LLVM CFI 와 비교한 Windows CFG 의 한계점

박상민 [1], 최형기 [2]
[1] 성균관대학교 컴퓨터공학과
parksm0521@gmail.com, meosery@skku.edu

Limitations of Windows CFG compared with LLVM CFI

Sang-min Park[1], Hyung-kee Choi[2]
[1]Dept. of Computer Engineering, SungkyunKwan University

요 약

CFI(Control Flow Integrity)는 제어 흐름을 검증해 프로그램을 보호하는 기법이다. Windows 에서는 CFG(Control Flow Guard)란 이름으로 CFI 를 지원하고 LLVM 에서는 동일하게 CFI 란 이름으로 지원한다. 본 논문에서는 Windows CFG 의 몇 가지 한계점을 LLVM IFCC 와 비교해서 찾아보고 대안책을 제안한다. CFG 에 성능, 확장성, 보안 측면에서 LLVM IFCC 와 비교하여 한계점이 존재한다는 것을 확인하였다. 본 논문에서는 각 항에 대한 이론적 근거를 제시하고 문제를 해결할 수 있는 몇 가지 대응책을 소개한다.

1. 서론.

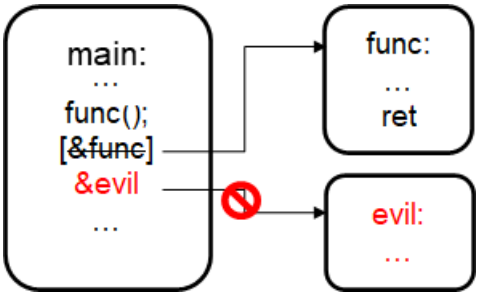
제어 흐름 변조기법은 프로그램을 공격하는 주요 기법 중 하나이다. 공격자는 메모리 오염을 통해 함수 포인터나 복귀주소를 덮어써 프로그램의 제어 흐름을 변조한다. 제어 흐름 변조로부터 프로그램을 보호하는 기법 중 하나로 CFI(Control Flow Integrity)가 개발되었다. CFI 는 컴파일 시 제어 흐름의 일부를 저장하여 프로그램 실행 시 제어 흐름을 비교해 검증하는 기법이다. 제어 흐름이 변경되는 주요 분기점은 함수 호출과 복귀이다. CFI 는 함수 호출을 보호하는 순방향(forward-edge) CFI 와 함수 복귀를 보호하는 역방향(backward-edge) CFI 두 종류로 제어 흐름을 보호한다[1]. CFI 는 운영체제와 컴파일러를 통해 구현된다. Windows 에서는 Control Flow Guard 란 CFI 구현체가 운영체제와 컴파일러를 통해 제공된다. Linux 에서는 CFI 구현체가 GCC 와 LLVM 컴파일러를 통해 제공된다.

본 논문에서는 Windows CFG 와 LLVM CFI 의 순방향 CFI 보호기법에 대해 비교 분석하여 Windows CFG 의 성능, 확장성, 보안 측면에서 한계점을 드러내고 그 대응책을 제시한다.

2. 배경

Windows CFG(이후 CFG)와 LLVM CFI 의 동작구조를 알아보고 각 환경에서 보호받을 수 있는 코드의 범위에 대해 설명한다.

