J. Internet Comput. Serv.
ISSN 1598-0170 (Print) / ISSN 2287-1136 (Online)
http://www.jics.or.kr
Copyright ⓒ 2020 KSII

CFI(Control Flow Integrity) 적용을 통한 GOT(Global Offset Table) 변조 공격 방지 방안 연구☆

CFI Approach to Defend against GOT Overwrite Attacks

정 승 훈[1] 황 재 준[2] 권 혁 진[3] 신 동 규[1*]
Seunghoon Jeong Jaejoon Hwang Hyukjin Kwon Dongkyoo Shin

요 약

유닉스 계열 시스템 환경에서 GOT 변조(GOT overwrite) 공격은 소프트웨어 권한 탈취를 위한 전통적인 제어흐름 탈취 기법 중 하나이다. 그 동안 GOT 변조를 방어하기 위한 몇 가지 기법들이 제안되었는데, 그 중 프로그램 로딩 단계에서 GOT 영역을 읽기전용 속성으로 메모리 배치하여 실행 시간에 GOT 변조를 원천적으로 차단하는 Full Relro(Relocation Read only) 기법이 가장 효과적인 방어 기법으로 알려져 왔다. 하지만, Full Relro 기법은 로딩 시간의 지연을 가져와 시작 성능에 민감한 프로그램의 적용에는 제약이 있고, 라이브러리에 적용시 의존 라이브러리에 의한 연쇄적인 로딩 지연 문제 등으로 라이브러리에는 현재 적용되지 않고 있다. 또한, LLVM을 포함한 다수의 컴파일러들은 Full Relro 기법을 기본 적용하지 않아 실행환경의 프로그램은 GOT 공격에 여전히 취약하다. 이 논문에서는 현재 코드 재사용 공격 방어를 위해 가장 적합한 기법으로 인식되고 있는 CFI(Control Flow Integrity) 기법을 사용한 GOT 보호 장치를 제안한다. LLVM을 기반으로 본 기법을 구현하고 binutils-gdb 프로그램 그룹에 적용해 보안성, 성능, 호환성 등을 평가하였다. 본 CFI 기반 GOT 보호 장치는 우회하기 어렵고, 빠르며 기존 라이브러리 프로그램과도 호환되어 적용가능성이 높다.

☞ 주제어 : 보안, 제어흐름 무결성, 링킹과 로딩, GOT/PLT

ABSTRACT

In the Unix-like system environment, the GOT overwrite attack is one of the traditional control flow hijacking techniques for exploiting software privileges. Several techniques have been proposed to defend against the GOT overwrite attack, and among them, the Full Relro(Relocation Read only) technique, which blocks GOT overwrites at runtime by arranging the GOT section as read-only in the program startup, has been known as the most effective defense technique. However, it entails loading delay, which limits its application to a program sensitive to startup performance, and it is not currently applied to the library due to problems including a chain loading delay problem caused by nested library dependency. Also, many compilers, including LLVM, do not apply the Full Relro technique by default, so runtime programs are still vulnerable to GOT attacks. In this paper, we propose a GOT protection scheme using the Control Flow Integrity(CFI) technique, which is currently recognized as the most suitable technique for defense against code reuse attacks. We implemented this scheme based on LLVM and applied it to the binutils-gdb program group to evaluate security, performance and compatibility. The GOT protection scheme with CFI is difficult to bypass, fast, and compatible with existing library programs.

☞ keyword : control flow integrity, GOT/PLT, linking and loading, security

1. 서 론

"CWS/SANS Top 25 most dangerous software errors" [1]는 소프트웨어 취약점을 유발하는 가장 중요한 소프트웨어 오류를 지적하고 있다. 그 중 사용자의 입력값 검사 누락에 따른 버퍼 오버플로우가 가장 위험한 오류로 지적되고 있다. 제어흐름탈취 공격이란, 버퍼 오버플로우 등의 메모리 오류 취약점을 악용하여 공격자가 의도한 프로그램 코드 주소로 프로그램의 실행흐름을 바꾸어 프로그램의 제어권을 탈취하는 행위를 일컫는다. GOT(Global Offset Table) 변조공격 [2] 은 유닉스 계열 시스템 환경에

1 Department of Computer Engineering, Sejong University, 209, Neungdong-ro, Gwangjin-gu, Seoul, 05006, South Korea
2 Hanhwa Systems, 188 Pangyoyeok-ro, Bundang-gu, Seongnam-si, Gyeonggi-do, 13524, South Korea
3 Ministry of National Defense, 22 Itaewon-ro, Yongsan-gu, Seoul 04383, South Korea
* Corresponding author (shindk@sejong.ac.kr)
[Received 14 November 2019, Reviewed 21 November 2019, Accepted 10 December 2019]
☆ 이 논문은 2018년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임 (No.2018R1D1A1B07047395)

http://dx.doi.org/10.7472/jksii.2020.21.1.179

서 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 명령코드는 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 공격 기법이다.

그 동안 GOT 변조를 방어하기 위한 몇 가지 기법들이 제안되었는데 그 중 대표적인 기법이 바로 Full Relro (RELocation Read Only) 기법이다. Relro 기법은 프로그램 로딩시 동적 링커가 사용하는 ELF 구조체의 주요 섹션 데이터를 읽기전용으로 설정하여 실행시간 변조 공격을 차단한다. Relro 기법의 종류는 *.got.plt*(이하 GOT) 섹션을 제외한 *.ctors*, *.dtors*, *.dynamic* 섹션 등을 보호하는 Partial Relro와 GOT 섹션까지 모두 보호하는 Full Relro가 있다. Full Relro가 적용된 프로그램은 로딩 시간에 모든 호출 라이브러리 함수의 바인딩이 수행된 후 GOT 가 읽기전용 상태가 되어 실행시간 GOT 변조가 차단된다. 표 1은 Ubuntu 최신 버전에 탑재된 보안 기능 중 일부를 발췌한 것으로 Ubuntu 18.04 LTS 버전부터는 X86-64(amd64) 시스템에도 gcc 컴파일러 도구를 통해 Full Relro가 기본 적용되었다.

