(19) 대한민국특허청(KR)
(12) 등록특허공보(B1)

(45) 공고일자 2022년01월17일
(11) 등록번호 10-2351663
(24) 등록일자 2022년01월11일

(51) 국제특허분류(Int. Cl.)
G06F 21/56 (2013.01) *G06F 21/54* (2013.01)
(52) CPC특허분류
G06F 21/566 (2013.01)
G06F 21/54 (2013.01)
(21) 출원번호 10-2020-0038983
(22) 출원일자 2020년03월31일
심사청구일자 2020년03월31일
(65) 공개번호 10-2021-0108848
(43) 공개일자 2021년09월03일
(30) 우선권주장
1020200023549 2020년02월26일 대한민국(KR)
(56) 선행기술조사문헌
JP2019121203 A*
US20150356294 A1*
*는 심사관에 의하여 인용된 문헌

(73) 특허권자
세종대학교산학협력단
서울특별시 광진구 능동로 209 (군자동, 세종대학교)
(72) 발명자
신동규
서울특별시 강남구 언주로 201, 1505호(도곡동, 에스케이리더스뷰)
정승훈
서울특별시 송파구 양산로4길 16, 506동 305호(거여동, 거여5단지아파트)
(74) 대리인
양성보

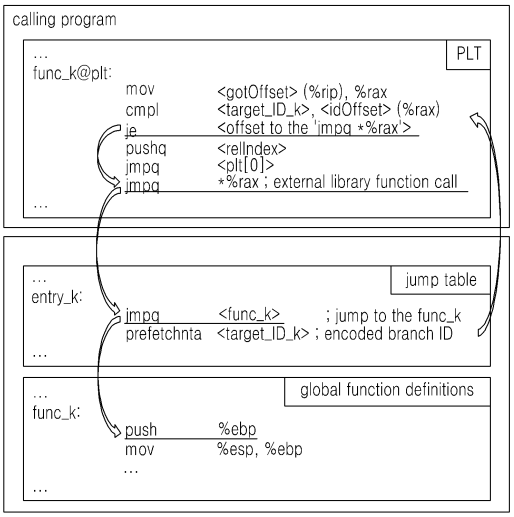
전체 청구항 수 : 총 6 항 심사관 : 정성훈

(54) 발명의 명칭 **CFI 기반 GOT 변조 공격 방지 장치 및 그 방법**

(57) 요 약

CFI 기반 GOT 변조 공격 방지 장치 및 그 방법이 개시된다. 일 실시예에 따른 컴퓨터로 구현되는 GOT(Global Offset Table) 보호 장치에 의해 수행되는 GOT 변조 공격 방지 방법은, 호출 프로그램과 라이브러리 간에 공유되는 동적 바인딩 심볼을 코드화하여 분기 식별자를 생성하는 단계; 상기 호출 프로그램에 포함된 분기 테이블에서 상기 코드화된 분기 식별자 코드를 검사하기 위한 명령 코드를 통해 분기 유효성 검사를 수행하는 단계; 및 상기 수행된 분기 유효성 검사에 기초하여 상기 호출 프로그램에 포함된 분기 테이블로부터 함수 호출을 통해 호출 프로그램을 실행시키는 단계를 포함할 수 있다.

대 표 도 - 도6



이 발명을 지원한 국가연구개발사업

과제고유번호	1345292887
과제번호	2018R1D1A1B07047395
부처명	교육부
과제관리(전문)기관명	한국연구재단
연구사업명	개인기초연구(교육부)(R&D)
연구과제명	빅데이터와 지능 추론에 기반한 사이버 위협 분석 및 예방 기술 연구
기 여 율	1/1
과제수행기관명	세종대학교
연구기간	2019.03.01 ~ 2020.02.29

공지예외적용 : 있음

명 세 서

청구범위

청구항 1

컴퓨터로 구현되는 GOT(Global Offset Table) 보호 장치에 의해 수행되는 GOT 변조 공격 방지 방법에 있어서,

호출 프로그램과 라이브러리 간에 공유되는 동적 바인딩 심볼을 코드화하여 분기 식별자를 생성하는 단계;

상기 호출 프로그램에 포함된 분기 테이블에서 상기 코드화된 분기 식별자 코드를 검사하기 위한 명령 코드를 통해 분기 유효성 검사를 수행하는 단계; 및

상기 수행된 분기 유효성 검사에 기초하여 상기 호출 프로그램에 포함된 분기 테이블로부터 함수 호출을 통해 호출 프로그램을 실행시키는 단계

를 포함하고,

상기 분기 식별자는 생성하는 단계는,

상기 동적 바인딩 심볼을 이름 꾸밈(name mangling) 규칙 기반의 코드화를 수행하는 단계

를 포함하고,

상기 분기 유효성 검사를 수행하는 단계는,

프로그램의 실행 환경에 따라 호출 프로그램이나 라이브러리 중 어느 하나에 CFI 기반의 GOT 변조 공격 방지 방법이 적용된 채로 호출이 발생될 경우, 상기 분기 유효성 검사 수행 여부를 결정하고, 상기 라이브러리에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않을 경우, 상기 라이브러리에서 분기 식별자를 포함하고 있는지 여부를 체크하여 분기 유효성 검사 코드 추가 여부를 결정하고, 상기 라이브러리에 분기 식별자를 포함하고 있을 경우, 상기 호출 프로그램의 PLT 엔트리에 분기 식별자를 포함하는 명령 코드를 삽입하고, 상기 라이브러리에 분기 식별자를 포함하고 있지 않을 경우, 동적 링커에 의한 심볼 바인딩을 통해 호출 프로그램을 다이렉트로 호출하는 단계

를 포함하는 GOT 변조 공격 방지 방법.

청구항 2

제1항에 있어서,

상기 분기 유효성 검사를 수행하는 단계는,

상기 라이브러리에 상기 생성된 분기 식별자를 삽입하고, 상기 라이브러리에 삽입된 분기 식별자와 상기 호출 프로그램 PLT(Procedure Linkage Table) 엔트리에서 상기 코드화된 분기 식별자 코드를 검사하기 위하여 추가된 명령 코드를 통해 GOT 엔트리가 가리키는 분기 목표점에 위치한 값을 비교하고, 상기 비교한 결과값이 일치할 경우 분기를 진행하고, 상기 비교한 결과값이 일치하지 않을 경우, GOT 엔트리 값을 재설정하는 단계

를 포함하는 GOT 변조 공격 방지 방법.

청구항 3

삭제

청구항 4

제1항에 있어서,

상기 분기 유효성 검사를 수행하는 단계는,

상기 호출 프로그램에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않을 경우, 분기 식별자 검사를 미수행하는 단계

를 포함하는 GOT 변조 공격 방지 방법.

청구항 5

삭제

청구항 6

삭제

청구항 7

컴퓨터로 구현되는 GOT(Global Offset Table) 보호 장치에 있어서,

호출 프로그램과 라이브러리 간에 공유되는 동적 바인딩 심볼을 코드화하여 분기 식별자를 생성하는 식별자 생성부;

상기 호출 프로그램에 포함된 분기 테이블에서 상기 코드화된 분기 식별자 코드를 검사하기 위한 명령 코드를 통해 분기 유효성 검사를 수행하는 유효성 검사부; 및

상기 수행된 분기 유효성 검사에 기초하여 상기 호출 프로그램에 포함된 분기 테이블로부터 함수 호출을 통해 호출 프로그램을 실행시키는 프로그램 실행부

를 포함하고,

상기 식별자 생성부는,

상기 동적 바인딩 심볼을 이름 꾸밈(name mangling) 규칙 기반의 코드화를 수행하는 것을 포함하고,

상기 유효성 검사부는,

프로그램의 실행 환경에 따라 호출 프로그램이나 라이브러리 중 어느 하나에 CFI 기반의 GOT 변조 공격 방지 방법이 적용된 채로 호출이 발생될 경우, 상기 분기 유효성 검사 수행 여부를 결정하고, 상기 라이브러리에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않을 경우, 상기 라이브러리에서 분기 식별자를 포함하고 있는지 여부를 체크하여 분기 유효성 검사 코드 추가 여부를 결정하고, 상기 라이브러리에 분기 식별자를 포함하고 있을 경우, 상기 호출 프로그램의 PLT 엔트리에 분기 식별자를 포함하는 명령 코드를 삽입하고, 상기 라이브러리에 분기 식별자를 포함하고 있지 않을 경우, 동적 링커에 의한 심볼 바인딩을 통해 호출 프로그램을 다이렉트로 호출하는

GOT 보호 장치.

청구항 8

제7항에 있어서,

상기 유효성 검사부는,

상기 라이브러리에 상기 생성된 분기 식별자를 삽입하고, 상기 라이브러리에 삽입된 분기 식별자와 상기 호출 프로그램 PLT(Procedure Linkage Table) 엔트리에서 상기 코드화된 분기 식별자 코드를 검사하기 위하여 추가된 명령 코드를 통해 GOT 엔트리가 가리키는 분기 목표점에 위치한 값을 비교하고, 상기 비교한 결과값이 일치할 경우 분기를 진행하고, 상기 비교한 결과값이 일치하지 않을 경우, GOT 엔트리 값을 재설정하는

것을 특징으로 하는 GOT 보호 장치.

청구항 9

삭제

청구항 10

제7항에 있어서,

상기 유효성 검사부는,

상기 호출 프로그램에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않을 경우, 분기 식별자 검사를 미수행하는

것을 특징으로 하는 GOT 보호 장치.

청구항 11

삭제

청구항 12

삭제

발명의 설명

기 술 분 야

[0001] 아래의 설명은 GOT 변조 공격 방지 기술에 관한 것이다.

배 경 기 술

[0003] 제어흐름탈취 공격이란, 버퍼 오버플로우 등의 메모리 오류 취약점을 이용하여 공격자가 의도한 프로그램 코드 주소로 프로그램의 실행흐름을 바꾸어 프로그램의 제어권을 탈취하는 행위를 일컫는다. GOT(Global Offset Table) 변조공격은 유닉스 계열 시스템 환경에서 소프트웨어 권한 탈취를 위한 전통적인 제어흐름 탈취 기법 중 하나로서, ELF(Executable and Linkable Format) 프로그램의 동적 바인딩 장치를 이용한다. ELF 프로그램에는 호출되는 라이브러리 함수 별로 PLT(Procedure Linkage Table)라는 이름의 분기 테이블을 포함하는데, 이 엔트리 명령 코드는 해당 라이브러리 함수가 호출될 때 동적 링커가 주소값을 찾아 바인딩해 둔 GOT 엔트리 값을 참조한다. GOT 공격은 바로 이 GOT 엔트리를 공격자의 분기 목표점 주소값으로 변조하고, 프로그램에서 해당 라이브러리 함수가 호출될 때 제어흐름을 탈취하는 공격기법이다.