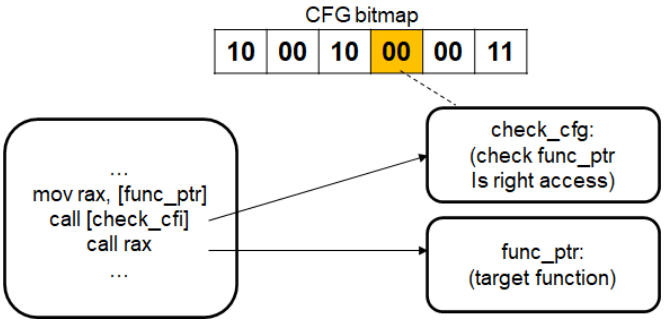
2.1 Control Flow Integrity



(그림 1) Forward-edge CFI

CFI(Control Flow Integrity)는 제어 흐름 변조를 막기 위한 보호기법이다. CFI 는 함수의 호출 흐름을 보호하는 순방향 CFI 와 함수의 복귀 흐름을 보호하는 역방향 CFI 가 있다[1]. 이 논문에서는 순방향(Forward-edge) CFI 만을 다루기 때문에 역방향 CFI 는 언급하지 않는다. 순방향 CFI 는 런타임에 결정되는 목적지 주소를 가진 간접 호출에 대해 보호한다. 그림 1 을 보면 main 에서 func 함수를 호출하는 것을 확인할 수 있다. 공격자가 func 의 제어 흐름을 탈취하여 evil 을 실행하려고 하는데 이런 함수의 호출 탈취를 보호하는 것이 순방향 CFI 이다. 순방향 CFI 를 적용하면 공격자가 메모리 오염 등을 통해 func 의 목적 주소(target address)를 evil 로 변조했을 때 목적 주소를 검증하여 evil 함수의 호출을 허용하지 않는다.

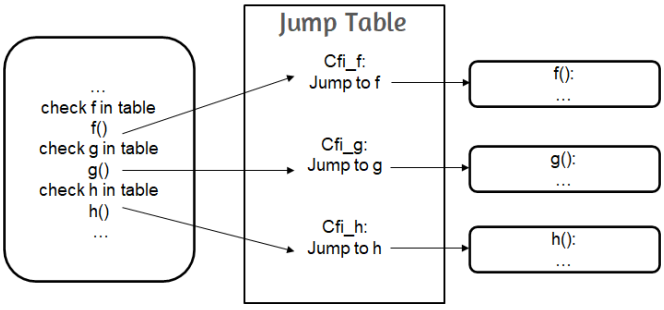
2.2 Windows Control Flow Guard



(그림 2) Windows CFG 동작구조

CFG 는 Windows 운영체제에서 적용되는 순방향 CFI 보호기법이다[2]. CFG 는 컴파일 시 프로그램 내 간접 호출 전에 제어 흐름을 검증하는 시스템 함수를 실행한다. CFG 검증을 하는 시스템 함수를 여기서는 check_cfg 라 칭하겠다. 이를 통해 프로그램 내 모든 간접 함수 호출은 호출되기 직전 제어 흐름이 변조되었는지 확인된다. check_cfg 에서 제어 흐름이 올바른 지 검증할 시 CFG bitmap 을 참고한다. CFG bitmap 은 프로그램의 코드영역 주소에 매핑(mapping) 되는 메모리 공간이다. 프로그램의 코드영역 주소를 인덱스(index)로 CFG bitmap 을 검색하면 해당 주소의 코드영역이 올바른 목적 주소인지를 반환한다. 10, 11 의 경우 올바른 목적 주소이고 01, 00 이면 올바르지 않은 주소이다[4]. check_cfg 는 매개변수로 받은 주소를 통해 CFG bitmap 에서 검색하여 올바른 목적 주소를 호출하고 있는지 검증하게 된다. 그림 2 를 보면 호출하려는 목적 주소인 func_ptr 를 호출하기 이전 func_ptr 의 주소를 매개변수로 check_cfg 를 호출하는 것을 확인할 수 있다. check_cfg 에서는 func_ptr 의 주소를 CFG bitmap 을 통해 검증한다. 올바른 접근일 경우 func_ptr 호출을 정상적으로 수행하고 그렇지 않을 경우 프로그램을 종료한다.

