(19) 대한민국특허청(KR)
(12) 공개특허공보(A)

(11) 공개번호 10-2024-0162785
(43) 공개일자 2024년11월18일

(51) 국제특허분류(Int. Cl.)
G06F 11/36 (2006.01) *G06F 8/41* (2018.01)
G06F 8/51 (2018.01)
(52) CPC특허분류
G06F 11/3604 (2013.01)
G06F 11/3664 (2013.01)
(21) 출원번호 10-2023-0059810
(22) 출원일자 2023년05월09일
심사청구일자 2023년05월09일

(71) 출원인
한국과학기술원
대전광역시 유성구 대학로 291(구성동)
충남대학교산학협력단
대전광역시 유성구 대학로 99 (궁동, 충남대학교)
경북대학교 산학협력단
대구광역시 북구 대학로 80, 경북대학교내(산격동)
(72) 발명자
류석영
대전광역시 유성구 대학로 291 (구성동, 한국과학기술원)
박지희
대전광역시 유성구 대학로 291 (구성동, 한국과학기술원)
(뒷면에 계속)
(74) 대리인
양성보

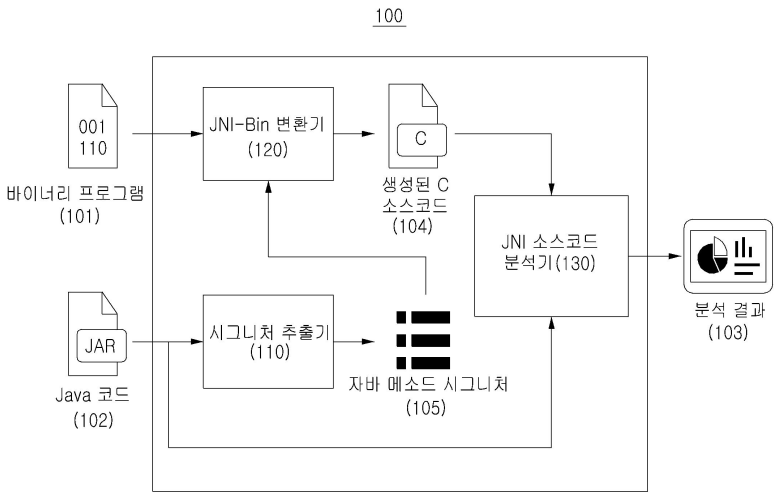
전체 청구항 수 : 총 10 항

(54) 발명의 명칭 **디컴파일러를 활용한 native JNI 프로그램의 정적 분석 시스템 및 방법**

(57) 요 약

디컴파일러를 활용한 native JNI 프로그램의 정적 분석 시스템 및 방법이 개시된다. 일 실시예에 따른 정적 분석 시스템은, Java 코드를 이용하여 바이너리 프로그램과 상호작용하는 자바 메소드 시그니처를 추출하는 시그니처 추출기; 상기 추출된 자바 메소드 시그니처를 이용하여 상기 바이너리 프로그램으로부터 C 소스코드를 생성하는 JNI-Bin 변환기; 및 상기 생성된 C 소스코드와 상기 Java 코드를 통해 JNI 프로그램의 정적 분석을 수행하는 JNI 소스코드 분석기를 포함할 수 있다.

대 표 도



(52) CPC특허분류
G06F 11/3688 (2013.01)
G06F 11/3692 (2013.01)
G06F 8/43 (2013.01)
G06F 8/447 (2013.01)
G06F 8/51 (2013.01)
(72) 발명자
홍재민
대전광역시 유성구 대학로 291 (구성동, 한국과학기술원)
이성호
대전광역시 유성구 노은동로 111, 1006동 302호 (노은동, 열매마을 10단지아파트)

이 발명을 지원한 국가연구개발사업
과제고유번호 1711110922
과제번호 2017R1A2B301202015
부처명 과학기술정보통신부
과제관리(전문)기관명 한국연구재단
연구사업명 이공분야기초연구사업
연구과제명 (N01200252)(통합EZ)SW를 진단하는 분석기를 사용자 요구에 맞춰 자동으로 합성하는 프레임워크 개발(2020년도)
기 여 율 5/100
과제수행기관명 한국과학기술원
연구기간 2020.03.01 ~ 2021.02.28
이 발명을 지원한 국가연구개발사업
과제고유번호 1711116307
과제번호 2017M3C4A706817723
부처명 과학기술정보통신부
과제관리(전문)기관명 한국연구재단
연구사업명 원천기술개발사업
연구과제명 (통합EZ)지능형 자동화를 통한 풀스택 SW 코드 검증(2020)
기 여 율 5/100
과제수행기관명 한국과학기술원
연구기간 2020.04.01 ~ 2021.03.31
이 발명을 지원한 국가연구개발사업
과제고유번호 1711156478
과제번호 2022R1A2C200366011
부처명 과학기술정보통신부
과제관리(전문)기관명 한국연구재단
연구사업명 이공분야기초연구사업
연구과제명 (N01220686)(통합EZ)다각화된 프로그래밍 언어로 작성된 SW를 위한 다양한 분석 도구를 자동으로 생성하는 프레임워크 개발(2022년도)
기 여 율 50/100
과제수행기관명 한국과학기술원
연구기간 2022.03.01 ~ 2023.02.28
이 발명을 지원한 국가연구개발사업
과제고유번호 1711137717
과제번호 2021R1A5A1021944
부처명 과학기술정보통신부
과제관리(전문)기관명 한국연구재단
연구사업명 이공분야기초연구사업
연구과제명 소프트웨어재난 연구센터
기 여 율 20/100
과제수행기관명 경북대학교
연구기간 2021.06.01 ~ 2022.02.28
이 발명을 지원한 국가연구개발사업
과제고유번호 1711170655
과제번호 2022-0-00460
부처명 과학기술정보통신부
과제관리(전문)기관명 정보통신기획평가원
연구사업명 정보통신.방송 기술개발사업
연구과제명 (N01220854)(통합EZ)고성능 동형암호 최적화 컴파일러 개발(2022년도)
기 여 율 20/100
과제수행기관명 한국전자통신연구원
연구기간 2022.04.01 ~ 2022.12.31

명 세 서

청구범위

청구항 1

정적 분석 시스템에 있어서,

Java 코드를 이용하여 바이너리 프로그램과 상호작용하는 자바 메소드 시그니처를 추출하는 시그니처 추출기;

상기 추출된 자바 메소드 시그니처를 이용하여 상기 바이너리 프로그램으로부터 C 소스코드를 생성하는 JNI-Bin 변환기; 및

상기 생성된 C 소스코드와 상기 Java 코드를 통해 JNI 프로그램의 정적 분석을 수행하는 JNI 소스코드 분석기

를 포함하는 정적 분석 시스템.

청구항 2

제1항에 있어서,

상기 시그니처 추출기는,

상기 바이너리 프로그램과 상호작용하는 Java 메소드의 이름, 메소드 인자와 메소드 실행 결과의 타입을 추출하는

것을 특징으로 하는 정적 분석 시스템.

청구항 3

제1항에 있어서,

상기 JNI-Bin 변환기는,

디스어셈블러에서 상기 바이너리 프로그램을 구문 분석하여 함수 객체와 데이터 객체를 포함하는 객체 정보를 생성하는

것을 특징으로 하는 정적 분석 시스템.

청구항 4

제3항에 있어서,

상기 함수 객체는, 상기 바이너리 프로그램에서 함수 이름, 함수 시그니처, 함수 본문의 주소 범위를 포함하는 함수 정보를 포함하고,

상기 데이터 객체는, 레지스터, 변수 또는 메모리 주소에 저장된 데이터 정보를 포함하는

것을 특징으로 하는 정적 분석 시스템.

청구항 5

제3항에 있어서,

상기 JNI-Bin 변환기는,

중간코드 변환기에서 자바 메소드 시그니처를 이용하여 상기 생성된 함수 객체와 데이터 객체를 포함하는 객체 정보를 수정하여 JNI에 특화된 객체 정보를 생성하는

것을 특징으로 하는 정적 분석 시스템.

청구항 6

제5항에 있어서,

상기 JNI-Bin 변환기는,

상기 중간코드 변환기에서 상기 생성된 JNI에 특화된 객체 정보를 이용하여 중간코드를 생성하는

것을 특징으로 하는 정적 분석 시스템.