하지만, Full Relro 기법은 시작 시간 지연이라는 성능 문제를 포함한다. 동적 바인딩은 의존 라이브러리 목록 검색, 라이브러리 함수명의 문자열 비교 등 시간 소요 작업을 필요로 한다 [3]. 다수의 라이브러리가 포함된 프로그램에서, 라이브러리 목록의 뒤에 배치된 라이브러리 함수에 대한 다량의 호출이 발생할 때 심볼 바인딩에 의한 로딩시간은 라이브러리 개수의 제곱에 비례해 느려질 수 있다 [4]. 라이브러리의 경우 일반적인 실행흐름에서 호출되지 않는 다량의 함수를 포함할 수 있고, 지연된 바인딩 없이는 의존 라이브러리에 의한 연쇄적인 로딩 지연 등의 문제가 발생할 수 있어 Full Relro 기법은 적용되지 않는다. 최신 운영체제가 아닌 환경에서는 Partial Relro 설정으로 빌드된 프로그램들이 배포되고, LLVM (Low Level Virtual Machine) [5] 을 포함한 다수의 컴파일러들은 여전히 Partial Relro 가 기본 설정이다. 따라서, 실행 환경의 많은 프로그램들은 여전히 GOT 공격에 노출되어 있다.

(표 1) 우분투의 Relro 보안 기능
(Table 1) Ubuntu security feature for Relro

Ubuntu Version	Built with Relro	Built with BIND_NOW
16.04 LTS (Xenial Xerus)	Gcc patch	gcc patch (s390x), package list for others
18.04 LTS (Bionic Beaver)	Gcc patch	gcc patch (amd64, ppc64el, s390x), package list for others
19.04 (Disco Dingo)	Gcc patch	gcc patch (amd64, ppc64el, s390x), package list for others
19.10 (Eoan)	Gcc patch	gcc patch (amd64, ppc64el, s390x), package list for others

한편, 코드 재사용 공격 [6] 이 제어흐름탈취 공격의 주류가 된 현재, 제어흐름 무결성(CFI: Control Flow Integrity) 점검 기법은 코드 재사용 공격 방어를 위한 효과적인 기법으로 활발한 연구가 이루어져 왔다. CFI 가 적용된 운영체제가 출시되고 [7], 일부 컴파일러에는 CFI 빌드 기능이 탑재되었다 [8]. CFI 기법이 최초 제안된 이후 안전한 모듈간 제어흐름 전이를 위한 CFI 관련 연구 [9, 10]도 진행되었다. 하지만, 약한 공격 모델을 가정한 방어 기법은 쉽게 우회되고, 실시간 코드 변환을 필요로 하는 감시 기법은 성능, 호환성의 문제로 사용되지 않는다. 그리고 현재의 LLVM 컴파일러에서 지원하는 모듈간 CFI 기법에서도, PLT를 경유하는 제어흐름 전이는 Full Relro기법에 의존하고 있어 GOT는 CFI의 보호 범위에서 벗어나 있다.

본 논문에서는 CFI(Control Flow Integrity) 기법이 사용된 모듈 간 제어흐름 보호를 위한 GOT 보호 장치를 제안한다. 호출 프로그램과 라이브러리 간에 동적 바인딩되는 함수 심볼을 분기 식별자로 사용하고, 호출 프로그램은 분기 식별자를 검사하는 코드를 포함하며, 라이브러리에는 분기 식별자를 담은 분기 테이블이 포함된다. 시험 도구는 X86-64 시스템의 Ubuntu 18.04 LTS 환경에서 LLVM 10 기반으로 구현되었고, *binutils_gdb* 프로그램 그룹에 대한 적용을 통해 보안성, 성능, 호환성 등을 평가했다. 본 기법의 GOT 보호 장치는 우회하기 어렵고, 빠르며 기존 라이브러리와도 호환되어 적용가능성이 높다. 본 논문의 기여점은 다음과 같다.

- GOT 보호를 위한 CFI 기반 보호장치 제안

- X86-64 기반의 GOT 보호 시험 도구 구현
- binutils_gdb 프로그램 그룹에 대한 제안 기법 평가

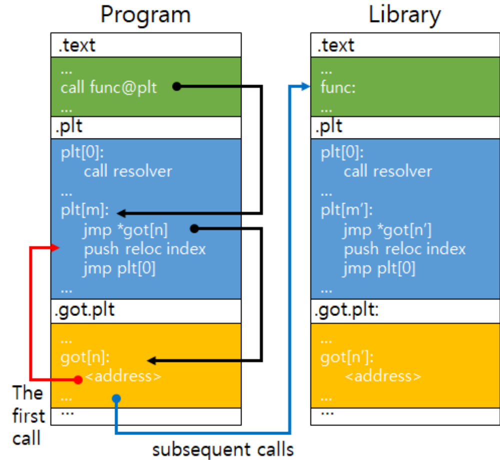
본 논문은 2장 연구배경, 3장 CFI 를 적용한 GOT 보호 장치 설계, 4장 구현, 5장 평가, 6장 토의사항, 7장 결론으로 구성된다.

2. 연구배경

제어흐름 무결성(CFI) [11] 기법은 프로그램의 실행 시간에 정상적인 실행흐름에서 벗어난 분기를 제한하고자 한다. CFI 기법에서 보호하고자 하는 대상은 간접함수 호출(indirect function call), 간접 분기(indirect jump), 함수 복귀(function return) 의 명령문이 사용된 실행 분기들이다 [11, 12]. 직접 호출(direct call), 직접 분기(direct jump) 명령문의 경우 컴파일 시간에 호출 원점과 목표 분기점과의 오프셋 값이 고정되어 메모리 주소 참조를 필요로 하지 않는 직접 분기 명령 코드로 생성된다. Data Execution Prevention(DEP) [13] 페이지 설정에 의해 공격자의 실행 시간에 명령 코드의 변조는 불가능하므로, CFI 기법은 간접 분기를 보호 대상으로 한다.

CFI 기법은 정상 제어흐름 그래프를 생성하는 분석단계와 실행 시간에 정상 제어흐름 그래프 하에서 제어흐름 분기가 이루어지도록 분기 유효성 검사 코드를 삽입하는 단계로 이루어진다. 분석단계에서는 간접 분기 명령 코드를 찾고, 각 분기 원점(branch source) 별로 가능한 분기 목표점(branch target) 그룹을 식별하여 제어흐름 그래프를 생성한다. 분기 검사 지점은 앞서 언급한 간접 호출이 발생하는 분기 원점이다. 일반적으로 정적분석의 정확성에는 한계가 있어, CFI는 분기 원점 별로 분기 목표 그룹을 보수적으로 생성한다. 따라서, 제어흐름 그래프는 일반적으로 과근사(overapproximation)하게 생성된다. 제어흐름 분기 그룹의 크기가 커질수록 도달 가능한 목표 분기점이 늘어나므로 CFI 보안성은 저하될 수 있다 [12].