[0004] GOT 변조를 방어하기 위한 기법들 중에서, ELF 파일 로딩시 주소 재배치를 마치고, GOT 영역을 읽기 전용 속성으로 만드는 Full Relro 기법이 GOT 변조 공격 방지를 위해서 효과적인 방어기법으로 활용되고 있다. 하지만, Full Relro 기법은 로딩시간의 지연을 초래하고, 라이브러리 파일에는 적용할 수 없는 문제가 있다. 특히, 최근의 정교한 코드 재사용 공격기법은 라이브러리 GOT 공격을 가능하게 하고, 있어 Full Relro 기법으로는 더 이상 방어할 수 없는 문제가 있다.

발명의 내용

해결하려는 과제

[0006] Full Relro에서 방어가 불가능하던 라이브러리 파일을 포함해 실행파일의 로딩 지연 및 성능 오버헤드를 최소화하면서도 GOT 공격을 효과적으로 차단하는 방법 및 장치를 제공할 수 있다.

[0007] CFI(Control Flow Integrity) 기법을 사용한 GOT 보호 장치 및 방법을 제공할 수 있다.

과제의 해결 수단

[0009] 컴퓨터로 구현되는 GOT(Global Offset Table) 보호 장치에 의해 수행되는 GOT 변조 공격 방지 방법은, 호출 프로그램과 라이브러리 간에 공유되는 동적 바인딩 심볼을 코드화하여 분기 식별자를 생성하는 단계; 상기 호출 프로그램에 포함된 분기 테이블에서 상기 코드화된 분기 식별자 코드를 검사하기 위한 명령 코드를 통해 분기 유효성 검사를 수행하는 단계; 및 상기 수행된 분기 유효성 검사에 기초하여 상기 호출 프로그램에 포함된 분기 테이블로부터 함수 호출을 통해 호출 프로그램을 실행시키는 단계를 포함할 수 있다.

[0010] 상기 분기 유효성 검사를 수행하는 단계는, 상기 라이브러리에 상기 생성된 분기 식별자를 삽입하고, 상기 라이브러리에 삽입된 분기 식별자와 상기 호출 프로그램 PLT(Procedure Linkage Table) 엔트리에서 상기 코드화된 분기 식별자 코드를 검사하기 위하여 추가된 명령 코드를 통해 GOT 엔트리가 가리키는 분기 목표점에 위치한 값을 비교하고, 상기 비교한 결과값이 일치할 경우 분기를 진행하고, 상기 비교한 결과값이 일치하지 않을 경우,

GOT 엔트리 값을 재설정하는 단계를 포함할 수 있다.

[0011] 상기 분기 유효성 검사를 수행하는 단계는, 프로그램의 실행 환경에 따라 호출 프로그램이나 라이브러리 중 어느 하나에 CFI 기반의 GOT 변조 공격 방지 방법이 적용된 채로 호출이 발생될 경우, 상기 분기 유효성 검사 수행 여부를 결정하는 단계를 포함할 수 있다.

[0012] 상기 분기 유효성 검사를 수행하는 단계는, 상기 호출 프로그램에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않을 경우, 분기 식별자 검사를 미수행하는 단계를 포함할 수 있다.

[0013] 상기 분기 유효성 검사를 수행하는 단계는, 상기 라이브러리에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않을 경우, 상기 라이브러리에서 분기 식별자를 포함하고 있는지 여부를 체크하여 분기 유효성 검사 코드 추가 여부를 결정하는 단계를 포함할 수 있다.

[0014] 상기 분기 유효성 검사를 수행하는 단계는, 상기 라이브러리에 분기 식별자를 포함하고 있을 경우, 상기 호출 프로그램의PLT 엔트리에 분기 식별자를 포함하는 명령 코드 삽입하고, 상기 라이브러리에 분기 식별자를 포함하고 있지 않을 경우, 동적 링커에 의한 심볼 바인딩을 통해 호출 프로그램을 다이렉트로 호출하는 단계를 포함할 수 있다.

[0015] 컴퓨터로 구현되는 GOT(Global Offset Table) 보호 장치는, 호출 프로그램과 라이브러리 간에 공유되는 동적 바인딩 심볼을 코드화하여 분기 식별자를 생성하는 식별자 생성부; 상기 호출 프로그램에 포함된 분기 테이블에서 상기 코드화된 분기 식별자 코드를 검사하기 위한 명령 코드를 통해 분기 유효성 검사를 수행하는 유효성 검사부; 및 상기 수행된 분기 유효성 검사에 기초하여 상기 호출 프로그램에 포함된 분기 테이블로부터 함수 호출을 통해 호출 프로그램을 실행시키는 프로그램 실행부를 포함할 수 있다.

[0016] 상기 유효성 검사부는, 상기 라이브러리에 상기 생성된 분기 식별자를 삽입하고, 상기 라이브러리에 삽입된 분기 식별자와 상기 호출 프로그램 PLT(Procedure Linkage Table) 엔트리에서 상기 코드화된 분기 식별자 코드를 검사하기 위하여 추가된 명령 코드를 통해 GOT 엔트리가 가리키는 분기 목표점에 위치한 값을 비교하고, 상기 비교한 결과값이 일치할 경우 분기를 진행하고, 상기 비교한 결과값이 일치하지 않을 경우, GOT 엔트리 값을 재설정할 수 있다.

[0017] 상기 유효성 검사부는, 프로그램의 실행 환경에 따라 호출 프로그램이나 라이브러리 중 어느 하나에 CFI 기반의 GOT 변조 공격 방지 방법이 적용된 채로 호출이 발생될 경우, 상기 분기 유효성 검사 수행 여부를 결정할 수 있다.

[0018] 상기 유효성 검사부는, 상기 호출 프로그램에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않을 경우, 분기 식별자 검사를 미수행할 수 있다.

[0019] 상기 유효성 검사부는, 상기 라이브러리에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않을 경우, 상기 라이브러리에서 분기 식별자를 포함하고 있는지 여부를 체크하여 분기 유효성 검사 코드 추가 여부를 결정할 수 있다.

[0020] 상기 유효성 검사부는, 상기 라이브러리에 분기 식별자를 포함하고 있을 경우, 상기 호출 프로그램의PLT 엔트리에 분기 식별자를 포함하는 명령 코드 삽입하고, 상기 라이브러리에 분기 식별자를 포함하고 있지 않을 경우, 동적 링커에 의한 심볼 바인딩을 통해 호출 프로그램을 다이렉트로 호출할 수 있다.

발명의 효과

[0022] 동적 바인딩 함수 심볼을 CFI의 분기 식별자로 사용하고, 호출 프로그램의 PLT 호출 지점에서 라이브러리 점프 테이블에 코드화된 분기 식별자 코드를 검사함으로써 GOT 변조 공격을 차단할 수 있다.

[0023] 또한, Full Relro에서 방어가 불가능하던 라이브러리 파일을 포함해 실행파일의 로딩 지연 및 성능 오버헤드를 최소화하면서도 GOT 공격을 효과적으로 차단할 수 있다.

[0024] 또한, 성능 오버헤드를 최소화하여 점진적인 빌드가 가능하고, 기존의 라이브러리와도 높은 호환성을 가진다.

도면의 간단한 설명

[0026] 도 1은 프로그램에서 동적 라이브러리 함수를 호출하는 것을 설명하기 위한 도면이다.

도 2는 일 실시예에 따른 GOT 보호 장치의 구성을 설명하기 위한 블록도이다.

도 3은 일 실시예에 따른 GOT 보호 장치에서 GOT 변조 공격을 방지하는 방법을 설명하기 위한 흐름도이다.

도 4는 일 실시예에 있어서, 프로그램과 라이브러리 간 동적 함수 심볼을 공유하는 것을 설명하기 위한 도면이다.

도 5는 일 실시예에 있어서, 호출 프로그램의 PLT 엔트리에서 분기 식별자를 검사하는 것을 설명하기 위한 도면이다.

도 6은 일 실시예에 있어서, 라이브러리 점프 테이블을 설명하기 위한 도면이다.

도 7은 일 실시예에 있어서, 라이브러리 함수를 호출하는 예제 프로그램에 대하여 라이브러리 점프 테이블을 구현한 것을 설명하기 위한 예이다.

도8은 일 실시예에 있어서, 호출 프로그램에서의 분기 검증을 설명하기 위한 도면이다.

도 9는 일 실시예에 있어서, 프로그램 그룹의 파일 크기 정보를 설명하기 위한 예이다.

도 10은 일 실시예에 있어서, 프로그램 그룹이 호출한 라이브러리의 파일 크기 정보를 설명하기 위한 도면이다.

도 11은 일 실시예에 있어서, 프로그램별 서로 다른 기법에 의한 동적 링커의 재배치 회수와 로딩시간을 설명하기 위한 도면이다.

발명을 실시하기 위한 구체적인 내용

[0027] 이하, 실시예를 첨부한 도면을 참조하여 상세히 설명한다.

[0029] 도 1은 프로그램에서 동적 라이브러리 함수를 호출하는 것을 설명하기 위한 도면이다.

[0030] 유닉스 계열 정적 링커는 동적 라이브러리 링크를 지원하기 위해 PLT, GOT 두 구조체를 생성하여 호출 프로그램의 ELF 파일의 섹션으로 포함시킬 수 있다. 이때, 동적 라이브러리 링크는 호출 프로그램과 라이브러리 간의 함수에 대한 심볼 바인딩 프로세스가 실행 시간에 동적으로 이루어지도록 하는 기법이다. 동적 링커와 정적 링커는 서로 협력하여 모듈 간 심볼 바인딩이 이루어지도록 한다. PLT는 호출 프로그램과 라이브러리 함수를 연결하는 함수 호출 테이블로서 정적 링커에 의해 최종 실행 프로그램 혹은 라이브러리 파일 단위로 포함된다. 정적 링커는 각 라이브러리 함수 호출 별로 PLT 엔트리를 생성하고, 원래의 함수 호출 명령문을 PLT 엔트리로의 오프셋 직접분기로 수정할 수 있다. PLT 엔트리에는 짝을 이루고 있는 GOT 엔트리의 값을 참조하여 목표 분기점 주소로 간접 호출을 실행하는 명령 코드가 포함될 수 있다. GOT는 호출 프로그램에 동적으로 바인딩되는 라이브러리 함수들의 실행시간 주소 값을 담는 포인터 배열 구조체로서 앞서 언급했듯이 동적 링커에 의해 주소값이 저장될 수 있다.

[0031] 실시예에서는 CFI(Control Flow Integrity)를 적용한 GOT 변조 공격 방지 방법 및 GOT 보호 장치를 설명하기로 한다. CFI 기법은 정상 제어흐름 그래프를 생성하는 분석 단계와 실행 시간에 정상 제어흐름 그래프 하에서 제어흐름 분기가 이루어지도록 분기 유효성 검사 코드를 삽입하는 단계로 구성될 수 있다.