청구항 7

제6항에 있어서,

상기 JNI-Bin 변환기는,

소스코드 생성기에서 상기 생성된 중간코드로부터 컴파일 가능 여부를 판단하여 컴파일이 가능한 C 소스코드를 생성하는

것을 특징으로 하는 정적 분석 시스템.

청구항 8

제6항에 있어서,

상기 JNI-Bin 변환기는,

상기 중간코드 변환기에서 상기 생성된 중간코드에 대한 데이터 흐름 분석을 수행한 결과에 기초하여 업데이트된 함수 객체와 데이터 객체를 이용하여 중간코드를 재생성하는

것을 특징으로 하는 정적 분석 시스템.

청구항 9

정적 분석 시스템에 의해 수행되는 정적 분석 방법을 실행시키기 위해 컴퓨터 판독 가능한 저장매체에 저장된 컴퓨터 프로그램에 있어서,

상기 정적 분석 방법은,

Java 코드를 이용하여 바이너리 프로그램과 상호작용하는 자바 메소드 시그니처를 추출하는 단계;

상기 추출된 자바 메소드 시그니처를 이용하여 상기 바이너리 프로그램으로부터 C 소스코드를 생성하는 단계; 및

상기 생성된 C 소스코드와 상기 Java 코드를 통해 JNI 프로그램의 정적 분석을 수행하는 단계

를 포함하는 것을 특징으로 하는 컴퓨터 판독 가능한 저장매체에 저장된 컴퓨터 프로그램.

청구항 10

정적 분석 시스템에 의해 수행되는 정적 분석 방법에 있어서,

Java 코드를 이용하여 바이너리 프로그램과 상호작용하는 자바 메소드 시그니처를 추출하는 단계;

상기 추출된 자바 메소드 시그니처를 이용하여 상기 바이너리 프로그램으로부터 C 소스코드를 생성하는 단계; 및

상기 생성된 C 소스코드와 상기 Java 코드를 통해 JNI 프로그램의 정적 분석을 수행하는 단계

를 포함하는 정적 분석 방법.

발명의 설명

기 술 분 야

[0001] 아래의 설명은 정적 분석 기술에 관한 것이다.

배 경 기 술

[0003] 소프트웨어의 결함을 검증하는 정적 분석 기술은 프로그램을 실행하기 전에 미리 생길 수 있는 문제를 진단함으로써 소프트웨어의 안전성과 보안성을 향상시킨다. 미국 국립표준기술연구소(NIST)의 발표에 따르면 제품 출시 이후에 결함을 해결하는데 드는 비용이 개발 초기단계에서 결함을 제거할 때보다 최대 30배의 비용이 더 요구된다고 발표하였다. 정적 분석 기술을 이용하면 가능한 모든 실행 결과를 미리 예측하여 결함을 미연에 방지할 수 있기 때문에 신뢰성이 중요한 프로그램 개발에 중요하게 사용된다.

[0004] 바이너리 프로그램이란 소스코드를 컴파일하여 만들어진 기계어로 작성된 프로그램을 뜻한다. 바이너리 프로그램은 사람이 읽을 수 없는 기계어의 나열로 이루어져 있으며, 컴파일한 소스코드의 의미를 그대로 실행하게 된다. 이렇게 사람이 읽기 어렵고 정적 분석을 수행하기도 어려운 바이너리 프로그램을 다시 소스코드 형태로 되돌려주는 기술을 바이너리 디컴파일이라 하고, 디컴파일을 수행하는 도구를 디컴파일러라고 한다.

[0005] JNI(Java Native Interface)란 Java 프로그램과 다른 언어로 작성되어 컴파일된 프로그램이 상호작용하며 실행하기 위한 규약으로, Java 프로그램 실행 중 다른 언어로 작성된 라이브러리를 호출하거나 반대로 다른 언어로 작성된 라이브러리에서 Java 함수를 호출하려고 할 때 그 호출 방식과 나와야 할 결과를 정해 둔 인터페이스이다. 이때 한 언어로 작성된 프로그램을 실행할 때와는 다른 종류의 취약점이 발생할 수 있다. Java 언어를 기반으로 코드를 작성하는 Android 앱의 경우에 특정 알고리즘을 빠르게 실행하기 위해 일부 코드를 다른 언어로 작성하여 JNI를 통해 통신하는 경우가 있다. JNI로 작성된 프로그램 중 라이브러리의 소스코드가 없고 기계어로 작성된 바이너리 프로그램과 통신하는 JNI 프로그램을 native JNI 프로그램이라고 한다.

[0006] 현재 JNI를 사용하여 만들어진 Java와 C 언어로 작성된 프로그램을 정적 분석할 때에는 C로 작성된 프로그램의 소스코드가 필요하다는 문제가 있다. JNI로 만들어진 많은 프로그램들은 소스코드 없이 컴파일된 바이너리 프로그램 형태로 제공된다. 한정된 목적에 한하여 native JNI 프로그램을 분석하는 기술은 존재하지만, 소스코드 분석기가 할 수 있는 임의의 분석을 native JNI 프로그램에 대해 적용하는 기술은 현재 존재하지 않는다.

발명의 내용

해결하려는 과제

[0008] 디컴파일러를 사용하여 소스코드없이 컴파일된 JNI 프로그램을 정적으로 분석하는 방법 및 시스템을 제공할 수 있다.

[0009] 바이너리 프로그램을 디컴파일함에 따라 생성된 C 소스코드에 대해 JNI 프로그램 분석기를 사용하여 JNI 프로그램을 분석하는 방법 및 시스템을 제공할 수 있다.

과제의 해결 수단

[0011] 정적 분석 시스템은, Java 코드를 이용하여 바이너리 프로그램과 상호작용하는 자바 메소드 시그니처를 추출하는 시그니처 추출기; 상기 추출된 자바 메소드 시그니처를 이용하여 상기 바이너리 프로그램으로부터 C 소스코드를 생성하는 JNI-Bin 변환기; 및 상기 생성된 C 소스코드와 상기 Java 코드를 통해 JNI 프로그램의 정적 분석을 수행하는 JNI 소스코드 분석기를 포함할 수 있다.

[0012] 상기 시그니처 추출기는, 상기 바이너리 프로그램과 상호작용하는 Java 메소드의 이름, 메소드 인자와 메소드 실행 결과의 타입을 추출할 수 있다.

[0013] 상기 JNI-Bin 변환기는, 디스어셈블러에서 상기 바이너리 프로그램을 구문 분석하여 함수 객체와 데이터 객체를 포함하는 객체 정보를 생성할 수 있다.

[0014] 상기 함수 객체는, 상기 바이너리 프로그램에서 함수 이름, 함수 시그니처, 함수 본문의 주소 범위를 포함하는 함수 정보를 포함하고, 상기 데이터 객체는, 레지스터, 변수 또는 메모리 주소에 저장된 데이터 정보를 포함할 수 있다.

[0015] 상기 JNI-Bin 변환기는, 중간코드 변환기에서 자바 메소드 시그니처를 이용하여 상기 생성된 함수 객체와 데이

터 객체를 포함하는 객체 정보를 수정하여 JNI에 특화된 객체 정보를 생성할 수 있다.

[0016] 상기 JNI-Bin 변환기는, 상기 중간코드 변환기에서 상기 생성된 JNI에 특화된 객체 정보를 이용하여 중간코드를 생성할 수 있다.

[0017] 상기 JNI-Bin 변환기는, 소스코드 생성기에서 상기 생성된 중간코드로부터 컴파일 가능 여부를 판단하여 컴파일이 가능한 C 소스코드를 생성할 수 있다.

[0018] 상기 JNI-Bin 변환기는, 상기 중간코드 변환기에서 상기 생성된 중간코드에 대한 데이터 흐름 분석을 수행한 결과에 기초하여 업데이트된 함수 객체와 데이터 객체를 이용하여 중간코드를 재생성할 수 있다.