코드 삽입단계에서는 생성된 제어흐름그래프에 따라 분기 유효성 검사 명령코드를 삽입한다. 점검방식은 CFI 기법 별로 상이할 수 있는데 대부분의 CFI 기법들은 호출 원점에서 정적으로 결정되는 분기 식별자를 검사하여 일치하면 분기를 허용하고, 그렇지 않으면 실행을 비정상 종료한다. 그래서 실행시간 공격자에 의해 분기 목표 주소가 변조되면, 프로그램 실행은 중단되고 공격은 실패하게 된다 [11, 12]. 분기 식별자는 CFI 구현마다 달라질 수 있는데 컴파일러에서 분기 별로 고유한 식별자를 생성할 수도 있고, 호출 함수의 형식 정보를 사용한 식별자를 생성할 수도 있다.



(그림 1) 동적 라이브러리 함수 호출을 위한 섹션들
(Figure 1) Sections for calling dynamic library functions

동적 라이브러리 링크는 호출 프로그램과 라이브러리 간의 함수에 대한 심볼 바인딩 프로세스가 실행 시간 동적으로 이루어지도록 하는 기법이다. 정적 링커과 동적 링커는 서로 협력하여 모듈 간 심볼 바인딩이 이루어지도록 한다. 프로그램 생성시 정적 링커는 각 라이브러리 함수 호출에 대해 포인터를 사용한 간접 호출 명령코드를 생성하고, 동적 링커는 실행 시간에 해당 포인터를 메모리에 로딩된 라이브러리 함수의 실제 주소값으로 채운다. 호출 프로그램은 의존 관계에 있는 또 다른 라이브러리를 호출하는 라이브러리일 수 있다.

Unix 계열 정적 링커는 동적 라이브러리 링크를 지원하기 위해 PLT, GOT 두 구조체를 생성하여 호출 프로그램의 ELF 파일의 섹션으로 포함시킨다(그림 1). 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 수정한다. PLT 엔트리에는 짝을 이루고 있는 GOT 엔트리의 값을 참조하여 목표 분기점 주소로 간접 호출을 실행하는 명령 코드가 포함된다. GOT는 호출 프로그램에

동적으로 바인딩되는 라이브러리 함수들의 실행시간 주소 값을 담는 포인터 배열 구조체로서 앞서 언급했듯이 동적 링커에 의해 주소 값이 저장된다.

3. CFI를 적용한 GOT 보호 장치 설계

본 논문에서 제안하는 CFI 보호 기법은 일반적인 리눅스 방어 기법이 동작하는 실행 환경과 강력한 공격 모델을 가정한다. 실행 프로그램 및 라이브러리 파일은 ELF 파일을 대상으로 하며, 메모리 페이지는 DEP 에 의해 실행과 동시에 쓰기 가능은 하지 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/GOT를 통해 이루어진다. 공격자는 프로그램 혹은 라이브러리의 GOT 테이블 엔트리 변조를 통한 제어흐름탈취 공격을 시도한다. 위의 프로그램 실행 환경에서 본 연구는 모듈 간 호출에 대해 GOT 변조 공격 방지 장치를 제안한다. PLT를 경유하지 않고 함수의 포인터를 사용한 간접호출이나, dlopen(), dlsym() 함수를 사용한 동적 라이브러리 로딩에 의한 함수 호출은 본 논문에서 제안하는 보호 범위에서 벗어난다.

3.1 동적 바인딩 심볼을 사용한 정적 분기 식별자 공유

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 사용한다. 동일한 라이브러리 함수를 호출하는 서로 다른 프로그램은 각각의 ELF 파일의 '*.dynamic*' 섹션에 동일한 라이브러리 파일에 대한 엔트리와, '*.dynsym*' 섹션에 동일한 함수 심볼에 대한 엔트리를 포함한다(그림 2). 동적 바인딩 함수 심볼은 PLT 엔트리에서 분기 식별자로 사용될 수 있는데 각 연결된 GOT 엔트리에 바인딩되는 함수 심볼은 고유하기 때문이다. 참고로, 정적 분석시 다른 간접 호출 분기점에서는 호출 함수의 심볼이 일반적으로 고유하게 결정되지는 않는다.

동적 바인딩 함수 심볼은 작은 크기의 분기 식별자 그룹의 요건을 충족한다. 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. 그러나 약한 연결 속성의 함수 심볼이 보안성 측면에서 미치는 영향은 제한적이고 이는 뒤에 오는 '평가' 절에서 다시 언급한다. 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(동적 심볼 테이블에 존재하지 않음) 동적 바인딩 함수 심볼은 고유한 분기 식별자 선택을 위한 좋은 후보가 될 수 있다.

동적 바인딩 함수 심볼을 CFI의 분기 식별자로 사용하기 위해서는 코드화가 필요하다. 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. 따라서, 본 CFI 기법의 분기 식별자로 사용하기 위해 동적 바인딩 심볼은 다음의 코드화 과정을 거친다. 해시함수를 사용하여 하나의 명령 코드에 로딩할 수 있는 고정된 길이의 비트코드를 생성하되 해시함수의 입력이 되는 문자열은 이름 꾸밈(name mangling) [14] 규칙에 의해 전처리된 심볼 이름을 사용한다. 컴파일러에 의해 이름 꾸밈된 심볼은 동적 링커가 바인딩시 사용하는 문자열이고, 중복성이 없기 때문에 본 CFI 적용단계에서는 식별자 생성을 위한 별도의 이름 꾸밈이 필요하지 않다.

