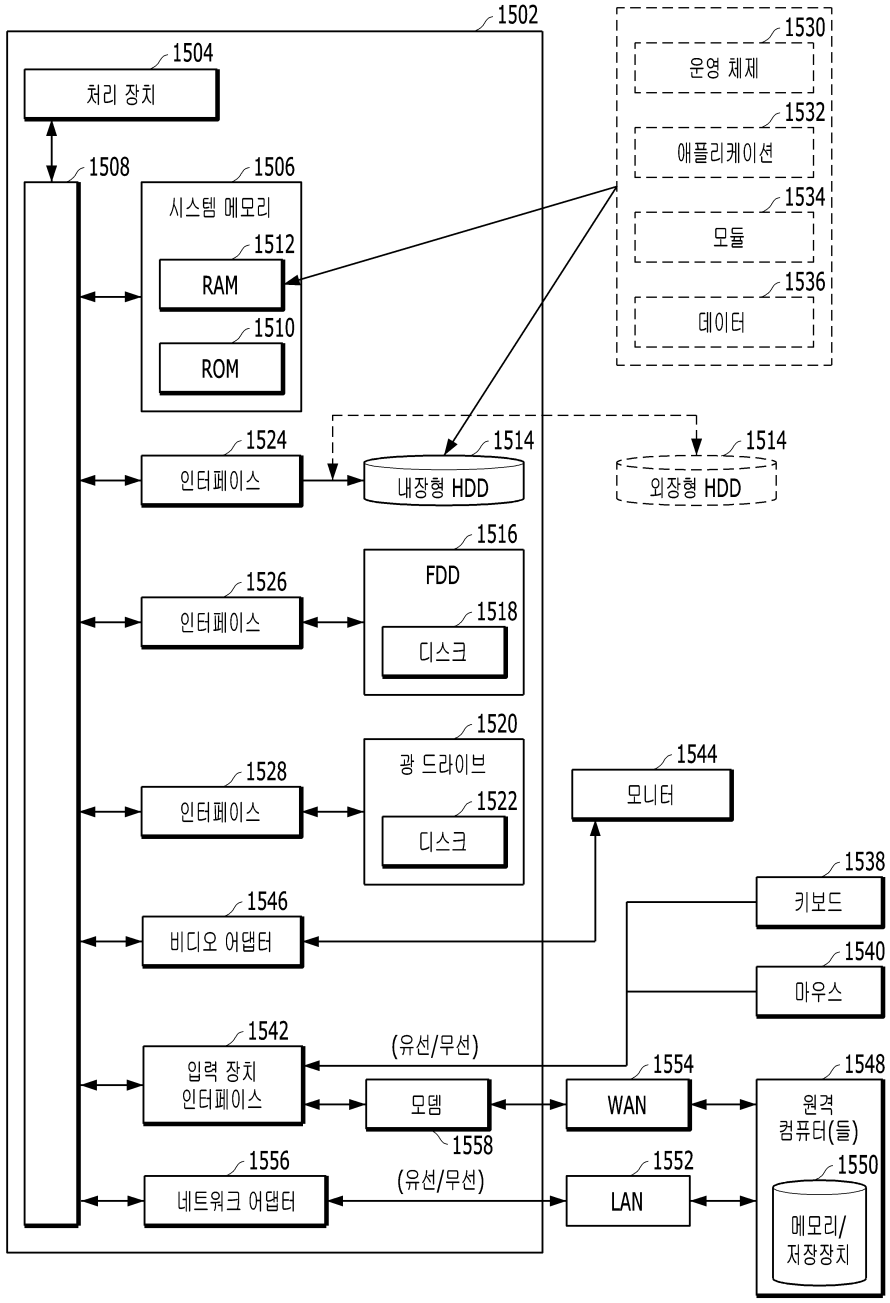
도면10


시 작
1010
소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성
1020
타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성
1030
취약점 시그니처 및 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별
종 료

도면11


1110
소프트웨어의 취약점 정보에 대한 전처리를 수행하여 취약점 시그니처를 생성하기 위한 로직
1120
타겟 바이너리 코드에 대한 전처리를 수행하여 타겟 함수 지문을 생성하기 위한 로직
1130
취약점 시그니처 및 타겟 함수 지문 간의 유사도 비교를 통해 타겟 소프트웨어에 취약점이 존재하는지 여부를 판별하기 위한 로직

도면12


1502
1504
처리 장치
1508
1506
시스템 메모리
1512
RAM
1510
ROM
1530
운영 체제
1532
애플리케이션
1534
모듈
1536
데이터
1524
인터페이스
1514
내장형 HDD
1514
외장형 HDD
1516
FDD
1526
인터페이스
1518
디스크
1520
광 드라이브
1528
인터페이스
1522
디스크
1544
모니터
1538
키보드
1546
비디오 어댑터
1540
마우스
1542
입력 장치 인터페이스
(유선/무선)
1554
WAN
1548
원격 컴퓨터(들)
모뎀
1558
1556
네트워크 어댑터
(유선/무선)
1552
LAN
1550
메모리/ 저장장치