[0019] 정적 분석 시스템에 의해 수행되는 정적 분석 방법을 실행시키기 위해 컴퓨터 판독 가능한 저장매체에 저장된 컴퓨터 프로그램에 있어서, 상기 정적 분석 방법은, Java 코드를 이용하여 바이너리 프로그램과 상호작용하는 자바 메소드 시그니처를 추출하는 단계; 상기 추출된 자바 메소드 시그니처를 이용하여 상기 바이너리 프로그램으로부터 C 소스코드를 생성하는 단계; 및 상기 생성된 C 소스코드와 상기 Java 코드를 통해 JNI 프로그램의 정적 분석을 수행하는 단계를 포함할 수 있다.

[0020] 정적 분석 시스템에 의해 수행되는 정적 분석 방법은, Java 코드를 이용하여 바이너리 프로그램과 상호작용하는 자바 메소드 시그니처를 추출하는 단계; 상기 추출된 자바 메소드 시그니처를 이용하여 상기 바이너리 프로그램으로부터 C 소스코드를 생성하는 단계; 및 상기 생성된 C 소스코드와 상기 Java 코드를 통해 JNI 프로그램의 정적 분석을 수행하는 단계를 포함할 수 있다.

발명의 효과

[0022] 안드로이드의 많은 앱들이 JNI를 사용하여 개발되는데, 모든 앱이 소스코드를 같이 제공하는 것이 아니기 때문에 소스코드가 없는 JNI 프로그램이 어떤 행위를 하는지 실행 전에 탐지하기 어렵다. 실시예에 따르면, 안드로이드 앱을 비롯한 native JNI를 사용하는 프로그램에 대해 소스코드 분석 수준으로 원하는 결함을 쉽게 검출할 수 있게 하여 소프트웨어의 품질 개선 및 결함 방지에 큰 도움이 될 것으로 예상한다.

도면의 간단한 설명

[0024] 도 1은 일 실시예에 있어서, native JNI 프로그램의 정적 분석 동작을 설명하기 위한 도면이다.

도 2는 일 실시예에 있어서, JNI-Bin 변환기를 설명하기 위한 도면이다.

도 3은 일 실시예에 있어서, 바이너리로 정의된 함수의 재구성 예를 나타낸 도면이다.

도 4는 일 실시예에 있어서, JNIEnv* 타입의 전파 예를 나타낸 도면이다.

도 5는 일 실시예에 있어서, 고정 아리티 시그니처가 있는 JNI 인터페이스 함수의 복구 예를 나타낸 도면이다.

도 6은 일 실시예에 있어서, 가변 시그니처가 있는 JNI 인터페이스 함수의 복구 예를 나타낸 도면이다.

도 7은 일 실시예에 있어서, thunk 함수 인식의 예를 나타낸 도면이다

도 8 및 도 9는 일 실시예에 있어서, 호출 가능 함수를 분석한 결과를 나타낸 도면이다.

도 10은 일 실시예에 있어서, 데이터 흐름 분석 결과를 나타낸 도면이다.

발명을 실시하기 위한 구체적인 내용

[0025] 이하, 실시예를 첨부한 도면을 참조하여 상세히 설명한다.

[0027] 도 1은 일 실시예에 있어서, native JNI 프로그램의 정적 분석 동작을 설명하기 위한 도면이다.

[0028] 정적 분석 시스템(100)은 바이너리 디컴파일을 이용하여 소스코드 없이 JNI 프로그램을 정적을 분석할 수 있다. 정적 분석 시스템(100)은 시그니처 추출기(110), JNI-Bin 변환기(120), 및 JNI 소스 코드 분석기(130)로 구성될

수 있다.

[0029] 정적 분석 시스템(100)은 컴파일된 JNI 프로그램을 바이너리 프로그램(101)과 Java 코드(102)로 분할할 수 있다. 정적 분석 시스템(100)은 바이너리 프로그램(101)과 Java 코드(102)로 이루어진 native JNI 프로그램을 입력받아 정적 분석을 수행하여 분석 결과(103)를 출력할 수 있다.

[0030] 정적 분석 시스템(100)은 바이너리 프로그램(101)과 상호작용하는 Java 코드의 자바 메소드 시그니처(105)를 추출할 수 있다. 정적 분석 시스템(100)은 시그니처 추출기(110)를 통해 Java 코드(102)를 이용하여 바이너리 프로그램(101)과 상호작용하는 Java 메소드의 이름, 메소드 인자와 메소드 실행 결과의 타입을 포함하는 자바 메소드 시그니처(105)를 추출할 수 있다.

[0031] 정적 분석 시스템(100)은 JNI-Bin 변환기(120)를 통해 바이너리 프로그램(101)과 추출된 자바 메소드 시그니처(105) 정보를 활용하여 생성된 C 소스코드(104)를 획득할 수 있다. 정적 분석 시스템(100)은 상호작용 가능한 디컴파일러에 자바 메소드 시그니처(105) 정보를 활용하기 위하여 도 2와 같은 구조가 발명되었다. JNI-Bin 변환기(120)는 바이너리 프로그램(101)과 Java 코드(102)의 자바 메소드 시그니처(105)를 이용하여 바이너리 프로그램(101)로부터 C 소스코드(104)를 생성할 수 있다.

[0032] 정적 분석 시스템(100)은 JNI 소스코드 분석기(130)를 통해 JNI-Bin 변환기(120)에서 생성된 C 소스코드(104)를 이용하여 native JNI 프로그램의 정적 분석을 수행할 수 있다. JNI 소스코드 분석기(130)는 JNI-Bin 변환기(120)에서 생성된 C 소스코드(104)와 Java 코드(102)를 입력으로 사용하여 JNI 프로그램에 대한 분석 결과(103)를 생성할 수 있다.

[0033] 도 2는 일 실시예에 있어서, JNI-Bin 변환기를 설명하기 위한 도면이다.

[0034] JNI-Bin 변환기(120)의 상호작용 가능한 디컴파일러(200)는 디스어셈블러(210), 중간코드 변환기(220) 및 소스코드 생성기(230)로 구성될 수 있다.

[0035] 디컴파일러(200)는 첫 번째 단계인 디스어셈블러(210)가 실행된 후 생성된 데이터 객체 및 함수 객체 정보(201)를 자바 메소드 시그니처(105)를 이용한 수정을 통해 JNI에 특화된 객체 정보(예를 들면, 바이너리 프로그램(101)에 정의된 함수의 올바른 시그니처)(202)를 생성할 수 있다. 디컴파일러(200)는 JNI에 특화된 객체 정보(202)를 이용하여 중간코드를 생성하고, 생성된 중간코드(203)가 실제 분석이 가능한 코드인지 확인할 수 있다. 디컴파일러(200)는 생성된 중간코드가 실제 분석이 불가능한 경우, 분석이 불가능한 원인을 제공하는 함수를 디컴파일 범위에서 제외한 뒤 중간코드를 재생성할 수 있다.

[0036] 보다 상세하게는, 디스어셈블러(210)는 바이너리 프로그램(101)을 구문 분석하고 데이터 객체 및 함수 객체라는 두 종류의 객체 정보를 생성할 수 있다. 함수 객체는 바이너리에서 함수 이름, 함수 시그니처 및 함수 본문의 주소 범위를 포함하는 함수 정보를 포함한다. 데이터 객체는 레지스터, 변수 또는 메모리 주소에 저장된 데이터 정보를 포함한다. 대부분의 데이터 객체에는 데이터에 대한 타입 정보만 있지만 일부 데이터 객체에는 메모리 주소에 저장된 상수 값이 포함된다. 함수 시그니처와 데이터 타입이 바이너리 파일에 없기 때문에 디스어셈블러(210)는 정적 분석과 경험적 접근법(heuristics)을 사용하여 데이터 객체 및 함수 객체를 추론할 수 있다. 객체는 중간코드 변환기(220) 및 소스 코드 생성기(230)의 후속 구성 요소에서 사용될 수 있다. 이때, 디스어셈블러(210)를 통해 사용자가 기존 객체를 검사, 수정 및 삭제하고 새로운 객체를 생성할 수 있다.