[0032] 실시예에서 제안하는 CFI를 적용한 GOT 변조 공격 방지 방법은 일반적인 리눅스 방어 기법이 동작하는 실행 환경과 강력한 공격 모델을 가정한다. 실행 프로그램 및 라이브러리 파일은 ELF 파일을 대상으로 하며, 메모리 페이지는 DEP에 의해 실행과 동시에 쓰기가 가능하지 않다. 과도한 프로그램 오류를 발생시키지 않는 수준에서 공격자는 메모리 변조와 정보 누출 공격을 통해 모든 데이터 페이지 메모리를 읽거나 쓸 수 있고 코드 페이지 메모리를 읽을 수 있다. 프로그램 및 라이브러리 파일은 Partial Relro가 적용되고, 프로그램의 외부 라이브러리 호출은 PLT/GOT 를 통해 이루어진다. 공격자는 프로그램 혹은 라이브러리의 GOT 테이블 엔트리 변조를 통한 제어흐름탈취 공격을 시도한다. 도 1에 도시된 바와 같이, 프로그램 실행 환경에서 모듈 간 호출에 대해 GOT 변조 공격을 방지하기 위한 방법 및 장치를 제안할 수 있다.

[0033] 도 2는 일 실시예에 따른 GOT 보호 장치의 구성을 설명하기 위한 블록도이고, 도 3은 일 실시예에 따른 GOT 보호 장치에서 GOT 변조 공격을 방지하는 방법을 설명하기 위한 흐름도이다.

[0034] GOT 보호 장치(100)에 포함된 프로세서는 식별자 생성부(210), 유효성 검사부(220) 및 프로그램 실행부(230)를 포함할 수 있다. 이러한 프로세서 및 프로세서의 구성요소들은 도 3의 GOT 변조 공격을 방지하는 방법이 포함하는 단계들(310 내지 330)을 수행하도록 GOT 보호 장치를 제어할 수 있다. 이때, 프로세서 및 프로세서의 구성요소들은 메모리가 포함하는 운영체제의 코드와 적어도 하나의 프로그램의 코드에 따른 명령(instruction)을 실행하도록 구현될 수 있다. 여기서, 프로세서의 구성요소들은 GOT 보호 장치(100)에 저장된 프로그램 코드가

제공하는 제어 명령에 따라 프로세서에 의해 수행되는 서로 다른 기능들(different functions)의 표현들일 수 있다.

[0035] 프로세서는 GOT 변조 공격을 방지하는 방법을 위한 프로그램의 파일에 저장된 프로그램 코드를 메모리에 로딩할 수 있다. 예를 들면, GOT 보호 장치(100)에서 프로그램이 실행되면, 프로세서는 운영체제의 제어에 따라 프로그램의 파일로부터 프로그램 코드를 메모리에 로딩하도록 GOT 보호 장치를 제어할 수 있다.

[0036] 단계(310)에서 식별자 생성부(210)는 호출 프로그램과 라이브러리 간에 공유되는 동적 바인딩 심볼을 코드화하여 분기 식별자를 생성할 수 있다.

[0037] 단계(320)에서 유효성 검사부(220)는 호출 프로그램에 포함된 분기 테이블에서 코드화된 분기 식별자 코드를 검사하기 위한 명령 코드를 통해 분기 유효성 검사를 수행할 수 있다. 유효성 검사부(220)는 라이브러리에 생성된 분기 식별자를 삽입하고, 라이브러리에 삽입된 분기 식별자와 호출 프로그램 PLT(Procedure Linkage Table) 엔트리에서 코드화된 분기 식별자 코드를 검사하기 위하여 추가된 명령 코드를 통해 GOT 엔트리가 가리키는 분기 목표점에 위치한 값을 비교하고, 비교한 결과값이 일치할 경우 분기를 진행하고, 비교한 결과값이 일치하지 않을 경우, GOT 엔트리 값을 재설정할 수 있다. 유효성 검사부(220)는 프로그램의 실행 환경에 따라 호출 프로그램이나 라이브러리 중 어느 하나에 CFI 기반의 GOT 변조 공격 방지 방법이 적용된 채로 호출이 발생될 경우, 분기 유효성 검사 수행 여부를 결정할 수 있다. 유효성 검사부(220)는 호출 프로그램에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않을 경우, 분기 식별자 검사를 미수행할 수 있다. 유효성 검사부(220)는 라이브러리에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않을 경우, 라이브러리에서 분기 식별자를 포함하고 있는지 여부를 체크하여 분기 유효성 검사 코드 추가 여부를 결정할 수 있다. 유효성 검사부(220)는 라이브러리에 분기 식별자를 포함하고 있을 경우, 호출 프로그램의PLT 엔트리에 분기 식별자를 포함하는 명령 코드 삽입하고, 라이브러리에 분기 식별자를 포함하고 있지 않을 경우, 동적 링커에 의한 심볼 바인딩을 통해 호출 프로그램을 다이렉트로 호출할 수 있다.

[0038] 단계(330)에서 프로그램 실행부(230)는 수행된 분기 유효성 검사에 기초하여 상기 호출 프로그램에 포함된 분기 테이블로부터 함수 호출을 통해 호출 프로그램을 실행시킬 수 있다.

[0039] 도 4는 일 실시예에 있어서, 프로그램과 라이브러리 간 동적 함수 심볼을 공유하는 것을 설명하기 위한 도면이다.

[0040] GOT 보호 장치는 동적 바인딩 심볼을 사용한 정적 분기 식별자를 공유할 수 있다. 모듈 간 제어흐름에 CFI 적용을 위해서는 라이브러리가 고려된 분기 식별자 할당이 이루어져야 한다. 동적 바인딩 함수 심볼은 호출 프로그램과 라이브러리 간에 정적으로 공유된 식별자이다. 동적 링커의 바인딩 과정은 호출 프로그램의 라이브러리 목록과 심볼 테이블을 사용할 수 있다. 도 4를 참고하면, 동일한 라이브러리 함수를 호출하는 서로 다른 프로그램은 각각의 ELF 파일의 '*.dynamic*' 섹션에 동일한 라이브러리 파일에 대한 엔트리와, '*.dynsym*' 섹션에 동일한 함수 심볼에 대한 엔트리를 포함한 것을 확인할 수 있다. 동적 바인딩 함수 심볼은 PLT 엔트리에서 분기 식별자로 사용될 수 있는데, 이때, 각 연결된 GOT 엔트리에 바인딩되는 함수 심볼은 고유하기 때문이다. 이때, 정적 분석시 다른 간접 호출 분기점에서는 호출 함수의 심볼이 일반적으로 고유하게 결정되지 않는다.

[0041] 동적 바인딩 함수 심볼은 작은 크기의 분기 식별자 그룹의 요건을 충족할 수 있다. 라이브러리 함수는 일반적으로 외부 연결(External Linkage)의 속성을 갖고 있어 프로세스 공간에서 고유하게 식별이 된다. 정적 링크시 외부 연결(External Linage) 속성을 갖고 있는 동일한 이름의 서로 다른 함수가 존재할 때, 정적 링커는 중복된 심볼 오류(duplicated symbol error)를 일으키기 때문에 실행 시간에 호출 함수의 심볼 충돌은 발생하지 않는다. 만약, 호출 함수의 심볼이 약한 연결(Weak Linkage) 속성을 갖고 있다면, 해당 함수는 다수의 라이브러리에 중복되어 포함되어 있을 수 있다. 이 경우, 호출 프로그램의 실행시간 환경에 따라 동일한 함수 심볼에 대해 서로 다른 주소가 바인딩 될 수 있어 결과적으로 약한 연결의 심볼은 다소 큰 분기 그룹을 형성할 수 있다. 한편, 하나의 모듈에서 정의된 외부 연결 속성의 함수 심볼이 다른 모듈에서는 지역 연결(Local Linkage) 속성으로 정의될 수도 있는데, 지역 연결 속성의 함수는 동적 링크의 대상에서 제외되므로(즉, 동적 심볼 테이블에 존재하지 않음) 동적 바인딩 함수 심볼은 고유한 분기 식별자 선택을 위한 좋은 후보가 될 수 있다.

[0042] GOT 보호 장치는 동적 바인딩 함수 심볼을 CFI의 분기 식별자로 사용하기 위하여 동적 바인딩 심볼의 코드화를 수행할 수 있다. 라이브러리 함수 심볼은 가변적 길이의 문자열이어서 분기 식별자 코드로 직접 사용하기에는 부적절하다. 또한, C++ 언어에서는 템플릿 선언 등에 의해 다른 형식의 동일한 함수 이름이 정의될 수 있다.

예를 들면, 분기 식별자로 사용하기 위해 동적 바인딩 심볼은 다음의 코드화 과정이 수행될 수 있다. 일례로, GOT 보호 장치는 해시함수를 사용하여 하나의 명령 코드에 로딩할 수 있는 고정된 길이의 비트코드를 생성하되 해시함수의 입력이 되는 문자열은 이름 꾸밈(name mangling) 규칙에 의해 전처리된 심볼 이름이 사용될 수 있다. 컴파일러에 의해 이름 꾸밈된 심볼은 동적 링커가 바인딩시 사용하는 문자열이고, 중복성이 없기 때문에 식별자 생성을 위한 별도의 이름 꾸밈이 필요하지 않다.

[0043] 도 5는 일 실시예에 있어서, 호출 프로그램의 PLT 엔트리에서 분기 식별자를 검사하는 것을 설명하기 위한 도면이다.

[0044] GOT 보호 장치는 호출 프로그램 PLT 엔트리의 분기 식별자를 점검할 수 있다. GOT 보호 장치는 전역 함수 심볼을 분기 식별자로 사용하기 때문에 분기원점에서 단순한 비교 명령문 실행을 통해 분기 유효성 검사를 수행할 수 있다. 도 5를 참고하면, 분기 목표 모듈에 분기 식별자 테이블이 존재할 경우 분기 유효성 검사 명령코드를 추가하는 링커의 의사 알고리즘을 나타낸 예이다. GOT 보호 장치는 GOT 엔트리가 가리키는 분기 목표점에 위치한 분기 식별자 명령 코드를 로딩하고, 분기 식별자 명령 코드를 로딩함에 따른 값과 라이브러리의 분기 식별자를 비교하고, 비교한 값이 일치하면 분기를 정상적으로 진행하고, 그렇지 않으면 동적 바인딩을 통해 GOT 엔트리 값을 재설정할 수 있다. 동적 바인딩은 동적 링커에 의한 지연된 바인딩(Lazy Binding or Lazy Symbol Resolution) 장치를 수정 없이 사용할 수 있다. 다시 말해서, 재배치 테이블의 엔트리 순번을 스택에 저장하고, PLT[0] 로 분기하여 동적 링커가 실행되도록 한다. 도 5의 'je' 분기문 다음에 오는 두 명령 코드는 동적 바인딩의 시작을 나타낸다.