3.2 호출 프로그램 PLT 엔트리의 분기 식별자 점검

CFI 기법이 모듈간 적용으로 확장될 때 분기 유효성 검사방식은 달라질 수 있다. 모듈내의 일반적인 CFI의 분기 유효성 검사는 분기 원점에서 분기 목표점의 분기 식별자를 직접 비교하거나, 분기 주소가 포함되어야 할

유효한 오프셋 범위를 점검하는 명령 코드를 실행함으로서 수행된다. 그러나, 모듈 간 범위에서는 분기 식별자의 중복 가능성에 의해 분기 식별 그룹이 커질 수 있고, 라이브러리의 경우 유효한 오프셋 범위가 동적으로 달라져 별도의 유효성 검사함수의 수행이 필요할 수 있다. 이러한 이유로 기존의 모듈 간 CFI 기법에서는 유효성 검사를 위한 동적인 테이블 관리 및 함수 호출이 필요한데 이는 적지 않은 성능 오버헤드를 유발하는 문제가 있다. 특히, LLVM cross-DSO처럼 분기 원점이 아닌 라이브러리의 분기 목표점에서 분기 유효성 검사 함수가 수행되어야 보안성이 유지될 수 있는 경우 분기 유효성 검사는 더욱 복잡해진다.

본 CFI 기법에서는 전역 함수 심볼을 분기 식별자로 사용하기 때문에 분기원점에서 단순한 비교 명령문 실행을 통해 분기 유효성 검사를 수행할 수 있다. 그림 3은 분기 목표 모듈에 분기 식별자 테이블이 존재할 경우 분기 유효성 검사 명령코드를 추가하는 링커의 의사 알고리즘을 보여준다. GOT 엔트리가 가리키는 분기 목표점에 위치한 분기 식별자 명령코드를 로딩하고, 값을 비교하여 일치하면 분기를 정상적으로 진행하고, 그렇지 않으면 동적 바인딩을 통해 GOT 엔트리 값을 재설정한다. 동적 바인딩은 동적 링커에 의한 지연된 바인딩(Lazy Binding or Lazy Symbol Resolution) 장치를 수정 없이 사용한다. 즉, 재배치 테이블의 엔트리 순번을 스택에 저장하고, PLT[0] 로 분기하여 동적 링커가 실행되도록 한다. 그림 3의 'je' 분기문 다음에 오는 두 명령코드는 동적 바인딩의 시작을 보여준다.

PLT 엔트리의 분기 유효성 검사는 단순하면서도 효과적이다. PLT 엔트리의 명령 코드는 GOT 엔트리 값을 참조하는 무조건 분기에서 분기 식별자 검사가 포함된 조건부 분기로 변경이 되는데 세 개의 명령 코드만 추가되었다. 식별자 검사가 실패하는 경우는 첫 호출 때와 공격자에 의해 GOT 엔트리가 변조 됐을 때이다 - 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 수정된다. 공격자는 분기 유효성 검사를 우회하기 어렵다. 앞서 언급한 강력한 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. 따라서, GOT 엔트리 변조에 성공하더라도 PLT 엔트리의 분기 유효성 검사에서

```
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 constant targetID offset 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)
```

(그림 3) 링커의 PLT 생성 의사 코드

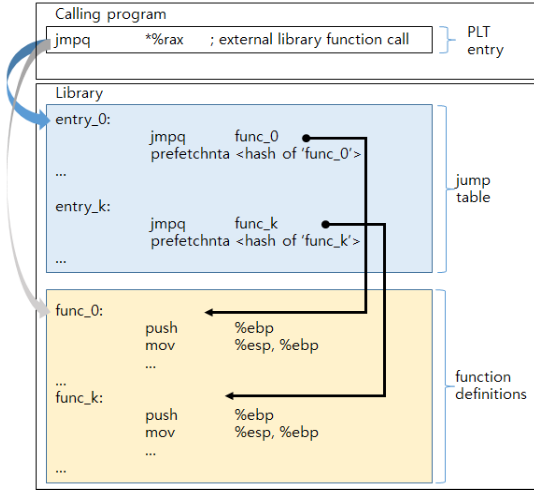
(Figure 3) PLT generation pseudo code by the linker

실패하고, 변조된 GOT 엔트리는 동적 링커에 의해 올바른 함수 주소로 갱신되어 GOT 변조에 의한 제어흐름탈취 공격은 실패한다.

3.3 라이브러리 점프 테이블 생성

PLT 엔트리에서의 분기 유효성 검사는 라이브러리 코드에 분기 식별자 코드 삽입을 필요로 한다. 라이브러리 함수에 분기 식별자 코드를 직접 삽입하는 방법은 함수의 오프셋 변화를 가져와 추가적인 코드 주소의 재배치 및 호환성 문제를 일으킬 수 있다. 따라서, 라이브러리 내 기존 코드 섹션의 수정이 없도록, 분기 식별자 검사를 위한 별도의 점프 테이블을 생성한다. 점프 테이블은 라이브러리의 외부 인터페이스가 되어 점프 테이블의 각 엔트리는 라이브러리 함수 호출 진입점이 되고, 라이브러리 내부에 정의된 각 함수와 연결된다.

그림 4는 분기 테이블 엔트리와 라이브러리 함수의 호출 연결 관계를 개념적으로 도시한다. 점프 테이블의 각 엔트리 코드는 짝을 이루는 라이브러리 함수로의 직접 분기 명령코드와 해당 함수 심볼의 분기 식별자 코드를 포함하는 명령코드로 구성된다. 점프 테이블 전체 명령코드는 하나의 독립된 함수로 생성하여 기존 라이브러리 내에 추가적인 주소 재배치가 없도록 한다. 또한, 호출 프로그램이 라이브러리 함수 호출시 점프 테이블 엔트리



(그림 4) 라이브러리에 포함된 점프 테이블
(Figure 4) Jump tables included in the library

```
#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       prefetchnta 0xffffffffdb18bdac
    112d:      cc                            int3
    112e:      cc                            int3
    112f:      cc                            int3

0000000000001130   <alloc_memory@@Base>:
    1130:      e9 bb ff ff ff                jmpq    10f0  <foo@@Base-0x30>
    1135:      0f 18 04 25 27 cc d0 2d       prefetchnta 0x2dd0cc27
    113d:      cc                            int3
    113e:      cc                            int3
    113f:      cc                            int3
```

(그림 5) 라이브러리 점프 테이블의 예제
(Figure 5) Example of a library jump table

주소가 바인딩 될 수 있도록 정적 링커는 라이브러리 함수의 심볼 주소가 점프 테이블 엔트리를 가리키도록 변경한다.

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

4. 구 현

CFI에 의한 GOT 보호 기법은 LLVM 10 버전의 Module Pass 프레임워크 [15] 및 LLD 링커 프로젝트 [16]를 기반으로 구현되었고, X86-64 아키텍처를 대상으로 한다. 라이브러리의 점프 테이블 생성 코드는 Module Pass에 기반하여 LTO(Link Time Optimization) [17] 라이브러리(LLVMgold.so)의 일부로 구현되어 LLD 링커의 플러그인으로 입력된다. 호출 프로그램의 분기 유효성 검사는 LLD 링커의 PLT 생성 코드를 수정하여 구현되었다.