[0037] 중간코드 변환기(220)는 JNI에 특화된 객체 정보(202)로써 어셈블리 명령을 중간코드로 변환하여 각 함수 객체를 디컴파일러(200)의 중간코드에 기록된 함수로 변환할 수 있다. 대부분의 중간코드는 C 소스코드와 유사하다. 그러나 중간코드 변환기(220)는 어셈블리 명령어와 동일한 의미 체계를 나타내는 명령문이 없을 때 어셈블리 명령을 높은 수준의 C 소스코드와 유사한 문으로 변환하지 못하는 경우가 있다. 그러한 어셈블리 명령은 그대로 유지된다. 이러한 이유로 중간코드는 C 소스코드와 구문적으로 유사하지만 본문에 어셈블리 명령을 포함할 수 있다. 어셈블리에서 중간코드로의 변환은 함수 객체 및 데이터 객체를 참조한다. 일련의 어셈블리 명령을 중간코드로 변환할 때 중간코드 변환기(220)는 명령어에서 액세스하는 레지스터, 변수, 메모리 주소의 데이터 객체와 호출 명령(어)의 피호출자의 함수 객체를 가져온다. 중간코드 변환기(220)는 객체 정보를 사용하여 명령을 가장 적합한 중간코드로 변환할 수 있다. 예를 들어, x에 대한 데이터 객체에 타입이 int일 때 중간코드 변환기(220)가 일부 어셈블리 명령을 *(int*)(x + 4) 문으로 변환한다고 가정하기로 한다. 타입을 int*로 변경하면 중간코드 변환기(220)는 변경된 타입에 따라 동일한 어셈블리 명령을 x[1]로 변환할 수 있다.

[0038] 소스 코드 생성기(230)는 생성된 중간코드(203)를 유사 C 소스코드로 변환할 수 있다. C 소스코드는 중간코드 변환기(220)에 의해 생성된 중간코드(203)가 종종 C의 타이핑 규칙을 무시하기 때문에 타입 오류가 있는 문장이

있을 수 있다. 예를 들어, 디스어셈블러(210)에 의해 추론된 잘못된 함수 시그니처로 인해 잘못된 수의 인수를 가진 함수 호출이 포함될 수 있다. 또한, 유사 C 소스코드는 중간코드의 어셈블리 명령으로부터 변환된 정의되지 않은 매크로 호출들을 포함할 수 있다. 이러한 이유로 유사 C 소스코드는 컴파일할 수 없다.

[0039] 디컴파일러(200)는 바이너리로 정의된 함수의 올바른 시그니처를 재구성할 수 있다. 바이너리에 함수 파라미터 유형을 올바르게 추론하기에 충분한 힌트를 가지고 있지 않기 때문에 디스어셈블러는 함수 시그니처를 올바르게 재구성하지 못하는 경우가 많다. 디스어셈블러가 파라미터 유형을 추론할 수 없는 경우, 파라미터는 기본적으로 유형 int를 가지고 있다고 가정하고 잘못된 함수 시그니처를 포함하는 잘못된 함수 객체를 생성한다. 파라미터 유형은 로컬 변수 유형을 유추하기 위한 힌트이기 때문에 함수 시그니처가 잘못되면 로컬 변수의 데이터 객체도 잘못될 수 있다. 잘못된 함수 객체 및 데이터 객체로 인해 중간코드 변환기(220)에서 잘못된 중간코드가 생성된다.

[0040] 디스어셈블러(210)의 부정확성을 완화하기 위해 디스어셈블러(210)에서 생성된 함수 객체 정보를 수정하여 바이너리 프로그램(101)에 정의된 함수의 올바른 시그니처를 복구할 수 있다. 바이너리 파일에는 충분한 유형 정보가 없지만 모든 시그니처는 Java 코드(102)에 남아 있다. JNI-Bin 변환기(120)는 시그니처 추출기(110)에 의해 Java 코드(102)에서 추출된 자바 메소드 시그니처(105)를 활용할 수 있다. 바이너리 프로그램(101)으로 정의된 각 함수에 대해 JNI-Bin 변환기(120)는 먼저 시그니처 추출기(110)에서 추출된 자바 메소드 시그니처(105) 중 native 함수에 연결된 Java 메소드를 찾을 수 있다. JNI-Bin 변환기(120)는 JNI 명명 규칙을 따라 연결된 Java 메소드의 이름을 결정할 수 있다. 그런 다음, JNI-Bin 변환기(120)는 Java 메소드의 시그니처에서 native 함수의 올바른 시그니처를 계산할 수 있다. 이때, JNI 사양에 따라 각 파라미터 타입 또는 반환 타입을 해당 C 타입으로 변환하고 파라미터 타입 앞에 JNIEnv* 및 jobject 타입이 추가될 수 있다. 마지막으로, JNI-Bin 변환기(120)는 함수 객체의 잘못된 시그니처를 올바른 시그니처로 교체할 수 있다. 교체 후 디스어셈블러(210)는 각 native 함수의 파라미터 및 로컬 변수에 대한 데이터 객체를 다시 계산하고 중간코드 변환기(220)는 올바른 중간코드를 생성할 수 있다.

[0041] 도 3을 참고하면, 바이너리 프로그램(101)으로 정의된 함수의 재구성 예를 나타낸 도면이다. - 및 +로 레이블이 지정된 줄(Line)은 각각 원본의 디컴파일 결과와 재구성 결과를 나타낸다. native 함수 Java_get_1column_1name에 해당하는 Java 메소드는 native String get_column_name(long,int)이며 native 함수의 올바른 시그니처는 (JNIEnv*, jobject, jlong, jint)이다. 그러나 2행에 표시된 것처럼 디컴파일러는 native 함수에 대해 잘못된 시그니처(int, int, int, int, int)을 생성한다. native 함수의 개수도 잘못 추론된다. 디스어셈블러(210)는 32비트 아키텍처에서 jlong 타입의 크기가 8바이트이고 int 타입의 크기가 4바이트이기 때문에 하나의 jlong 타입 파라미터를 두 개의 int 타입 파라미터로 잘못 해석한다. 6행의 JNI 인터페이스 함수 호출도 잘못 컴파일 된다. NewStringUTF의 함수 포인터는 JNIEnv 포인터에서 오프셋 167에 배치된다. 그러나 디컴파일러(200)는 디스어셈블러(210)가 파라미터 a1에 잘못된 타입 int를 제공하므로 함수 포인터 대신, 산술식 (*(_DWORD *)a1 + 668)을 생성한다. 디컴파일러(200)는 Java 메소드의 시그니처를 사용하여 native 함수의 시그니처를 재구성한 후 올바른 파라미터 유형을 사용하여 3행에 표시된 올바른 native 함수 시그니처와 8행과 9행에 표시된 올바른 JNI 함수 호출을 생성한다.

[0042] JNI-Bin 변환기(120)는 생성된 중간코드의 품질을 향상시키기 위해 5가지 변환 단계를 통해 중간코드를 조작할 수 있다. 이중 3가지 변환 단계는 중간코드에 정확한 타입 정보를 추가한다. JNI 관련 타입을 정확하게 만드는 것이 목적이기 때문에 이러한 단계를 JNI 관련 변환이라고 부르기로 한다. JNI 프로그램에서만 사용할 수 있는 타입 정보를 활용하므로 이 작업의 주요 기여이다. JNI-Bin 변환기(120)는 각각의 JNI 관련 변환에 대해 중간코드에 대한 데이터 흐름 분석을 수행하고 분석 결과로 함수 객체와 데이터 객체를 업데이트하여 코드를 수정할 수 있다. 나머지 다른 두 단계를 JNI와 무관한 변환이라고 부른다.

[0043] 우선적으로, JNI 관련 변환에 대하여 설명하기로 한다. JNI 프로그램에서 native 함수는 종종 JNIEnv 포인터를 다른 함수로 전달한다. 그러나 중간코드 변환기(220)는 JNIEnv 포인터의 유형인 JNIEnv*를 제대로 전파하지 않으며 잘못된 중간코드를 생성한다. 이때, 일부 native 함수에 대한 올바른 시그니처를 재구성한 후에도 다른 함수 시그니처는 변경되지 않고 int타입 파라미터를 사용하여 변경되지 않은 상태로 유지된다. 따라서 인수와 파라미터 간의 타입 호환성을 위해 중간코드 변환기(220)는 포인터를 인수로 전달하기 전에 JNIEnv*를 int로 변환하는 부적절한 캐스트를 삽입할 수 있다.