[0045] PLT 엔트리의 명령 코드는 GOT 엔트리 값을 참조하는 무조건 분기에서 분기 식별자 검사가 포함된 조건부 분기로 변경될 수 있다. 이때, 복수 개(예를 들면, 세 개)의 명령 코드가 추가될 수 있다. GOT 보호 장치는 프로그램의 첫 호출 때와 공격자에 의해 GOT 엔트리가 변조되었을 때, 식별자 검사에 실패할 수 있다. 첫 호출시 GOT 엔트리 초기값이 가리키는 주소에는 분기 식별자 코드가 존재하지 않는다. 이 경우, 동적 링커의 지연된 바인딩 과정에 의해 GOT 엔트리는 올바른 함수 주소 값으로 수정될 수 있다. 공격자는 분기 유효성 검사를 우회하기 어렵다. 상기에서 언급한 공격 모델을 가정할 때 공격자는 모든 GOT 엔트리를 원하는 분기 목표점 주소 값으로 조작할 수 있다. 그러나, PLT 코드 영역의 수정은 불가하므로 공격자는 분기 식별자 검사 명령 코드 및 분기 식별자 코드를 변조할 수 없다. 이에 따라, GOT 엔트리 변조에 성공하더라도 PLT 엔트리의 분기 유효성 검사에서 실패하고, 변조된 GOT 엔트리는 동적 링커에 의해 올바른 함수 주소로 갱신되어 GOT 변조에 의한 제어 흐름탈취 공격에 실패하게 된다.

[0046] 도 6은 일 실시예에 있어서, 라이브러리 점프 테이블을 설명하기 위한 도면이다. GOT 보호 장치는 라이브러리 점프 테이블을 생성할 수 있다. PLT 엔트리에서의 분기 유효성 검사는 라이브러리 코드에 분기 식별자 코드가 삽입되어야 한다. 이때, 라이브러리 함수에 분기 식별자 코드를 직접 삽입하는 방법은 함수의 오프셋 변화를 가져와 추가적인 코드 주소의 재배치 및 호환성 문제를 일으킬 수 있다. 이에, GOT 보호 장치는 라이브러리 내 기존 코드 섹션의 수정이 없도록, 분기 식별자 검사를 위한 별도의 점프 테이블을 생성할 수 있다. 점프 테이블은 라이브러리의 외부 인터페이스가 되어 점프 테이블의 각 엔트리는 라이브러리 함수 호출 진입점이 되고, 라이브러리 내부에 정의된 각 함수와 연결될 수 있다.

[0047] 도 6을 참고하면, 분기 테이블 엔트리와 라이브러리 함수의 호출 연결 관계를 도시한 것이다. 점프 테이블의 각 엔트리 코드는 짝을 이루는 라이브러리 함수로의 직접 분기 명령 코드와 해당 함수 심볼의 분기 식별자 코드를 포함하는 명령 코드로 구성될 수 있다. 점프 테이블 전체 명령 코드는 하나의 독립된 함수로 생성하여 기존 라이브러리 내에 추가적인 주소 재배치가 없도록 한다. 또한, 호출 프로그램이 라이브러리 함수 호출시 점프 테이블 엔트리 주소가 바인딩 될 수 있도록 정적 링커는 라이브러리 함수의 심볼 주소가 점프 테이블 엔트리를 가리키도록 변경할 수 있다.

[0048] 점프 테이블이 포함된 라이브러리는 적은 양의 명령 코드로 효율적인 모듈간 분기 유효성 검사를 가능하게 한다. 라이브러리는 호출 함수 당 점프 테이블 내에 하나의 엔트리를 가지며, 각 엔트리는 2개의 명령 코드만을 포함한다. 하나의 직접 분기 명령 코드만 부가적으로 실행되므로 메모리 및 성능 오버헤드는 제한적이다. 분기 식별자가 인코딩된 명령코드는 분기 원점에서 로딩 되지만, 라이브러리에서 실행되지는 않는다.

[0049] 도 7은 일 실시예에 있어서, 라이브러리 함수를 호출하는 예제 프로그램에 대하여 라이브러리 점프 테이블을 구현한 것을 설명하기 위한 예이다.

[0050] 일례로, CFI 기반의 GOT 변조 공격 방지 방법은 LLVM 10 버전의 Module Pass 프레임워크 및 LLD 링커 프로젝트

를 기반으로 구현될 수 있고, X86-64 아키텍처를 대상으로 한다. 라이브러리의 점프 테이블 생성 코드는 Module Pass에 기반하여 LTO(Link Time Optimization) 라이브러리(LLVMgold.so)의 일부로 구현되어 LLD 링커의 플러그인으로 입력될 수 있다. 호출 프로그램의 분기 유효성 검사는 LLD 링커의 PLT 생성 코드를 수정하여 구현되는 것을 예를 들어 설명하기로 한다.

[0051] LLVM 컴파일러는 특정 단위의 코드 분석 및 변환을 위한 프레임워크를 제공할 수 있다. 예를 들면, LLVM은 '*Module*', '*Function*', '*Basic Block*' 단위의 코드 분석 및 변환을 위한 *Pass* 프레임워크를 제공할 수 있으며, 실시예에서는 *Module Pass* 단계에서 구현될 수 있다. *Module Pass* 단계에서는 입력 파일 단위의 분석 및 코드 최적화가 수행될 수 있다. 구현된 Pass는 모듈 내 지역 연결 속성이 아닌 정의된 함수들을 목록화하고 모듈 별로 하나의 점프 테이블을 구성한 후 각 함수의 시작점으로 분기하는 엔트리 함수를 생성할 수 있다. 점프 테이블은 타 컴파일러(예를 들면, LLVM)의 함수 형식 검증 기반 CFI 구현에 사용된 점프 테이블과 유사한 방식으로 구성될 수 있다. 점프 테이블 엔트리의 심볼은 정의된 함수의 심볼명과 외부 연결 속성을 갖도록 한다. 정의된 함수 심볼에는 *.cfi* 접미사를 붙이고 지역 연결 속성으로 변환하여, 외부 호출 모듈을 포함해 기존 함수 심볼을 참조하던 코드들은 모두 점프 테이블 엔트리를 참조하도록 한다. 타 컴파일러(예를 들면, LLVM)에서 구현된 점프 테이블과는 달리 엔트리 별 분기 명령 코드 다음에는 분기 식별자가 인코딩된 명령 코드를 배치하여 호출 원점에서의 분기 유효성 검사를 가능하게 한다. 분기 식별자로는 이름 꾸밈된 함수의 심볼에서 MD5 해시 값을 취한 후 상위 4바이트를 리틀 엔디안(little endian) 형식으로 변환된 값이 사용될 수 있다. 이 값은 분기 식별자를 인코딩하기 위해 사용된 명령 코드(prefetchnta instruction)의 하위 4바이트에 배치될 수 있다. 'prefetchnta' 는 8바이트로 구성된 X86-64 명령 코드로서, 캐시에 명령코드를 미리 읽어오는 기능을 수행하며 실행시 기능적 부작용(side effect)은 없다. 도7 은 2개의 전역 함수가 포함된 라이브러리 예제 소스코드와 생성된 점프 테이블 코드를 *objdump* 명령으로 확인한 결과를 나타낸 예이다. 각 점프 테이블 엔트리는 16바이트로 구성되고 'prefetchnta' 명령 코드 다음에 오는 3개의 'int3' 명령 코드는 16바이트 정렬을 위한 패딩 바이트이다.

[0052] 모듈 별 점프 테이블 생성은 링크 시간에 코드 최적화를 수행하는 LTO 단계에서 이루어진다. LTO 단계의 코드 변환은 LLVM IR 비트코드(bitcode) 형식의 입력 모듈을 요구하기 때문에 실시예에서는 Clang 10 컴파일러에 *-flto* 컴파일 옵션을 사용하여 링커의 입력 모듈을 컴파일할 수 있다. 비트코드가 아닌 오브젝트 모듈은 링킹 시 LTO 단계에서 코드 변환이 수행되지 않아 해당 함수들은 생성 라이브러리의 점프 테이블 엔트리에 포함되지 않는다. 실시예에서는 동적 분기를 보호대상으로 하므로 동적 라이브러리로 링크될 오브젝트 모듈에 대해서만 LLVM IR 형식의 파일 생성이 필요하고, 실행 파일이나 정적 아카이브로 링크될 모듈에 대해서는 LLVM IR 비트코드를 생성할 필요는 없다.

[0053] 호출 프로그램에서의 분기 유효성 검증은 LLVM의 링커 프로젝트인 LLD 의 PLT 생성 코드를 수정하여 구현될 수 있다. 각 PLT 엔트리 별로 분기 유효성을 검사하기 위한 명령 코드가 생성될 수 있다. 분기 식별자는 참조할 GOT 엔트리와 연결된 함수 심볼로부터 점프 테이블 엔트리에서와 같은 방식으로 MD5 해시 값을 취하여 도출될 수 있다. 분기 식별자 검사를 위한 명령 코드는 GOT 엔트리가 가리키는 주소로부터 9바이트('jmp' 명령코드 5바이트 + 'prefetchnta' 명령코드 내의 분기 식별자 오프셋 4바이트) 떨어진 곳에 위치한 4바이트의 값의 분기 식별자와 비교될 수 있다. 도 8은 도 7의 라이브러리 함수를 호출하는 예제 프로그램에 대해 실시예에서 제안된 방법을 적용한 결과로서, 분기 식별자 점검 코드가 포함된 PLT 를 보여준다. 'je' 명령 코드를 통해 분기 식별자가 동일하면 라이브러리 함수로의 분기가 이루어지고 그렇지 않으면 지연된 바인딩을 위한 명령 코드가 실행될 수 있다. 각 PLT 엔트리는 32 바이트로 구성되고, 'jmpq' 명령 코드 다음에 오는 4개의 'int3' 명령 코드는 32바이트 정렬을 위한 패딩 바이트이다.

[0054] PLT 엔트리 명령코드의 변화는 GOT 엔트리 초기값에 영향을 줄 수 있다. Partial Relro의 경우 GOT 엔트리 초기값은 동적 링커가 사용할 재배치 인덱스를 저장하는 'pushq' 명령 코드의 주소를 가리킨다. 실시예에 따른 PLT 엔트리에서는 분기 유효성 검사가 실패하면 'pushq' 명령 코드로 분기하여 동적 링킹을 수행하므로 GOT 엔트리 값 초기화는 별도로 필요하지 않다. 한편, GOT 보호 장치는 제어흐름 보호가 적용되지 않은 라이브러리 함수에 대해 호출 프로그램에서 분기 유효성 검사를 수행할 수 있다. 이 경우 불필요한 분기 식별자 점검에 의한 성능 오버헤드가 발생할 수 있어, PLT 엔트리 별로 선택적인 분기 유효성 검사가 가능하도록 제공할 수 있다. 링커에 의해 PLT 엔트리 생성 시 라이브러리 함수가 제어흐름 보호가 적용되어 분기 테이블을 포함하고 있는지를 판단될 수 있다. GOT 보호 장치는 라이브러리의 지원 여부에 따라 분기 유효성 검사 코드의 추가 여부를 결정할 수 있다. 실시예에서는, 라이브러리의 경로에 특정 문자열이 포함된 경우에 한해 해당 함수 호출시 분기 유효성 검사 코드를 추가할 수 있다.