4.1 라이브러리

LLVM은 '*Module*', '*Function*', '*Basic Block*' 단위의 코드 분석 및 변환을 위한 *Pass* 프레임워크를 제공하는데 본 기법은 *Module Pass* 단계에서 구현되었다. *Module Pass* 단계에서는 입력 파일 단위의 분석 및 코드 최적화가 수행된다. 구현된 Pass는 모듈 내 지역 연결 속성이 아닌 정의된 함수들을 목록화하고 모듈 별로 하나의 점프 테이블을 구성한 후 각 함수의 시작점으로 분기하는 엔트리 함수를 생성한다. 점프 테이블은 LLVM 의 함수 형식 검증 기반 CFI 구현에 사용된 점프 테이블과 유사한 방식으로 구성된다. 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. LLVM CFI의 점프 테이블과는 달리 엔트리 별 분기 명령 코드 다음에는 분기 식별자가 인코딩된 명령코드를 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 변환한 값을 사용한다. 이 값은 분기 식별자를 인코딩하기 위해 사용된 명령코드(prefetchnta instruction)의 하위 4바이트에 배치된다. 'prefetchnta' 는 8바이트로 구성된 X86-64 명령 코드로서, 캐시에 명령코드를 미리 읽어오는 기능을 수행하며 실행시 기능적 부작용(side effect)은 없다. 그림 5는 2개의 전역 함수가 포함된 라이브러리 예제 소스코드와 [18] 생성된 점프 테이블 코드를 *objdump* 명령으로 확인한 결과이다. 각 점프 테이블 엔트리는 16바이트로 구성되고 'prefetchnta' 명령코드 다음에 오는 3개의 'int3' 명령코드

는 16바이트 정렬을 위한 패딩 바이트이다.

모듈 별 점프 테이블 생성은 링크 시간에 코드 최적화를 수행하는 LTO 단계에서 이루어진다. LTO 단계의 코드 변환은 LLVM IR 비트코드(bitcode) [19] 형식의 입력 모듈을 요구하므로, 본 구현에서는 Clang 10 컴파일러에 *-flto* 컴파일 옵션을 사용하여 링커의 입력 모듈을 컴파일하였다. 비트코드가 아닌 오브젝트 모듈은 링킹시 LTO 단계에서 코드 변환이 수행되지 않아 해당 함수들은 생성 라이브러리의 점프 테이블 엔트리에 포함되지 않는다. 본 기법은 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.

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

extern int 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 %p₩n", ret, p);
    free(p);