[0044] JNI-Bin 변환기(120)는 중간코드를 수정하여 JNIEnv* 타입을 올바르게 전파할 수 있다. JNI-Bin 변환기(120)는 먼저 절차 내 데이터 흐름 분석을 수행하여 JNIEnv 포인터를 추적할 수 있다. 데이터 흐름 정보를 통해 JNI-

Bin 변환기(120)는 세 가지 종류의 중간코드를 처리할 수 있다. 첫째, JNIEnv 포인터의 캐스팅의 경우, 실제로 그러한 캐스팅이 드물기 때문에 단순히 캐스팅을 제거한다. 둘째, JNIEnv 포인터를 변수에 저장하는 할당의 경우 변수에 대한 데이터 객체의 유형을 JNIEnv*로 변경한다. 셋째, JNIEnv 포인터를 인수로 전달하는 함수 호출의 경우, 호출 수신자에 대한 함수 객체의 파라미터 타입을 JNIEnv로 변경한다. 수정에서 나온 타입 힌트를 감안할 때 디컴파일러(200)는 JNIEnv 타입과 JNI 인터페이스 함수의 호출 사이트를 정확하게 추론할 수 있다. JNI-Bin 변환기(120)는 절차 간 타입 추론 결과를 획득하기 위해 JNIEnv*가 더 이상 전파되지 않을 때까지 호출 수신자 호출 함수를 통해 재귀적으로 수정을 수행할 수 있다.

[0045] 도 4를 참고하면, JNIEnv* 타입의 전파 예를 나타낸 도면이다. 중간코드 변환기(220)에서 처음 생성한 중간코드에서 create_graphics의 첫 번째 파라미터는 10행에 잘못된 타입 int를 가지고 있으며, 4행에는 JNIEnv 포인터의 캐스트를 가지고 있다. 또한, 14, 15행에서는 파라미터 a1의 잘못된 유형으로 인해 JNI 인터페이스 함수 호출이 식별되지 않는다. JNI-Bin 변환기(120)는 캐스트를 제거하고 a1의 유형을 JNIEnv*로 변경한다. 그런 다음, 중간코드 변환기(220)는 FindClass 함수 호출을 올바르게 식별한다.

[0046] 디스어셈블러(210)가 바이너리에서 호출된 JNI 인터페이스 함수의 시그니처를 잘못 추론하면 중간코드 변환기(220)가 잘못된 함수 호출문을 생성할 수 있다. 이러한 잘못된 함수 호출문은 잘못된 수의 인수, 잘못된 타입의 인수 또는 잘못된 반환 값의 부적절한 캐스트를 가지고 있다.

[0047] JNI 인터페이스 함수의 시그니처를 사용하여 JNI 인터페이스 함수 호출을 올바르게 재구성할 수 있다. JNI 사양은 각 JNI 인터페이스 함수의 시그니처를 설명한다. 대부분의 JNI 인터페이스 함수는 고정된 아리티 시그니처(Fixed Arity Signatures)를 가지고 있기 때문에 추가적인 노력 없이도 시그니처를 사용할 수 있다. JNI-Bin 변환기(120)는 절차 내 데이터 흐름 분석을 수행하여 JNIEnv 포인터를 추적하고 JNI 인터페이스 함수의 호출을 식별할 수 있다. JNI-Bin 변환기(120)는 식별된 각 호출에 대해, 대상 JNI 인터페이스 함수의 시그니처의 함수 포인터에 대한 데이터 객체의 유형을 변경할 수 있다. 디스어셈블러(210)는 변경된 데이터 객체를 사용하여 호출과 관련된 데이터 객체를 업데이트하고 중간코드 변환기(220)는 올바른 함수 호출문을 생성할 수 있다.

[0048] 도 5를 참고하면, 고정 아리티 시그니처가 있는 JNI 인터페이스 함수의 복구 예를 나타낸 도면이다. 중간코드 변환기(220)의 초기 결과에서 8행의 JNI 인터페이스 함수 호출은 단독 인수를 갖는 반면, 함수 호출 수신자 NewStringUTF는 두 가지 인수를 가진다. 또한, 7행은 반환 값을 jstring으로 캐스팅하는데, 이는 NewStringUTF의 반환 타입이 이미 jstring이기 때문에 불필요하다. JNI-Bin 변환기(120)는 NewStringUTF의 함수 포인터에 대한 데이터 객체를 알려진 시그니처 jString NewStringUTF(JNIEnv*, const char*)로 변경할 수 있다. 수정 후, 중간코드 변환기(220)는 올바른 함수 호출문을 생성할 수 있다.

[0049] 시그니처에 정확한 수의 파라미터와 유형이 포함되어 있지 않기 때문에 앞서 언급한 접근 방식을 사용하여 가변 시그니처가 있는 JNI 인터페이스 함수의 잘못 추론된 시그니처를 수정하기 위해 사용할 수 없다. 이에, 모든 가변 JNI 인터페이스 함수는 C에서 Java 메소드를 호출한다. 각 함수는 적어도 세 개의 인수를 필요로 한다. 첫 번째 인수는 JNIEnv 포인터이고, 두 번째 인수는 메소드 호출의 수신자 객체이며, 세 번째 인수는 대상 메소드를 식별하는 jmethodID이다. 메소드에 자체 인수가 있는 경우, JNI 인터페이스 함수 호출은 이를 추가 인수로 받아들인다.

[0050] 호출 사이트에서 대상 Java 방법의 시그니처로 각 JNI 인터페이스 함수 호출의 시그니처를 추론할 수 있다. 세 번째 인수인 jmethodID는 호출의 대상 Java 메소드를 식별하며, 추가 인수의 수와 타입이 대상 메소드의 시그니처와 일치한다. 호출 사이트에서 대상 메소드를 결정하기 위해 JNI-Bin 변환기(120)는 역방향 절차 내 데이터 흐름 분석을 수행하여 호출 사이트에서 생성까지 jmethodID를 추적할 수 있다. 이때, 각 jmethodID는 GetMethodID, GetStaticMethodID, FromReflectedMethod와 같은 JNI 인터페이스 함수 중 하나에 의해 생성될 수 있다. 이 중 Java 리플렉션을 포함하는 FromReflectedMethod는 무시한다. GetMethodID, GetStaticMethodID는 Java 클래스 정보, 메소드 이름 및 메소드 시그니처를 사용하며 이름과 시그니처는 종종 문자열 리터럴(string literals)로 제공될 수 있다. 따라서 JNI-Bin 변환기(120)가 jmethodID의 생성에 도달하면, 종종 문자열 리터럴에서 대상 메소드의 시그니처를 가져올 수 있다. 그런 다음, 최종적으로 JNI 인터페이스 함수 호출의 시그니처를 결정하고 함수 포인터에 대한 데이터 객체의 유형을 업데이트 할 수 있다.

[0051] 도 6을 참고하면, 가변 시그니처가 있는 JNI 인터페이스 함수의 복구 예를 나타낸 도면이다. 14행은 v17과 a4라는 두 개의 인수로 Java 메소드를 호출하는 반면, 대상 메소드는 단일 문자열 인수를 사용한다. JNI-Bin 변환기(120)는 11행의 GetMethod에 대한 세 번째 인수 v11을 추적하며 두 문자열 상수 "set_param"과 "(Ljava/lang/String;)I"에서 메소드 시그니처를 획득할 수 있다. 시그니처를 통해 CallIntMethod에서 요구하

는 실제 인수 개수와 인수 유형을 유추할 수 있다. JNI 인터페이스 함수 포인터의 데이터 객체를 업데이트한 후 중간코드 변환기(220)는 15행에 표시된 것처럼 올바른 함수 호출을 생성할 수 있다.

[0052] 다음으로, JNI와 무관한 변환에 대하여 설명하기로 한다. 변환 후에도 중간코드 변환기(220)에 다른 결함이 존재하며, 정확한 C 소스코드를 획득하는 것을 방해한다. 결함을 처리하기 위해 두 가지 변환을 더 제안할 수 있다. 이러한 변환은 JNI 고유하지 않으며 바이너리 프로그램의 디컴파일에 적용할 수 있다.