[0055] 도8은 일 실시예에 있어서, 호출 프로그램에서의 분기 검증을 설명하기 위한 도면이다.

[0056] GOT 보호 장치는 효용성 검증을 위해 binutils-gdb 프로그램 그룹을 대상으로 증가된 파일 사이즈의 크기를 측정하고, 보안성, 성능, 호환성을 평가할 수 있다. binutils-gdb 프로그램 그룹은 리눅스에서 활용도가 높고, 복잡성과 규모면에서 시험 평가에 적절한 다종의 프로그램을 포함하고 있다. 일례로, AMD Ryzen 3700X CPU, Ubuntu 18.04 LTS 환경에서 binutils-gdb 2.33 버전을 사용하여 평가가 수행될 수 있다.

[0057] 도 8을 참고하면, 호출 프로그램에서는 분기 유효성 검사 코드가 추가되고, 라이브러리에서는 분기테이블이 생성되므로 모듈의 바이너리 크기가 증가한다. 호출 프로그램에서는 X86-64 의 경우 원래 PLT의 각 엔트리 크기는16바이트이나 실시예에 따른 CFI 기반의 GOT 변조 공격 방지 방법을 적용하였을 경우, PLT 엔트리 크기는 32바이트로 증가한다. 이에 따라, 호출 프로그램의 바이너리 파일은 {PLT 엔트리 개수*16바이트} 만큼 크기가 증가될 수 있다.

[0058] 도9 를 참고하면, binutils-gdb 프로그램 그룹의 파일 크기 정보를 설명하기 위한 도면이다. Partial Relro 로 컴파일 후 디버그 정보가 제거된 binutils-gdb 프로그램 그룹에 대해 실행파일 별로 증가된 파일 크기 정보를 나타낸다. 실제 바이너리 크기 증가량은 PLT 크기 증가량과 일치하지는 않는데 이는 Partial Relro 적용에 따른 *.dynamic* 섹션 엔트리 개수의 감소, *.got.plt* 섹션의 배치 세그먼트 변경 및 세그먼트 페이지 바운더리 정렬에 의한 바이트 패딩 수 차이 때문이다. 일반적으로 PLT 엔트리 개수와 무관하게 PLT 의 크기가 바이너리 전체 크기에서 차지하는 비율은 낮다. 이에 따라, 호출 프로그램의 바이너리 크기 증가율은 도 9에서 보는 것과 같이 낮은 수준을 보인다.

[0059] 또한, 라이브러리에는 정의된 전역 함수 별로 분기테이블 엔트리가 생성될 수 있다. 도 7을 참고하면, 각 점프 테이블 엔트리는 5바이트의 무조건 분기 명령코드와 분기 식별자가 인코딩된 8바이트의 명령 코드 및 16바이트 정렬을 위한 3바이트의 패딩으로 구성될 수 있다. 따라서, 라이브러리의 경우 {호출 함수 개수*16바이트} 크기의 점프 테이블이 포함되어 파일 크기는 증가하며, 라이브러리가 또 다른 의존 라이브러리를 포함하는 경우 분기 유효성 검사 코드에 의해 바이너리 크기는 더 커질 수 있다.

[0060] 표 1은 binutils-gdb 프로그램 그룹에 포함된 동적 라이브러리 *libinproctrace.so* 에 대해 각각의 변조 공격 방지 동작의 적용 전후 바이너리 크기 정보를 비교한 결과이다.

[0061] 표 1:

original file size	increased PLT size	jump table size	transformed file size	overall increased ratio
64312	736	96	68272	6.15%

[0062]

[0063] 일례로, binutils-gdb 프로그램 그룹은 PLT 개수(53개) 및 전역 호출 함수의 개수(6개)가 작은 하나의 동적 라이브러리만 포함하고 있어, 규모가 큰 라이브러리에 대한 파일 크기가 측정될 수 있다. 도 10은 프로그램 그룹이 호출한 라이브러리의 파일 크기 정보를 설명하기 위한 도면이다. 도 10을 참고하면, binutils-gdb 프로그램 그룹에서 의존하는 동적 라이브러리의 PLT 엔트리 개수와 정의된 전역 함수의 개수를 기반으로 변조 공격 방지 동작의 적용에 따른 파일 크기 증가치의 추정값을 나타낸 것이다. 실제의 파일 크기는 세그먼트 바운더리 정렬 등에 의해 페이지 크기 범위에서 차이가 발생할 수 있다. 라이브러리 모듈은 특성상 일반적으로 실행파일에 비해 더 많은 전역 함수 심볼을 포함할 수 있으나 실시예를 적용한 바이너리 크기의 증가치는 크지 않다. 특히, 실제의 파일 크기의 규모가 클수록 변조 공격 방지 동작 적용에 의한 증가율은 낮다. 이에 따라 실시예를 적용한 호출 프로그램 및 라이브러리 모듈의 파일 크기 증가는 허용할 수 있는 수준임을 확인할 수 있다.

[0064] 도 11은 일 실시예에 있어서, 프로그램별 서로 다른 기법에 의한 동적 링커의 재배치 회수와 로딩시간을 설명하기 위한 도면이다.

[0065] 도 11은 전체/부분(Full/Partial) Relro에서 재배치 엔트리 및 로딩 시간을 변경하는 것을 설명하기 위한 도면이다. 실시예에 따른 CFI 기반의 GOT 변조 공격 방지 방법이 적용되었을 때 공격자가 무력화 혹은 우회할 수 있는 가능성에 대해 분석될 수 있다. 도8의 cmpl 명령코드 참조하면, GOT 보호 장치는 호출 원점에 하드 코딩된 4바이트의 분기 식별자와 호출 목표점에서 0x9 바이트 떨어진 곳에 위치한 4바이트의 분기 식별자를 비교함으로써 분기의 유효성 검사를 수행할 수 있다. 공격자는 프로세스의 메모리 공간에서 분기 식별자와 동일한 값

을 포함한 코드를 찾아 -0x9 지점의 주소로 GOT 엔트리 값을 변조한다면 분기 식별자 검사가 성공하여 해당 주소로 제어흐름을 분기시킬 수 있다. MD5 해시값의 충돌 가능성을 고려할 때, 해시값과 동일한 연속된 4바이트가 우연히 존재하고, 더욱이 -0x9 오프셋 지점부터 시작하는 주소 영역이 실행 가능한 공격명령 코드의 집합으로 구성되어 있을 가능성은 사실상 없다.

[0066] 일반적인 DEP 환경에서 공격자는 코드 삽입 공격이 불가능하다. 실시예에 따른 CFI 기반의 GOT 변조 공격 방지 방법의 보안성은 프로세스 공간에 호출 라이브러리 함수와 동일한 심볼을 가진 또 다른 함수가 배치될 수 있는 확률과 해당 함수의 공격 코드로의 활용 가능성에 의존한다. 지역 연결 속성 함수는 분기 식별자 및 분기 엔트리 생성 대상이 아니므로 고려대상에서 제외된다. 또한, 외부 연결 속성의 함수는 중복될 경우 링크 과정에서 실패하므로 고려되지 않는다. 상기에서 언급한 바와 같이, 약한 연결 속성의 함수는 프로세스 공간에 중복적으로 배치될 수 있다. 그러나, 이 함수 집합은 약한 연결 속성의 특성상 유사한 기능을 수행할 가능성이 높고, 공격자에 의해 그룹 내 다른 함수로 분기되더라도 최적화의 차이만 있을 뿐, 실행 결과의 차이는 없어 공격 코드로 사용될 수 있는 가능성은 희박하다. binutils-gdb 에 포함된 프로그램 중 프로세스 공간 코드 영역의 크기가 가장 큰 *gdb* 에 대해 함수 심볼 및 분기 식별자 중첩 가능성이 분석될 수 있다. *gdb*는 도10 에 포함된 binutils-gdb 프로그램 그룹이 의존하는 라이브러리 중 *liblto_plugin.so*를 제외한 모든 라이브러리를 로딩할 수 있다. *gdb* 가 의존하는 13개의 라이브러리와 이 라이브러리에 포함된 모든 외부 및 약한 연결 속성의 함수 9,151개의 함수 심볼을 분석한 결과 동일한 함수 심볼은 발견되지 않았고, MD5 분기 식별자의 해시값 충돌의 경우도 나타나지 않음을 확인할 수 있다.

[0067] 도 11을 참고하면, binutils-gdb 프로그램 그룹에 대해 Full Relro를 적용한 그룹과 Partial Relro 와 실시예에서 제안된 CFI 기반의 GOT 변조 공격 방지 방법을 함께 적용한 그룹의 로딩 및 실행 성능을 비교한 것이다. Full Relro 기법은 현재 실행파일의 GOT 보호를 위해서 가장 널리 구현되고 있는 기법이고, Partial Relro와 비교함에 따라 *.got.plt* 섹션을 읽기전용 속성으로 변경하여 GOT 변조 공격을 차단할 수 있다. 이때, 실행파일에 대해서는 Full Relro 기법과 실시예에서 제안된 CFI 기반의 GOT변조 공격 방지 방법은 보호 대상이 동일하다. 성능적인 측면에서 Full Relro의 경우 프로그램 로딩시간에 프로그램 내의 라이브러리 함수 심볼에 대한 일괄적 바인딩으로 로딩 지연이 발생될 수 있다. 실시예에서 제안된 CFI 기반의 GOT변조 공격 방지 방법의 경우 Partial Relro 적용으로 지연 바인딩에 의한 빠른 로딩 효과를 얻을 수 있지만, 실행시간에 동적 바인딩 오버헤드 및 분기 유효성 검사에 의한 시간 지연이 발생할 수 있다. 예를 들면, 'LD_DEBUG=statistics' 동적 링커의 옵션을 사용하여 프로그램 별 동적 링커에서 수행한 재배치 회수(number of relocations)와 총 시작 시간("Total startup time in dynamic loader")이 측정될 수 있다. 측정 시 캐싱 효과를 없애기 위하여 'echo 3 > /proc/sys/vm/drop_caches' 명령이 사용될 수 있고, 측정 시간 편차를 줄이기 위해 100 차례 로딩 시간의 평균이 측정될 수 있다.

[0068] 도 11을 참고하면, 프로그램 별 두 기법에 의한 동적 링커의 재배치 회수와 로딩시간을 나타낸 것으로, 도 11(a)에 도시된 바와 같이, 주소 재배치 회수는 함수 심볼뿐만 아니라 전역 데이터 심볼 및 모든 재배치를 포함할 수 있다. Full Relro에서 증가한 재배치 회수가 로딩 시간의 지연을 가져오는 함수 심볼에 대한 재배치 회수이다. 도 11(b)에 도시된 바와 같이, 적은 숫자의 함수 심볼 재배치 회수는 상당한 시간 지연이 초래될 수 있다. binutils-gdb 프로그램 그룹의 작은 실행 파일 크기를 고려할 때, 큰 규모의 프로그램에서는 로딩 시간 오버헤드가 더욱 클 것으로 추정된다.