    return 0;
}

0000000000001270  <plt+40>:
   1270:        48 8b 05 b9 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                    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                    jmpq   1230 <_fini+0x18>
   12aa:        ff e0                       jmpq   *%rax
   12ac:        cc                          int3
   12ad:        cc                          int3
   12ae:        cc                          int3
   12af:        cc                          int3
```

(그림 6) 호출 프로그램의 PLT 예제

(Figure 6) PLT example of the calling program

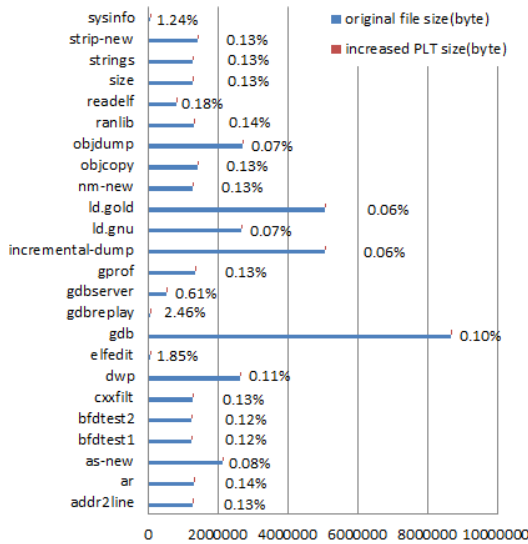
4.2 호출 프로그램

호출 프로그램에서의 분기 유효성 검증은 LLVM의 링커 프로젝트인 LLD 의 PLT 생성 코드를 수정하여 구현되었다. 각 PLT 엔트리 별로 분기 유효성을 검사하기 위한 명령 코드가 생성된다. 분기 식별자는 참조할 GOT 엔트리와 연결된 함수 심볼로부터 점프 테이블 엔트리에서와 같은 방식으로 MD5 해시 값을 취하여 구한다. 분기 식별자 검사 명령코드는 GOT 엔트리가 가리키는 주소로부터 9바이트('jmp' 명령코드 5바이트 + 'prefetchnta' 명령코드내의 분기식별자 오프셋 4바이트) 떨어진 곳에 위치한 4바이트의 값을 분기 식별자와 비교한다. 그림 6은 그림 5의 라이브러리 함수를 호출하는 예제 프로그램 [18]에 대해 본 기법을 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 명령코드가 실행된다. 각 PLT 엔트리는 32 바이트로 구성되고, 'jmpq' 명령코드 다음에 오는 4개의 'int3' 명령코드는 32 바이트 정렬을 위한 패딩 바이트이다.

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. 본 기법이 적용된 PLT 엔트리에서는 분기 유효성 검사가 실패하면 'pushq' 명령 코드로 분기하여 동적 링킹을 수행하므로 GOT 엔트리 값 초기화는 별도로 필요하지 않다. 한편, 보호 기법이 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, 본 기법은 PLT 엔트리 별로 선택적인 분기 유효성 검사가 가능하도록 구현되었다. 링커는 PLT 엔트리 생성 시 라이브러리 함수가 GOT 보호 기법이 적용되어 분기 테이블을 포함하고 있는지를 판단할 수 있는데, 라이브러리의 지원 여부에 따라 분기 유효성 검사 코드 추가 여부를 결정할 수 있다. 본 구현에서는 구현의 단순함을 위해 라이브러리의 경로에 특정 문자열이 포함된 경우에 한해 해당 함수 호출시 분기 유효성 검사 코드를 추가하도록 하였다.

5. 평 가

본 기법의 효용성 검증을 위해 binutils-gdb 프로그램 그룹을 대상으로 증가된 파일 사이즈의 크기를 측정하고, 보안성, 성능, 호환성을 평가하였다. binutils-gdb 프로그램 그룹은 리눅스에서 활용도가 높고, 복잡성과 규모면에서 시험 평가에 적절한 다종의 프로그램을 포함하고 있다. 평가에는 AMD Ryzen 3700X CPU, Ubuntu 18.04 LTS 환경에서 binutils-gdb 2.33 버전이 사용되었다.



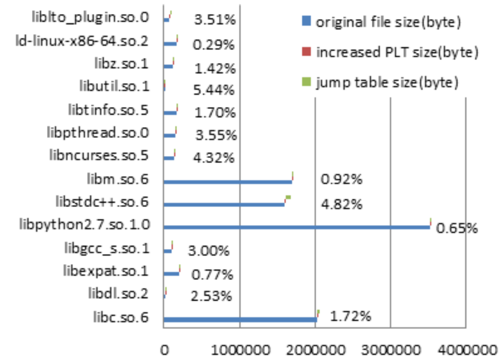
(그림 7) binutils-gdb 프로그램 별 파일크기 변화
(Figure 7) File size increment of the binutils-gdb program

(표 2) '*libinproctrace.so*' 라이브러리 파일크기 변화
(Table 2) File size increment of the library '*libinproctrace.so*'

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

5.1. 파일 크기

본 기법에 의해 호출 프로그램에서는 분기 유효성 검사 코드가 추가되고, 라이브러리에서는 분기테이블이 생성되므로 모듈의 바이너리 크기가 증가한다. 호출 프로그램에서는 X86-64 의 경우 원래 PLT의 각 엔트리 크기는 16바이트이나 본 기법 적용시 PLT 엔트리 크기는 32바이트로 증가한다(그림 6 참조). 따라서, 호출 프로그램의 바이너리 파일은 {PLT 엔트리 개수*16바이트} 만큼 크기가 증가할 수 있다. 그림 7은 Partial Relro 로 컴파일 후 디버그 정보가 제거된 binutils-gdb 프로그램 그룹에 대해 실행파일 별로 증가된 파일 크기 정보를 보여준다. 실제 바이너리 크기 증가량은 PLT 크기 증가량과 일치



(그림 8) binutils-gdb 호출 라이브러리의 파일크기 변화
(Figure 8) File size increment of libraries called by binutils-gdb program

하지는 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. 따라서, 호출 프로그램의 바이너리 크기 증가율은 그림 7에서 보는 것처럼 낮은 수준이다.

또한, 라이브러리에는 정의된 전역 함수 별로 분기테이블 엔트리가 생성된다. 각 점프 테이블 엔트리는 5바이트의 무조건 분기 명령코드와 분기 식별자가 인코딩된 8바이트의 명령코드 및 16바이트 정렬을 위한 3바이트의 패딩으로 구성된다(그림 5 참조). 따라서, 라이브러리의 경우 {호출 함수 개수*16바이트} 크기의 점프 테이블이 포함되어 파일 크기는 증가하며, 라이브러리가 또 다른 의존 라이브러리를 포함하는 경우 PLT 분기 유효성 검사 코드에 의해 바이너리 크기는 더 커질 수 있다. 표 2는 binutils-gdb 프로그램 그룹에 포함된 동적 라이브러리 *libinproctrace.so* 에 대해 기법 적용 전후 바이너리 크기 정보를 비교한 결과이다. 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 크기 측정 실험을 추가하였다. 그림 8은 binutils-gdb 프로그램 그룹에서 의존하는 동적 라이브러리의 PLT 엔트리 개수와 정의된 전역 함수의 개수를 기반으로 기법 적용에 따른 파일 크기 증가치의 추정값을 보여주고 있다. 실제 파일 크기는 세그먼트 바운더리 정렬 등에 의해 페이지 크기 범위에서 차이날 수 있다. 라이브러리 모듈은 특성상 일반적으로 실행파일에

비해 더 많은 전역 함수 심볼을 포함할 수 있으나 본 기법 적용에 의한 바이너리 크기의 증가치는 크지 않다. 특히 원래 파일 크기의 규모가 클수록 기법 적용에 의한 증가율은 낮다. 따라서, 본 기법 적용에 의한 호출 프로그램 및 라이브러리 모듈의 파일 크기 증가는 허용할 수 있는 수준이다.

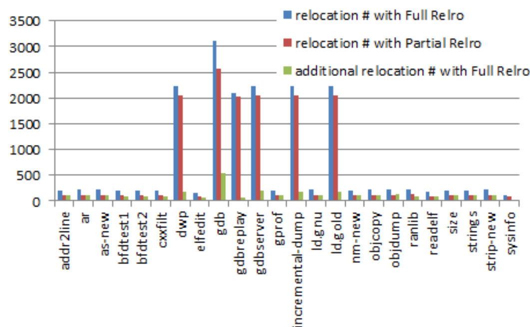
5.2. 보안성

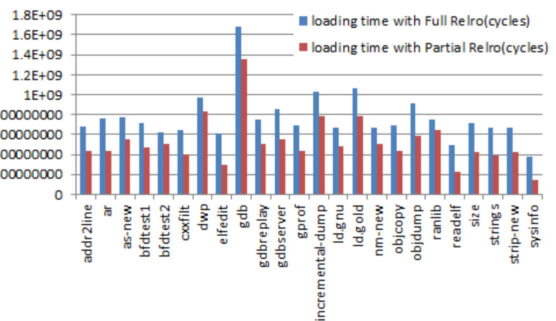