[0053] 바이너리 프로그램(101)은 외부 함수를 동적으로 연결하는 썽크(thunk) 함수를 포함한다. 링커는 링크 시간에 동적으로 연결된 함수의 주소를 확인할 수 없다. 동적 링크를 허용하기 위해 컴파일러는 바이너리 코드 내에 프로시저 링크 테이블(PLT; Procedure Linkage Table))과 글로벌 오프셋 테이블(GOT; Global Offset Table)을 기록한다. PLT의 각 항목을 thunk 함수라고 한다. 각 thunk 함수는 동적 링커를 대상으로 하는 아키텍처별 점프 명령어로 구성될 수 있다. GOT는 외부 함수에 대한 포인터를 저장한다. 그러나 런타임에 동적 링크가 발생하기 전에 외부 함수의 주소를 결정할 수 없기 때문에 GOT는 초기에 PLT 항목에 대한 포인터를 갖는다. 외부 함수에 대한 모든 호출은 GOT의 해당 함수 포인터에 대한 호출로 컴파일될 수 있다. 런타임에서, 특정 외부 함수에 대한 첫 번째 호출은 PLT에서 해당하는 thunk 함수를 호출하게 된다. thunk 함수는 동적 링커를 호출하고, 링커는 외부 함수에 대한 올바른 포인터로 GOT를 덮어쓰고 외부 함수를 호출하여 원래 호출을 해결한다.

[0054] 중간코드 변환기(220)는 가변 인수가 있는 외부 함수가 있으면 잘못된 코드를 생성할 수 있다. 이러한 함수에 대한 호출은 가변 인수를 사용하여 thunk 함수에 대한 호출로 컴파일 될 수 있다. 각각의 thunk 함수는 하나의 점프 명령어로만 구성되어 있기 때문에, 주어진 인수를 동적 링커에 직접 전달하고, 인수들은 결국 대상 외부 함수에 전달된다. 안타깝게도 중간코드는 모든 변수 인수를 다른 함수로 전달하는 것을 지원하지 않는다. 이러한 이유로 thunk 함수의 재구성된 본문에는 인수가 누락된 호출이 포함되어 있다.

[0055] 이러한 문제를 해결하기 위해 JNI-Bin 변환기(120)는 생성된 중간코드(203)를 직접 수정할 수 있다. 생성된 중간코드(203)에서 thunk 함수 호출을 모두 인식한다. 그런 다음 호출 수신자를 수정하여 각 thunk 함수 호출을 외부 함수 호출로 대체한다. 이러한 변환은 thunk 함수를 호출하지 않도록 하므로 thunk 함수 본문의 부정확성이 추가 분석을 오염시키는 것을 방지한다.

[0056] 도 7을 참고하면, thunk 함수 인식의 예를 나타낸 도면이다. log는 변수 인수가 있는 외부 함수이며, j_log는 해당하는 thunk 함수이다. j_log가 모든 인수를 로그에 전달하는 동안 중간코드는 변수 인수를 표현할 수 없다. 결과적으로 생성된 중간코드의 j_log는 두 개의 인수만 사용하여 호출 로그를 기록한다. JNI-Bin 변환기(120)는 j_log에 대한 호출을 인식하고 로그에 대한 호출로 대체하여 문제를 해결할 수 있다.

[0057] 중간코드 변환기(220)는 변환된 C 코드를 컴파일할 수 없게 만드는 알 수 없는 유형 이름을 가진 중간코드를 생성할 수 있다. 중간코드 변환기(220)는 잘 알려진 C/C++ 라이브러리에 정의된 thread_struct_t 및 AndroidBitmapInfo를 포함한 일련의 타입을 사용하여 높은 수준의 추상화를 재구성할 수 있다. 재구성 프로세스에서 구조가 알려진 타입에 해당하는 값을 찾으면 중간코드 변환기(220)는 타입을 사용하는 중간코드를 생성한다. 그러나 타입 정보는 중간코드 변환기(220)의 내부에 상주하며 중간코드의 타입 정의의 형식으로 사용자에게 직접 노출되지는 않는다. 따라서 중간코드에서 정의하지 않고 사용한다는 점에서 알 수 없는 유형 이름이다. 중간코드가 C소스코드로 변환될 때, 결과 C 소스코드에도 알 수 없는 타입 이름도 포함되어 컴파일 오류를 발생시킨다.

[0058] JNI-Bin 변환기(120)는 알 수 없는 유형 이름을 void*로 대체하여 제거하고 컴파일 오류를 방지할 수 있다. 대체 솔루션은 중간코드 변환기(220)에서 타입 정보를 추출하고, 추출된 타입 정보를 사용하여 코드에 타입 정의를 추가하는 것이지만, void*의 도입으로 인한 정밀도 손실이 상호 연동 분석 결과에 영향을 미치지 않는다.

[0059] JNI-Bin 변환기(120)는 소스코드 생성기(230)를 사용하여 중간코드에서 C소스코드를 생성할 수 있다. JNI-Bin 변환기(120)는 중간코드의 다양한 결함을 휴리스틱으로 해결하지만, 중간코드에서 생성된 유사 C 소스코드는 여전히 컴파일이 불가능할 수 있다. 지금까지 논의된 디컴파일의 개선은 JNI 상호 연동에 초점을 맞추고 있기 때문에, JNI 상호 연동과 무관한 중간코드는 부정확하게 남아있을 수 있다.

[0060] JNI-Bin 변환기(120)는 IR 함수에서 생성된 각 C 함수의 컴파일 가능성을 개별적으로 확인하고 컴파일이 불가능한 C 함수를 제거할 수 있다. JNI-Bin 변환기(120)는 Clang 컴파일러의C 함수의 "컴파일 전용" 옵션을 사용하여 각 C 함수를 컴파일하여 C 함수의 컴파일 가능 여부를 결정할 수 있다. 이러한 제거는 C 코드가 원래 바이너리 코드의 일부 동작을 잃게 만들 수 있지만, 나머지 C 함수는 일반적으로 JNI 소스 코드 분석기가 JNI 프로

그램의 상호 연동 의미 체계를 분석하기에 충분하다.

[0061] 도 8 및 도 9는 일 실시예에 있어서, 호출 가능 함수를 분석한 결과를 나타낸 도면이다.

[0062] 도 8 및 도 9는 소스코드가 있는 안드로이드 앱에 대해 기존의 소스 코드 분석기를 이용하여 어떤 함수가 호출 가능한지 분석한 결과와 같은 앱의 소스코드를 제거하고 실시예에서 제안된 정적 분석 시스템을 통해 정적 분석을 수행한 결과를 나타낸 것이다. 실험을 위해 만들어낸 앱과 실제 앱 마켓에서 다운로드 받을 수 있는 앱에 대해 모두 실험이 진행될 수 있다. 실험 결과 소스코드가 없을 때도 실시예에서 제안된 JNI 소스코드 분석기를 사용하면 종래의 소스 코드 분석기의 결과를 대부분 재현할 수 있었다.

[0063] 도 10은 일 실시예에 있어서, 데이터 흐름 분석 결과를 나타낸 도면이다.

[0064] 도 10은 기존에 존재하는 native JNI의 데이터 흐름 분석기와 실시예에서 제안된 정적 분석 시스템을 데이터 흐름 분석에 사용했을 때의 데이터 흐름 분석 결과를 비교한 것이다. 실시예에서 제안된 정적 분석 시스템은 데이터 흐름 결함 14개를 정확하게 분석한 반면, 기존의 native JNI 데이터 흐름 분석기는 데이터 흐름 결함 8개를 정확하게 분석하고 1개 프로그램에 대해서는 시간 초과로 분석에 실패하였다. 또한 기존의 native JNI 데이터 흐름 분석기는 데이터 흐름 분석 외에 다른 분석을 수행할 수 없지만 실시예에서는 임의의 소스코드 분석을 수행할 수 있다.

[0065] 이상에서 설명된 장치는 하드웨어 구성요소, 소프트웨어 구성요소, 및/또는 하드웨어 구성요소 및 소프트웨어 구성요소의 조합으로 구현될 수 있다. 예를 들어, 실시예들에서 설명된 장치 및 구성요소는, 예를 들어, 프로세서, 콘트롤러, ALU(arithmetic logic unit), 디지털 신호 프로세서(digital signal processor), 마이크로컴퓨터, FPGA(field programmable gate array), PLU(programmable logic unit), 마이크로프로세서, 또는 명령(instruction)을 실행하고 응답할 수 있는 다른 어떠한 장치와 같이, 하나 이상의 범용 컴퓨터 또는 특수 목적 컴퓨터를 이용하여 구현될 수 있다. 처리 장치는 운영 체제(OS) 및 상기 운영 체제 상에서 수행되는 하나 이상의 소프트웨어 애플리케이션을 수행할 수 있다. 또한, 처리 장치는 소프트웨어의 실행에 응답하여, 데이터를 접근, 저장, 조작, 처리 및 생성할 수도 있다. 이해의 편의를 위하여, 처리 장치는 하나가 사용되는 것으로 설명된 경우도 있지만, 해당 기술분야에서 통상의 지식을 가진 자는, 처리 장치가 복수 개의 처리 요소(processing element) 및/또는 복수 유형의 처리 요소를 포함할 수 있음을 알 수 있다. 예를 들어, 처리 장치는 복수 개의 프로세서 또는 하나의 프로세서 및 하나의 콘트롤러를 포함할 수 있다. 또한, 병렬 프로세서(parallel processor)와 같은, 다른 처리 구성(processing configuration)도 가능하다.