[0069] 실시예에서 제안된 CFI 기반의 GOT 변조 공격 방지 방법을 적용하면 함수 호출시 실행시간 동적 바인딩에 의한 시간 지연과 분기 식별자 검사 시간에 의한 성능 오버헤드가 발생될 수 있다. 동적 바인딩에 소요되는 전체 시간은 호출되는 함수의 개수에 의존하고, 프로그램의 실행환경에 따라 달라질 수 있다. 실행 프로그램에 포함된 라이브러리 함수 중 실제 호출되는 비율은 일반적으로 낮은 것으로 알려져 있다. 또한, 각 호출은 프로그램의 실행 중 분산되어 발생하므로, 개별 동적 바인딩 시간은 체감되지 않고 측정이 쉽지 않다. 분기 유효성 검사에서는 세 개의 명령코드가 추가 실행되는데, 함수 호출시 발생하는 문맥 전환(context switching) 오버헤드를 고려할 때 미치는 영향은 미미하다. 이에 따라, 실시예에서는 Full Relro에서 잃게 되는 지연된 바인딩에 의한 로딩 성능 향상 효과를 획득할 수 있다.

[0070] 한편, 호출 프로그램과 라이브러리에 함께 변조 공격 방지 방법이 적용될 수 있다. 그러나, 프로그램 실행환경에 따라 호출 프로그램이나 라이브러리 중 한쪽에만 CFI 기반의 GOT 변조 공격 방지 방법이 적용된 채로 호출이 발생할 수 있다. 호출 프로그램에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않은 경우 실행 흐름은 분기 식별자 검사없이 기존의 심볼 바인딩 과정과 함수 호출 과정을 수행할 수 있다. 이 경우, 라이브러리의 심볼 테이블은 점프 테이블의 엔트리 주소를 가리키게 되므로, 호출 원점에서는 목표 함수가 아닌 점프 테이블로

분기될 수 있다. 따라서, 한 번의 직접 분기 오버헤드가 발생할 수 있다. 반대로, 라이브러리 모듈에 CFI 기반의 GOT 변조 공격 방지 방법이 적용되지 않은 경우 호출시 매번 분기 식별자 검사가 실패하고 동적 링커에 의한 심볼 바인딩이 수행될 수 있다. 이 경우 자주 호출되는 함수일 경우 성능 하락이 발생할 수 있다. 다시 말해서, 모듈 간 호출에서 한쪽만 변조 공격 방지 기법이 적용된 경우 성능 오버헤드의 우려는 있지만 프로그램 기능상의 변화는 없다.

[0071] 다시 말해서, GOT 보호 장치는 분기 유효성 검사 코드 생성시 라이브러리의 CFI 기반의 GOT 변조 공격 방지 동작의 적용 여부를 확인하여 심볼 별로 변조 공격 방지 방법을 적용할 수 있다. 실시예에 따르면, 성능 오버헤드를 최소화하여 점진적인 빌드가 가능하고, 기존의 라이브러리와도 높은 호환성을 가진다.

[0072] 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.

[0073] 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.

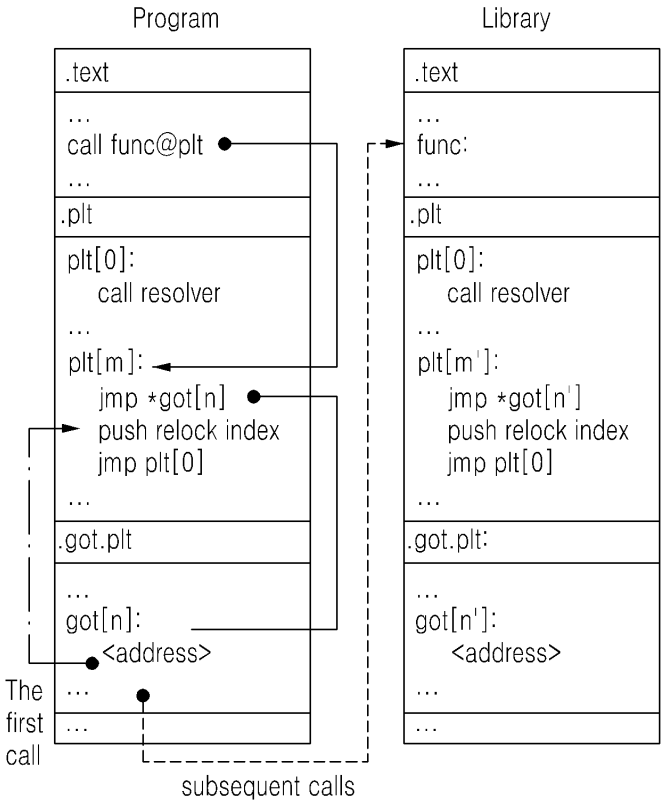
[0074] 실시예에 따른 방법은 다양한 컴퓨터 수단을 통하여 수행될 수 있는 프로그램 명령 형태로 구현되어 컴퓨터 판독 가능 매체에 기록될 수 있다. 상기 컴퓨터 판독 가능 매체는 프로그램 명령, 데이터 파일, 데이터 구조 등을 단독으로 또는 조합하여 포함할 수 있다. 상기 매체에 기록되는 프로그램 명령은 실시예를 위하여 특별히 설계되고 구성된 것들이거나 컴퓨터 소프트웨어 당업자에게 공지되어 사용 가능한 것일 수도 있다. 컴퓨터 판독 가능 기록 매체의 예에는 하드 디스크, 플로피 디스크 및 자기 테이프와 같은 자기 매체(magnetic 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.

[0075] 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.

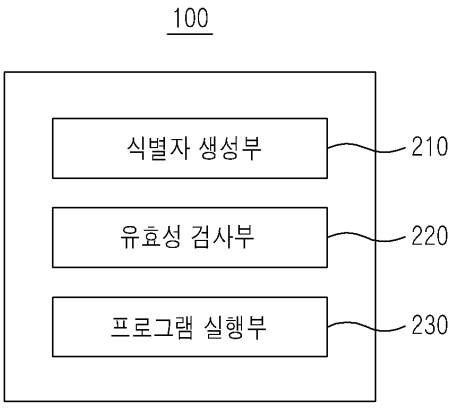
[0076] 그러므로, 다른 구현들, 다른 실시예들 및 특허청구범위와 균등한 것들도 후술하는 특허청구범위의 범위에 속한다.

도면

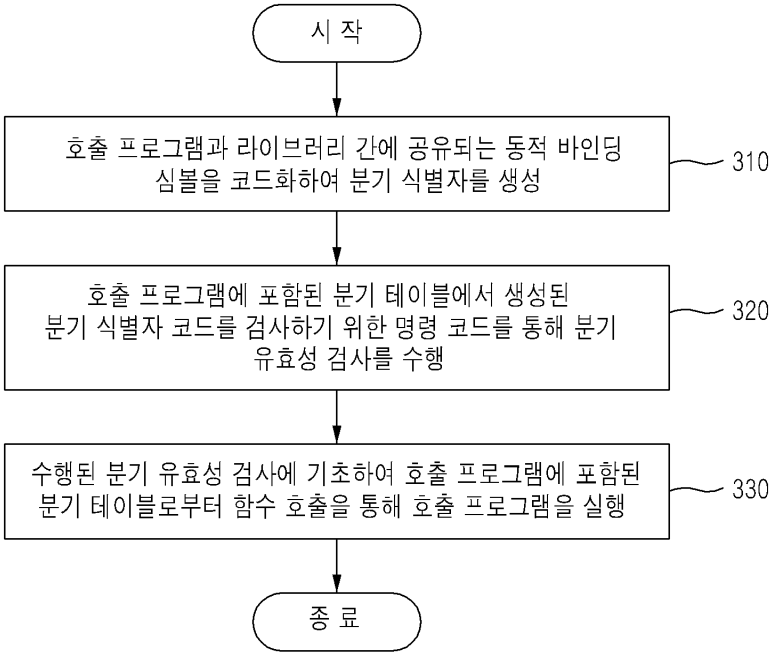
도면1


Program
Library
.text
…
call func@plt
…
.plt
plt[0]:
call resolver
…
plt[m]:
jmp *got[n]
push relock index
jmp plt[0]
…
.got.plt
…
got[n]:
<address>
…
.text
…
func:
…
.plt
plt[0]:
call resolver
…
plt[m']:
jmp *got[n']
push relock index
jmp plt[0]
…
.got.plt:
…
got[n']:
<address>
…
The first call
subsequent calls

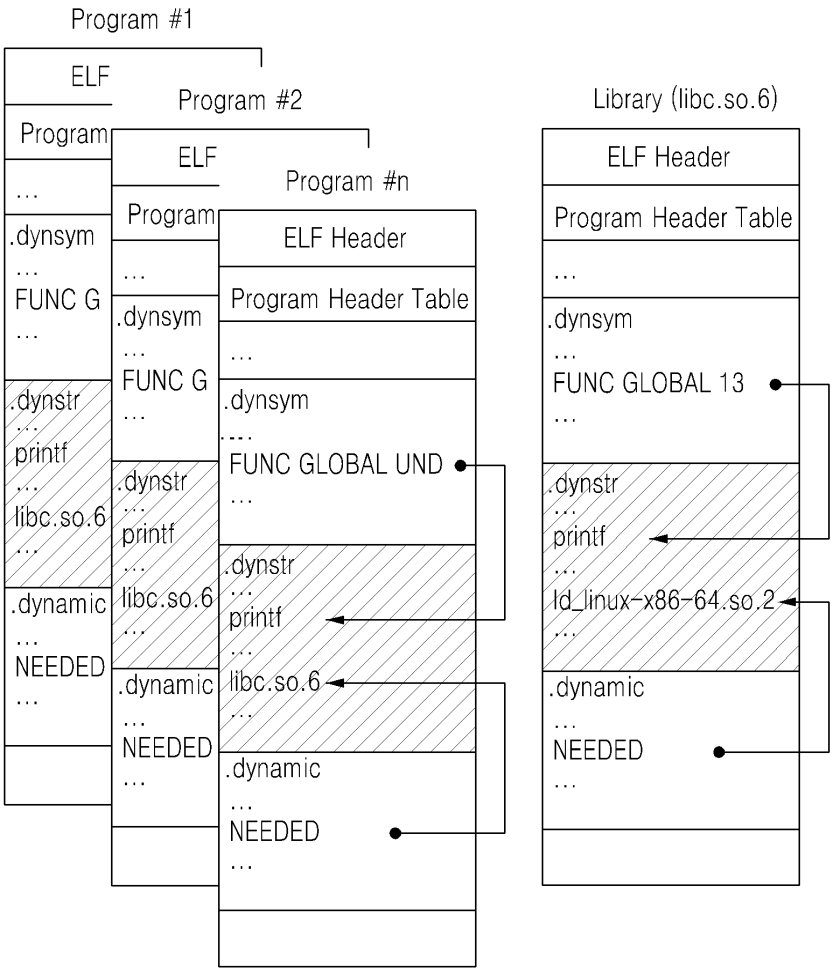
도면2


100
식별자 생성부
210
유효성 검사부
220
프로그램 실행부
230

도면3


시 작
호출 프로그램과 라이브러리 간에 공유되는 동적 바인딩 심볼을 코드화하여 분기 식별자를 생성
310
호출 프로그램에 포함된 분기 테이블에서 생성된 분기 식별자 코드를 검사하기 위한 명령 코드를 통해 분기 유효성 검사를 수행
320
수행된 분기 유효성 검사에 기초하여 호출 프로그램에 포함된 분기 테이블로부터 함수 호출을 통해 호출 프로그램을 실행
330
종 료