보안성 평가에서는 본 기법이 적용되었을 때 공격자가 본 기법을 무력화 혹은 우회할 수 있는 가능성에 대해 분석하였다. 분기의 유효성 검사는 호출 원점에 하드 코딩된 4바이트의 분기 식별자와 호출 목표점에서 0x9 바이트 떨어진 곳에 위치한 4바이트의 분기식별자를 비교함으로서 수행된다(그림 6의 cmpl 명령코드 참조). 공격자는 프로세스의 메모리 공간에서 분기 식별자와 동일한 값을 포함한 코드를 찾아 -0x9 지점의 주소로 GOT 엔트리 값을 변조한다면 분기 식별자 검사가 성공하여 해당 주소로 제어흐름을 분기시킬 수 있다. MD5 해시값의 충돌 가능성을 고려할 때, 해시값과 동일한 연속된 4바이트가 우연히 존재하고, 더욱이 -0x9 오프셋 지점부터 시작하는 주소 영역이 실행가능한 공격명령 코드의 집합으로 구성되어 있을 가능성은 사실상 없다.

일반적인 DEP 환경에서 공격자는 코드 삽입 공격이 불가능하므로, 본 기법의 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. 앞서 언급했듯이 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 가능성을 분석하였다. 참고로, *gdb*는 그림 8에 포함된 binutils-gdb 프로그램 그룹이 의존하는 라이브러리 중 *liblto_plugin.so*를 제외한 모든 라이브러리를 로딩한다. *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 않았다.

5.3 성능

binutils-gdb 프로그램 그룹에 대해 Full Relro를 적용한 그룹과 Partial Relro 와 본 기법을 함께 적용한 그룹의 로딩 및 실행 성능을 비교하였다. Full Relro 기법은 현재 실행파일의 GOT 보호를 위해서 가장 널리 구현되고 있는 기법이고, Partial Relro와 비교해 *.got.plt* 섹션을 읽기 전용 속성으로 변경하여 GOT 변조 공격을 차단한다. 따라서, 실행파일에 대해서는 Full Relro 기법과 본 기법은 보호 대상이 동일하다. 성능적인 측면에서 Full Relro의 경우 프로그램 로딩시간에 프로그램 내의 라이브러리 함수 심볼에 대한 일괄적 바인딩으로 로딩 지연이 발생한





(그림 9) Full/Partial Relro 적용에 따른 재배치 엔트리 개수 변화 및 로딩 성능 변화
(Figure 9) Changes in the number of relocation entries and loading time according to Full/Partial Relro

다. 본 기법의 경우 Partial Relro 적용으로 지연 바인딩에 의한 빠른 로딩 효과를 얻을 수 있지만, 실행시간에 동적 바인딩 오버헤드 및 분기 유효성 검사에 의한 시간 지연이 발생한다. '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")을 측정하였다. 측정 시 캐싱 효과를 없애기 위해 'echo 3 > /proc/sys/vm/drop_caches' 명령을 사용했고, 측정 시간 편차를 줄이기 위해 100 차례 로딩 시간의 평균을 측정하였다.

그림 9는 프로그램 별 두 기법에 의한 동적 링커의 재배치 회수와 로딩시간을 보여준다. 왼쪽 그림에서 주소 재배치 회수는 함수 심볼 뿐만 아니라 전역 데이터 심볼 및 모든 재배치를 포함한다. 따라서, Full Relro에서 증가한 재배치 회수가 로딩 시간의 지연을 가져오는 함수 심볼에 대한 재배치 회수이다. 우측 그림에서 보듯이, 적은 숫자의 함수 심볼 재배치 회수는 상당한 시간 지연을 초래한다. binutils-gdb 프로그램 그룹의 작은 실행 파일 크기를 고려할 때, 큰 규모의 프로그램에서는 로딩 시간 오버헤드가 더욱 클 것으로 추정된다.

본 기법을 적용하면 함수 호출시 실행시간 동적 바인딩에 의한 시간 지연과 분기 식별자 검사 시간에 의한 성능 오버헤드가 발생한다. 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 [20]. 또한, 각 호출은 프로그램의 실행 중 분산되어 발생하므로, 개별 동적 바인딩 시간은 체감되지 하고 측정이 쉽지 않다. 분기 유효성 검사에서는 세 개의 명령코드가 추가 실행되는 데, 함수 호출시 발생하는 문맥 전환(context switching) 오버헤드를 고려할 때 미치는 영향은 미미하다. 따라서, 본 기법에서는 Full Relro에서 잃게 되는 지연된 바인딩에 의한 로딩 성능 향상 효과를 얻을 수 있다.

5.4 호환성

프로그램 실행환경에 따라 호출 프로그램이나 라이브러리 중 한쪽에만 보호 기법이 적용된 채로 호출이 발생할 수 있다. 호출 프로그램에 보호 기법이 적용되지 않은 경우 실행 흐름은 분기 식별자 검사없이 기존의 심볼 바인딩 과정과 함수 호출 과정을 거친다. 이 경우, 라이브러리의 심볼 테이블은 점프 테이블의 엔트리 주소를 가리키게 되므로, 호출 원점에서는 목표 함수가 아닌 점프 테이블로 분기한다. 따라서, 한 번의 직접 분기 오버헤드가 발생한다. 반대로, 라이브러리 모듈에 보호 기법이 적용되지 않은 경우 호출시 매번 분기 식별자 검사가 실패하고 동적 링커에 의한 심볼 바인딩이 수행된다. 이 경우 자주 호출되는 함수일 경우 성능 하락이 발생할 수 있다. 정리하면, 모듈 간 호출에서 한쪽만 보호 기법이 적용된 경우 성능 오버헤드의 우려는 있지만 프로그램 기능상의 변화는 없다. 하지만, 성능적인 측면에서 호출 프로그램과 라이브러리에 보호 기법이 함께 적용되는 것이 바람직하여, 앞서 언급했듯이 본 기법은 이를 위한 장치를 제공한다. 즉, 분기 유효성 검사 코드 생성시 라이브러리의 기법 적용 여부를 확인하여 심볼 별로 보호기법을 적용할 수 있다. 따라서, 본 기법은 성능 오버헤드를 최소화하여 점진적인 빌드가 가능하고, 기존의 라이브러리와도 높은 호환성을 가진다.

6. 토의사항

본 기법의 보호 범위는 아니지만, 모듈간의 함수 호출은 PLT를 경유하지 않고도 가능하다. 함수의 포인터를 사용한 간접호출, *dlopen()*, *dlsym()* API 함수를 사용한 라이브러리 동적 로딩 및 함수 호출은 PLT를 경유하지 않는다. 하지만, 함수의 심볼이 고유하게 결정되는 호출점에서는 본 기법을 확장하여 적용할 수 있다. 이를 위해서는 호출 함수의 심볼을 찾기 위한 추가적인 정적 분석이 필요하다. 전자의 경우 포인터 앨리어스 분석을 사용해 참조되는 함수 심볼을 구하고, 후자의 경우 *dlsym()* 호출의 파라미터로 입력되는 함수 심볼을 구한다. 함수 심볼로부터 분기 식별자가 생성되고, 함수 포인터가 호출되는 위치에 분기 유효성 검사 코드가 추가된다. 본 기법에 의한 라이브러리 점프 테이블은 호출 함수에 대한 분기 식별자를 이미 포함하고 있으므로, 본 기법의 확장 적용을 위한 코드 수정은 호출 프로그램에서만 필요하다. 이러한 확장된 보호 기법에 대한 구현과 평가는 추후 연구 과제로 남겨둔다.

7. 결　론

본 논문에서는 CFI를 적용한 GOT 변조 방지 방안에 대해 기술하였다. CFI 기법은 런타임 검증 기술 중의 하나로서 보호 대상 프로그램 코드에 내장되어 실행시간

비정상적인 제어흐름을 효과적으로 감지한다. 특히, DEP, ASLR(Address Space Layout Randomization) [21] 등 운영체제 수준의 보호 기술을 우회하는 지능적인 공격에 더욱 효과적일 수 있다. 또한, 런타임 검증 기술은 필요에 따라 코드에 선택적으로 적용이 가능한 특징이 있다. 무기체계 소프트웨어는 높은 보안성과 실행 성능 조건을 함께 요구하여 보안 기술 적용이 쉽지 않다. 따라서, 보안성과 성능의 유연성에서 장점이 큰 런타임 검증 기술은 무기체계 소프트웨어의 보안성을 제고할 수 있는 핵심 기술이 될 수 있을 것이다.

참고문헌(Reference)