[0066] 소프트웨어는 컴퓨터 프로그램(computer program), 코드(code), 명령(instruction), 또는 이들 중 하나 이상의 조합을 포함할 수 있으며, 원하는 대로 동작하도록 처리 장치를 구성하거나 독립적으로 또는 결합적으로(collectively) 처리 장치를 명령할 수 있다. 소프트웨어 및/또는 데이터는, 처리 장치에 의하여 해석되거나 처리 장치에 명령 또는 데이터를 제공하기 위하여, 어떤 유형의 기계, 구성요소(component), 물리적 장치, 가상 장치(virtual equipment), 컴퓨터 저장 매체 또는 장치에 구체화(embody)될 수 있다. 소프트웨어는 네트워크로 연결된 컴퓨터 시스템 상에 분산되어서, 분산된 방법으로 저장되거나 실행될 수도 있다. 소프트웨어 및 데이터는 하나 이상의 컴퓨터 판독 가능 기록 매체에 저장될 수 있다.

[0067] 실시예에 따른 방법은 다양한 컴퓨터 수단을 통하여 수행될 수 있는 프로그램 명령 형태로 구현되어 컴퓨터 판독 가능 매체에 기록될 수 있다. 상기 컴퓨터 판독 가능 매체는 프로그램 명령, 데이터 파일, 데이터 구조 등을 단독으로 또는 조합하여 포함할 수 있다. 상기 매체에 기록되는 프로그램 명령은 실시예를 위하여 특별히 설계되고 구성된 것들이거나 컴퓨터 소프트웨어 당업자에게 공지되어 사용 가능한 것일 수도 있다. 컴퓨터 판독 가능 기록 매체의 예에는 하드 디스크, 플로피 디스크 및 자기 테이프와 같은 자기 매체(magnetic media), CD-ROM, DVD와 같은 광기록 매체(optical media), 플롭티컬 디스크(floptical disk)와 같은 자기-광 매체(magneto-optical media), 및 롬(ROM), 램(RAM), 플래시 메모리 등과 같은 프로그램 명령을 저장하고 수행하도록 특별히 구성된 하드웨어 장치가 포함된다. 프로그램 명령의 예에는 컴파일러에 의해 만들어지는 것과 같은 기계어 코드뿐만 아니라 인터프리터 등을 사용해서 컴퓨터에 의해서 실행될 수 있는 고급 언어 코드를 포함한다.

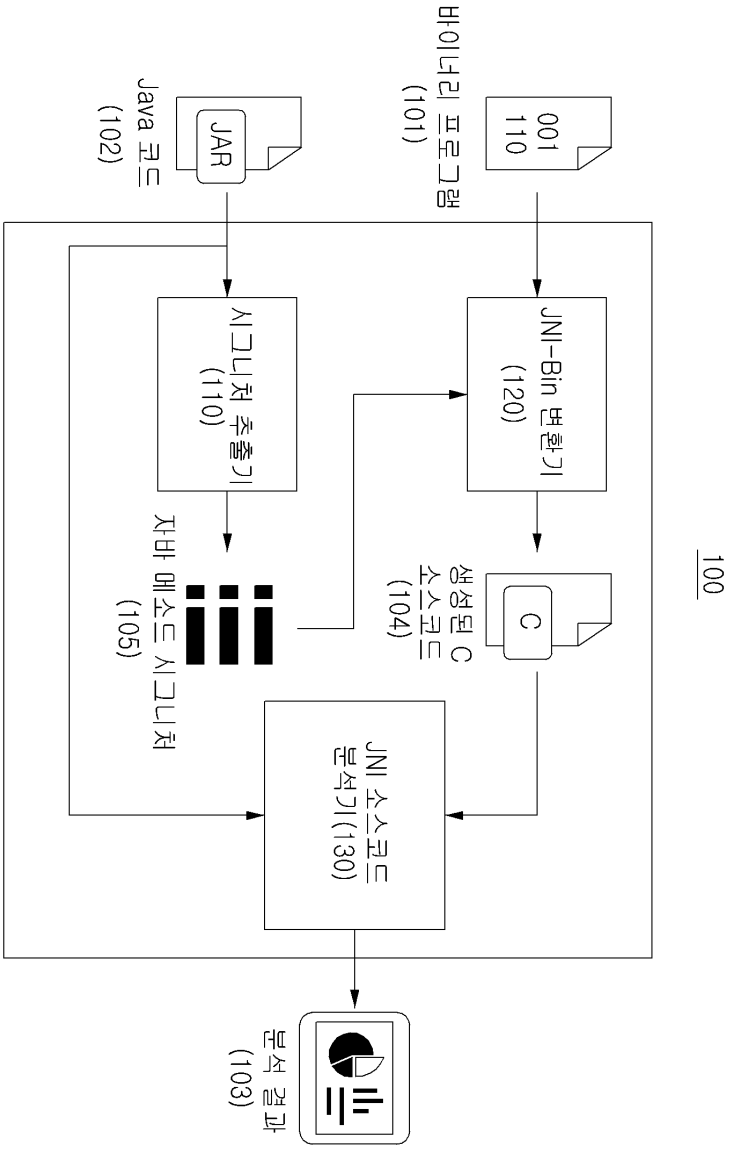
[0068] 이상과 같이 실시예들이 비록 한정된 실시예와 도면에 의해 설명되었으나, 해당 기술분야에서 통상의 지식을 가진 자라면 상기의 기재로부터 다양한 수정 및 변형이 가능하다. 예를 들어, 설명된 기술들이 설명된 방법과 다른 순서로 수행되거나, 및/또는 설명된 시스템, 구조, 장치, 회로 등의 구성요소들이 설명된 방법과 다른 형태로 결합 또는 조합되거나, 다른 구성요소 또는 균등물에 의하여 대치되거나 치환되더라도 적절한 결과가 달성될

수 있다.

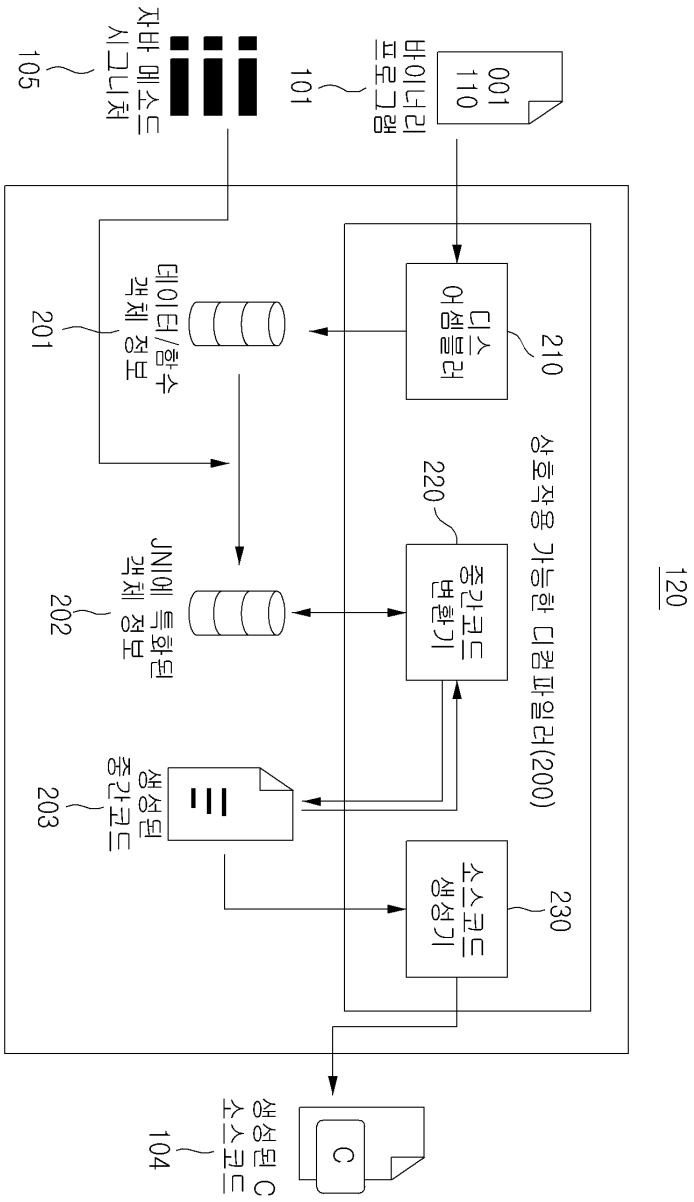
[0069] 그러므로, 다른 구현들, 다른 실시예들 및 특허청구범위와 균등한 것들도 후술하는 특허청구범위의 범위에 속한다.