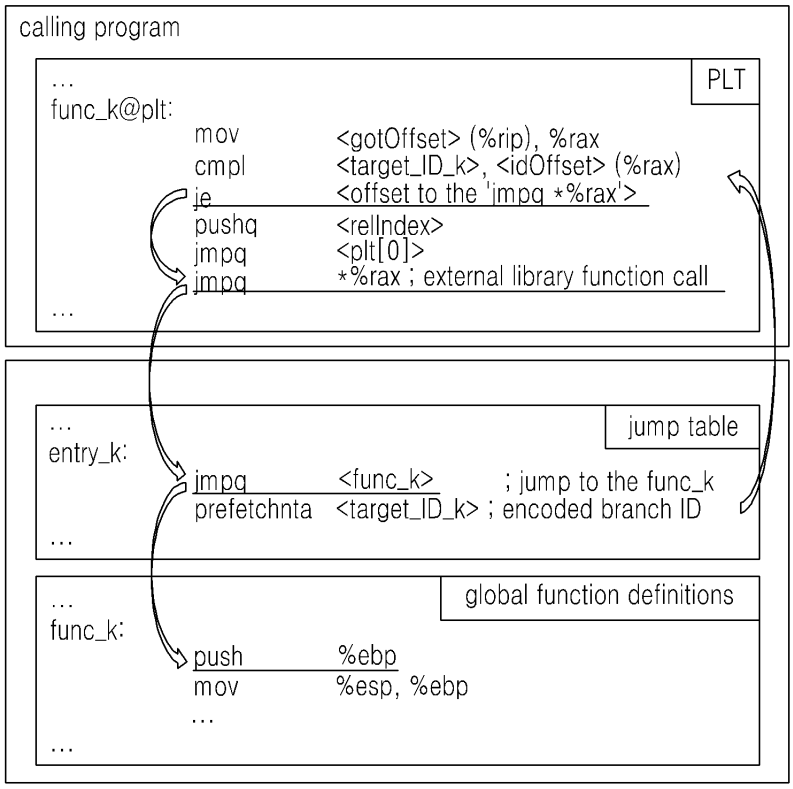
도면4


Program #1
ELF
Program
...
.dynsym
...
FUNC G
...
.dynstr
...
printf
...
libc.so.6
...
.dynamic
...
NEEDED
...
Program #2
ELF
Program
...
.dynsym
...
FUNC G
...
.dynstr
...
printf
...
libc.so.6
...
.dynamic
...
NEEDED
...
Program #n
ELF Header
Program Header Table
...
.dynsym
...
FUNC GLOBAL UND
...
.dynstr
...
printf
...
libc.so.6
...
.dynamic
...
NEEDED
...
Library (libc.so.6)
ELF Header
Program Header Table
...
.dynsym
...
FUNC GLOBAL 13
...
.dynstr
...
printf
...
ld_linux-x86-64.so.2
...
.dynamic
...
NEEDED
...

도면5

```
function writePlt(got,plt, relindex)
Input GOT, PLT, and a relocation entry index
Output PLT entry assembly instructions

symbol ← get the function symbol using plt and relindex
module ←  get the library module using symbol

gotOffset ← get an instruction offset value to the GOT entry using got and plt
targetID ← get MD5 hash value using symbol
idOffset ← get the constatnt targetIDoffset from the branch target

if module does NOT has a jump table for the branch validation then
        instuctions ← {
                jmpq        *<gotOffset>(%rip)
                pushq       <relindex>
                jmpq        <plt[0]>
        }
else
        instuctions ← {
                mov         <gotOffset>(%rip), %rax
                cmpl        <targetID>, <idOffset> (%rax)
                je          <skip to the indirect jump instruction>
                pushq       <relindex>
                jmpq        <plt[0]>
                jmpq        *%rax    ;external library function call
        }
end

return (instructions)
```

도면6

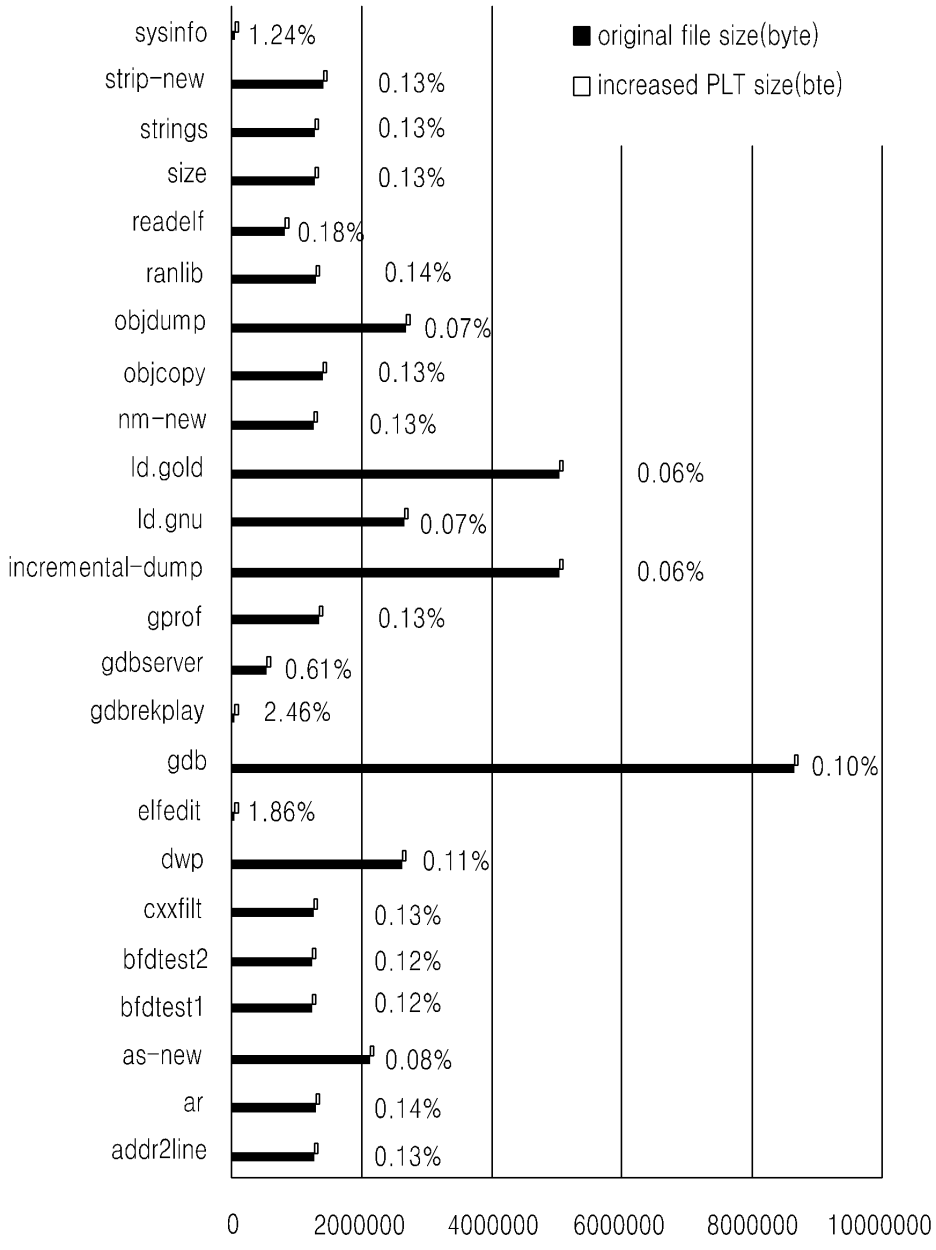

calling program
PLT
...
func_k@plt:
mov <gotOffset> (%rip), %rax
cmpl <target_ID_k>, <idOffset> (%rax)
je <offset to the 'jmpq *%rax'>
pushq <relIndex>
jmpq <plt[0]>
jmpq *%rax ; external library function call
...
jump table
...
entry_k:
jmpq <func_k> ; jump to the func_k
prefetchnta <target_ID_k> ; encoded branch ID
...
global function definitions
...
func_k:
push %ebp
mov %esp, %ebp
...
...

도면7

```
#include <stdlib.h>

__attribute__((visibility("default")))  int foo() {
    return 42;
}

__attribute__((visibility("default"))) void *alloc_memory(size_t sz) {
    void *p = (void *) malloc(sz);
    return p;
}
```

```
0000000000001120 <foo@@Base>:
   1120:    e9 bb ff ff ff          jmpq   10e0 <foo@@Base-0x40>
   1125:    0f 18 04 25 ac bd 18 db prefecthnta 0xffffffffdb18bdac
   112d:    cc                      int3
   112e:    cc                      int3
   112f:    cc                      int3

0000000000001130 <alloc_memory@@Base>:
   1130:    e9 bb ff ff ff          jmpq   10e0 <foo@@Base-0x30>
   1135:    0f 18 04 25 27 cc d0 2d prefecthnta 0x2dd0cc27
   113d:    cc                      int3
   113e:    cc                      int3
   113f:    cc                      int3
```