[1] MITRE. CWE/SANS Top 25 Most Dangerous Software Errors. [Online]. http://cwe.mitre.org/top25/archive/2019/2019_cwe_top25.html

[2] c0ntex. How to hijack the global offset table with pointers for root shells. [Online]. http://www.infosecwriters.com/text_resources/pdf/GOT_Hijack.pdf

[3] M. Zhang and R. Sekar, "Squeezing the dynamic loader for fun and profit," 2015. [Online]. http://www.seclab.cs.stonybrook.edu/seclab/pubs/seclab15-12.pdf

[4] P. Pluzhnikov. Dynamic linking with large number of dsos degrades into linear lookup. [Online]. https://sourceware.org/bugzilla/show_bug.cgi?id=16709

[5] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, IEEE Computer Society, p. 75, 2004. https://doi.org/10.1109/cgo.2004.1281665

[6] H. Shacham et al., "The geometry of innocent flesh on the bone: returninto-libc without function calls (on the x86)," in ACM conference on Computer and communications security, New York,, pp. 552–561, 2007. https://doi.org/10.1145/1315245.1315313

[7] Microsoft. Control flow guard. [Online]. https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard

[8] The Clang Team. Control flow integrity. [Online]. https://clang.llvm.org/docs/ControlFlowIntegrity.html

[9] M. Zhang and R. Sekar, "Control Flow Integrity for COTS Binaries," in USENIX Security Symposium, 2013, pp. 337–352. Available: https://doi.org/10.1145/2818000.2818016

[10] B. Niu and G. Tan, "Modular control-flow integrity," ACM SIGPLAN Notices, vol. 49, no. 6, pp. 577–587, 2014. Available: https://doi.org/10.1145/2666356.2594295

[11] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in Proceedings of the 12th ACM conference on Computer and communications security, ACM, pp. 340–353, 2005. https://doi.org/10.1145/1102120.1102165

[12] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-flow integrity: Precision, security, and performance," ACM Computing Surveys (CSUR), vol. 50, no. 1, p. 16, 2017. https://doi.org/10.1145/3054924

[13] Microsoft Support. Data execution prevention (dep). [Online]. http://support.microsoft.com/kb/875352/EN-US/

[14] F. Kefallonitis, "Name mangling demystified," 2007. [Online]. http://www.int0x80.gr/papers/name_mangling.pdf

[15] LLVM Project. Writing an LLVM Pass. [Online]. http://llvm.org/docs/WritingAnLLVMPass.html

[16] LLVM Project. LLD - The LLVM Linker. [Online]. https://lld.llvm.org/

[17] LLVM Project. LLVM Link Time Optimization: Design and Implementation. [Online]. https://llvm.org/docs/LinkTimeOptimization.html

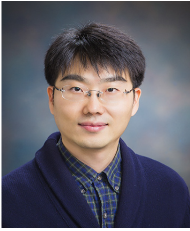
[18] C. Rohlf. Cross dso cfi - llvm and android. [Online]. https://struct.github.io/cross_dso_cfi.html

[19] LLVM Project. LLVM Language Reference Manual. [Online]. https://llvm.org/docs/LangRef.html

[20] L. Presser and J. R. White, "Linkers and loaders," ACM Computing Surveys (CSUR), vol. 4, no. 3, pp. 149–167, 1972. https://doi.org/10.1145/356603.356605

[21] Wikipedia, Address Space Layout Randomization. [Online]. https://en.wikipedia.org/wiki/Address_space_layout_randomization

◐ 저 자 소 개 ◑



정 승 훈(Seunghoon Jeong)
2005년 포항공과대학교 컴퓨터공학과 졸업
2008년 University of California, San Diego 컴퓨터과학과 석사
2013년~현재 국방과학연구소 선임연구원
관심분야 : 네트워크 시스템, 정보보호
E-mail : undukii@gmail.com

황 재 준(Jaejoon Hwang)
1980년 서울대학교 전기공학과 졸업
1982년 서울대학교 제어계측공학과 석사
2004년 한국과학기술원 전산학과 박사
1998년~2019 국방과학연구소 수석연구원
2019년~한화시스템 연구위원
관심분야 : 사이버전, 사이버전자전, 정보보호
E-mail : jjhwang@naver.com

권 혁 진(Hyukjin Kwon)
1989년 성균관대학교 산업공학과 졸업
1991년 성균관대학교 산업공학과 석사
2000년 성균관대학교 산업공학과 박사
1991년~2017 한국국방연구원 책임연구위원
2017년~현재 국방부 정보화기획관
관심분야 : 정보시스템 평가, 정보보호
E-mail : khjsjy2001@hanmail.net

신 동 규(Dongkyoo Shin)
1986년 서울대학교 계산통계학과 졸업
1992년 Illinois Institute of Technology 컴퓨터과학과 석사
1997년 Texas A&M University 컴퓨터공학과 박사
1998년~현재 세종대학교 컴퓨터공학과 정교수
관심분야 : 정보보호, 사이버전, 머신러닝, u-헬스케어
E-mail : shindk@sejong.ac.kr