도면

도면1



도면2


자바 메소드 시그니처
105
바이너리 프로그램
101
001
110
데이터/함수 객체 정보
201
디스어셈블러
210
상호작용 가능한 디컴파일러(200)
220
중간코드 변환기
JNI에 특화된 객체 정보
202
생성된 중간코드
203
소스코드 생성기
230
120
생성된 C 소스코드
104
C

도면3

```
1     int Java_get_1column_1name(
2   -     int a1, int a2, int a3, int a4, int a5) {
3   +     JNIEnv* a1, jobject a2, jlong a3, jint a4) {
4         int result;
5   -     if (sub_F1AC(a3, a5, a0))
6   -       result = (*(int (**)(int))(*(_DWORD *)a1 + 668))(a1);
7   +     if (sub_F1AC(a3, a4, a0))
8   +       result = (jstring)
9   +         (*(int (*)(JNIEnv *))(*a1)->NewStringUTF)(a1);
10        else
11          result = 0;
12        return result;
13      }
```

도면4

```
1     int Java_createFrame(JNIEnv* a1, ...) {
2       ...
3   -   jbitmap = (jobject)create_graphics(
4   -     (int)a1, v69[0], v69[1]);
5   +   jbitmap = (jobject)create_graphics(
6   +     a1, v69[0], v69[1]);
7       ...
8     }
9
10  - int create_graphics(int a1, int a2, int a3) {
11  + int create_graphics(JNIEnv* a1, int a2, int a3) {
12      ...
13  -   int v9;
14  -   v9 = (*(int (**)(int, const char*))(*(int *)a1 + 24)(
15  -     a1, "android/graphics/Bitmap"));
16  +   jclass v9;
17  +   v9 = (*a1)->FindClass(a1, "android/graphics/Bitmap");
18      ...
19    }
```

도면5

```
1       jstring Java_get1_column_1name(
2         JNIEnv *a1, jobject a2, jlong a3, jint a4) {
3   +     char* v5;
4         jstring result;
5
6   -     if (sub_F1AC(a3, a4, 0))
7   -       result = (jstring)(
8   -         (int (*)(JNIEnv *))(*a1)->NewStringUTF)(a1);
9   +     v5 = (char *)sub_F1AC(a3, a4, 0);
10  +     if (v5)
11  +       result = (*a1) -> NewStringUTF(a1, v5);
12        else
13          result = 0;
14        return result;
15    }
```

도면6

```
1     jstring Java_prepare(JNIEnv* a1, jobject a2,
2       jobject a3, jobject a4, jint a5) {
3
4       jint v10;
5       jmethodID v11;
6       jclass v14;
7       jstring v17;
8       ...
9       v14 = (*a1)->GetObjectClass(a1, a3);
10      if ( v14 ) {
11        v11 = (*a1)->GetMethodID(
12          a1, v14, "set_param", "(Ljava/lang/String;)I");
13        ...
14  -     v10 = (*a1)->CallIntMethod(a1, a3, v11, v17, a4);
15  +     v10 = (*a1)->CallIntMethod(a1, a3, v11, v17);
16        ...
17      }
18    }
```

도면7

```
1     const char *errmsg(int a1) {
2         ...
3 -       j_log(21, "API call with %s ...", "invalid");
4 +       log(21, "API call with %s ...", "invalid");
5         ...
6     }
7
8     int j_log(int a1, const char *a2, ...) {
9       return log(a1, a2);
10    }
```

도면8

Benchmark	JLoC	CLoC	Call J→C			Call C→J			Field C→J		
			$JSA_{DEC-IDA}$	$JSA_{DEC-Ghidra}$	JSA_{SC}	$JSA_{DEC-IDA}$	$JSA_{DEC-Ghidra}$	JSA_{SC}	$JSA_{DEC-IDA}$	$JSA_{DEC-Ghidra}$	JSA_{SC}
native complexdata	90	35	2	2	2	2	2	2	0	0	0
native complexdata stringop	88	29	1	1	1	0	0	0	0	0	0
native heap modify	63	26	1	1	1	2	2	2	2	2	2
native leak	61	17	1	1	1	0	0	0	0	0	0
native leak array	63	21	1	1	1	0	0	0	0	0	0
native method overloading	63	32	1	1	1	0	0	0	0	0	0
native multiple interactions	73	37	2	2	2	1	1	1	1	1	1
native multiple libraries	63	35	1	1	1	0	0	0	0	0	0
native noleak	62	13	1	1	1	0	0	0	0	0	0
native noleak array	63	21	1	1	1	0	0	0	0	0	0
native nosource	41	8	1	1	1	0	0	0	0	0	0
native set field from arg	109	22	1	1	1	0	0	0	2	2	2
native set field from arg field	113	23	1	1	1	0	0	0	3	3	3
native set field from native	100	42	1	1	1	3	3	3	5	5	5
native source	58	19	1	1	1	2	2	2	1	1	1
native source clean	89	19	1	1	1	0	0	0	1	1	1
Total			18	18	18	10	10	10	15	15	15

도면9

Application	JLoC	CLoC	Summary (#)			Call $_{J\rightarrow C}$			Field $_{C\rightarrow J}$		
			$JSA_{DEC-IDA}$	$JSA_{DEC-Ghidra}$	JSA_{SC}	$JSA_{DEC-IDA}$	$JSA_{DEC-Ghidra}$	JSA_{SC}	$JSA_{DEC-IDA}$	$JSA_{DEC-Ghidra}$	JSA_{SC}
AsciiCam	2272	120	3	3	3	0	0	0	0	0	0
PracticeHub	1058	348	1	0	1	1	0	1	0	0	0
simpleRT	97	493	3	3	3	3	3	3	0	0	0
SpiritF	6479	13226	2	2	2	1	1	1	0	0	0
AndroSS	1681	334	2	2	2	4	4	4	0	0	0
Overchan	52051	1721	18	15	18	0	0	0	0	0	0
Fwknop2	2220	8418	1	1	1	1	1	1	13	13	13
Compass	1683	42	3	3	3	0	0	0	0	0	0
Andlodine	1178	7972	9	9	9	5	5	5	0	0	0
Obsqr	1070	8673	1	1	1	0	0	0	0	0	0
Total			43	39	43	15	14	15	13	13	13

도면10

Benchmark	Data leakage		
	$JSA_{DEC-IDA}$	$JSA_{DEC-Ghidra}$	JN-SAF
native complexdata	◯	◯	◯
native complexdata stringop			
native_heap_modify	◯	◯	×
native leak	◯	◯	◯
native_leak_array	◯	◯	◯
native_method_overloading	◯	◯	◯
native_multiple_interactions	◯	◯	◯
native_multiple_libraries	◯	◯	◯
native_noleak			
native_noleak_array	⊗	⊗	⊗
native_nosource			
native_set_field_from_arg	◯◯	◯◯	◯×
native_set_field_from_arg_field	◯◯	◯◯	◯×
native_set_field_from_native	◯◯	◯◯	TIMEOUT
native_source	◯	◯	×
native_source_clean	⊗	⊗	
Total	16	16	9

◯ = True positive ⊗ = True positive ×= False negative