도면8

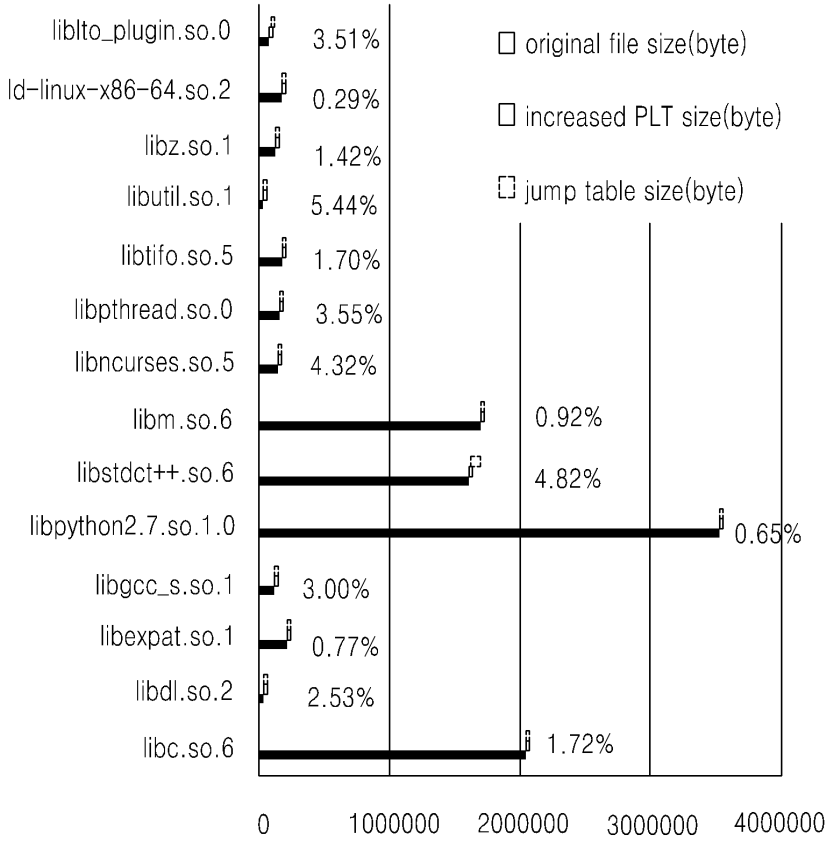

original file size(byte)
increased PLT size(bte)
sysinfo
strip-new
strings
size
readelf
ranlib
objdump
objcopy
nm-new
ld.gold
ld.gnu
incremental-dump
gprof
gdbserver
gdbrekplay
gdb
elfedit
dwp
cxxfilt
bfdtest2
bfdtest1
as-new
ar
addr2line
1.24%
0.13%
0.13%
0.13%
0.18%
0.14%
0.07%
0.13%
0.13%
0.06%
0.07%
0.06%
0.13%
0.61%
2.46%
0.10%
1.86%
0.11%
0.13%
0.12%
0.12%
0.08%
0.14%
0.13%
0
2000000
4000000
6000000
8000000
10000000

도면9

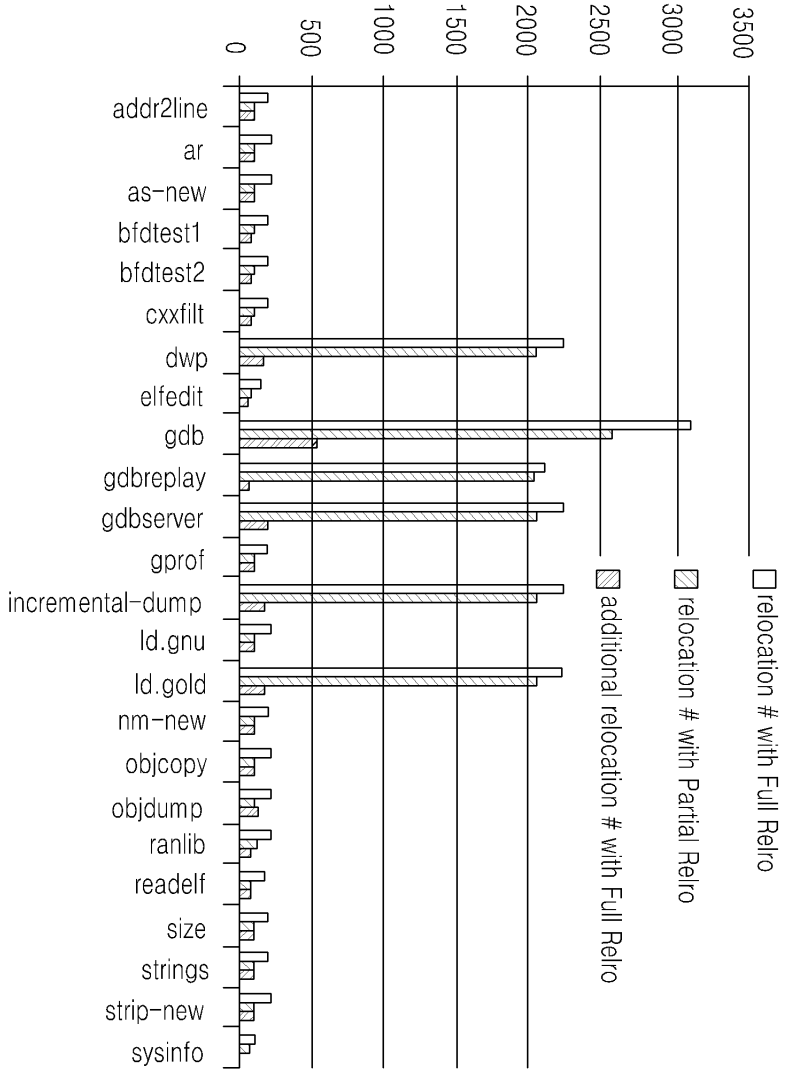
```
#include <stdio.h>
#include <stdlib.h>

extern inf foo();
extern void *alloc_memory(size_t);

int main(int argc, char *argv[]) {
    int ret = foo();
    void *p = alloc_memory( ret );

    printf("Allocated %d bytes at %pWn", ret, p);
    free(p);

    return 0;
}
```

```
0000000000001270 <.plt+40>:
    1270: 48 8b 05 b0 1d 00 00   mov    0x1db9(%rip),%rax      # 3030 <foo@Base>
    1277: 81 78 09 ac bd 18 db   cmpl   $0xdb18bdac,0x9(%rax)
    127e: 74 0a                  je     128a <_fini+0x72>
    1280: 68 01 00 00 00         pushq  $0x1
    1285: e9 a6 ff ff ff         jmpq   1230 <_fini+0x18>
    128a: ff e0                  jmpq   *%rax
    128c: cc                     int3
    128d: cc                     int3
    128e: cc                     int3
    128f: cc                     int3
    1290: 48 8b 05 a1 1d 00 00   mov    0x1da1(%rip),%rax   # 3038 <alloc_memory@Base>
    1297: 81 78 09 27 cc d0 2d   cmpl   $0x2dd0cc27,0x9(%rax)
    129e: 74 0a                  je     12aa <_fini+0x92>
    12a0: 68 02 00 00 00         pushq  $0x2
    12a5: e9 86 ff ff ff         jmpq   1230 <_fini+0x18>
    12aa: ff e0                  jmpq   *%rax
    12ac: cc                     int3
    12ad: cc                     int3
    12ae: cc                     int3
    12af: cc                     int3
```

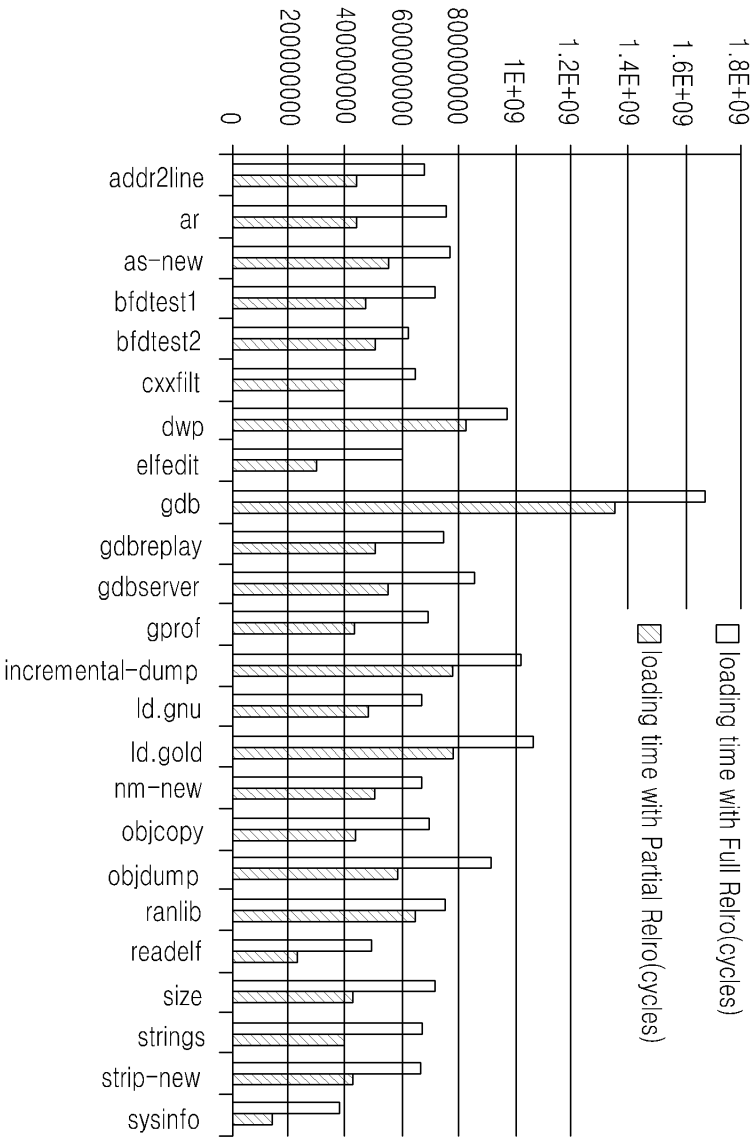
도면10


liblto_plugin.so.0
ld-linux-x86-64.so.2
libz.so.1
libutil.so.1
libtifo.so.5
libpthread.so.0
libncurses.so.5
libm.so.6
libstdct++.so.6
libpython2.7.so.1.0
libgcc_s.so.1
libexpat.so.1
libdl.so.2
libc.so.6
3.51%
0.29%
1.42%
5.44%
1.70%
3.55%
4.32%
0.92%
4.82%
0.65%
3.00%
0.77%
2.53%
1.72%
original file size(byte)
increased PLT size(byte)
jump table size(byte)
0
1000000
2000000
3000000
4000000

도면11a


0
500
1000
1500
2000
2500
3000
3500
addr2line
ar
as-new
bfdtest1
bfdtest2
cxxfilt
dwp
elfedit
gdb
gdbreplay
gdbserver
gprof
incremental-dump
ld.gnu
ld.gold
nm-new
objcopy
objdump
ranlib
readelf
size
strings
strip-new
sysinfo
additional relocation # with Full Relro
relocation # with Partial Relro
relocation # with Full Relro

도면11b


0
200000000
400000000
600000000
800000000
1E+09
1.2E+09
1.4E+09
1.6E+09
1.8E+09
addr2line
ar
as-new
bfdtest1
bfdtest2
cxxfilt
dwp
elfedit
gdb
gdbreplay
gdbserver
gprof
incremental-dump
ld.gnu
ld.gold
nm-new
objcopy
objdump
ranlib
readelf
size
strings
strip-new
sysinfo
loading time with Full Relro(cycles)
loading time with Partial Relro(cycles)