2.3 LLVM CFI



(그림 3) LLVM Indirect Function Call Check

LLVM CFI 에서 순방향 제어 흐름 보호는 간접함수의 호출을 보호하는 IFCC(Indirect Function Call Check)와 클래스의 가상 멤버 함수 호출을 보호하는 NVFCC(Non-Virtual Member Function Call Check)가 있다. NVFCC 의 경우는 이 논문에서는 자세히 다루지 않는다. IFCC 의 경우, 컴파일 시 각 함수로 점프할 수 있는 점프함수의 테이블을 생성한다. 각 함수의 진입은 대응되는 점프함수를 통해서만 할 수 있기 때문에 제어 흐름이 보호된다. 그림 을 보면 함수 f,g,h 의 호출은 각각 그와 대응되는 Cfi_f, Cfi_g, Cfi_h 를 거쳐서 진입하게 된다. 두 보호기법의 구현 코드는 모두 컴파일러에 의해 생성되며 바이너리 코드 내부에 포함된다[3].

3. Windows Control Flow Guard 의 한계점

CFG 의 성능, 확장성, 보안 3 가지 측면에서 한계점을 소개한다.

3.1 성능(Performance)의 한계

<표 1> AMD Ryzen 5 3600 6-Core Processor, Windows 21H2 22000.856 환경에서 VC++2022 를 통해 빌드한 프로그램의 Windows CFG 적용/미적용 시 함수 호출 10,000,000 번의 시간(seconds)

Windows CFG	*Min*	*Max*	*Avg*	*Avg Ratio*
Non-CFI	0.020	0.024	0.020940	1.00
CFI	0.022	0.028	0.023420	1.12

<표 2> AMD Ryzen 5 3600 6-Core Processor, Ubuntu 20.04.4 LTS 환경에서 clang 10.0.0-4 를 통해 빌드한 프로그램의 환경에서 LLVM CFI 적용/미적용 시 함수 호출 10,000,000 번의 시간(seconds)

LLVM Clang	*Min*	*Max*	*Avg*	*Avg Ratio*
Non-CFI	0.017	0.022	0.019433	1.00
CFI	0.020	0.024	0.020726	1.07

CFG 는 LLVM CFI 와 비교해 두가지 성능상 한계점이 존재한다. 첫번째 한계점은 추가적인 함수호출 비용이다. CFG 의 제어 흐름 보호 코드는 Windows 의 시스템 함수 내부에서 실행된다[4]. 즉 제어 흐름 보호를 위해 외부 모듈의 함수를 호출한다. LLVM CFI 는 보호를 위한 코드가 소스코드에 포함된 것에 비해 CFG 에서는 외부 모듈의 함수 호출을 위한 태스크가 추가된다. 두번째 한계점은 추가적인 메모리 접근 비용이다. CFG 는 제어 흐름 보호 코드를 실행하면서 메모리 공간에 존재하는 CFG bitmap 을 접근한다. LLVM CFI 는 CFG bitmap 역할을 하는 각 함수의 고유한 메모리 주소의 베이스 주소(base address)를 소스코드에 포함한다. 이 때문에 CFG 는 LLVM CFI 에 비해 추가적인 메모리 접근이 발생한다. 위 두가지 한계점에

의해 CFG 의 실행속도가 LLVM CFI 에 비해 상대적으로 느리다. 표 1 과 표 2 는 CFI 가 적용된 함수와 적용되지 않은 함수의 호출 시간을 구하는 프로그램을 Windows 와 LLVM CFI 의 환경에서 비교한 결과이다. 이를 확인하면 CFG 를 적용했을 때의 시간 증가율이 LLVM CFI 를 적용했을 때의 증가율보다 큰 것을 확인할 수 있다.

3.2 확장성(Scalability)의 한계

여기서 말하는 확장성은 비용 대비 개발난이도의 크고 낮음을 뜻한다. CFG 는 제어 흐름 보호를 위해 시스템 함수를 호출한다는 점이 LLVM CFI 에 비해 확장성을 떨어뜨린다. LLVM CFI 의 경우 제어 흐름 보호를 위한 로직이 실행 프로그램 내부에 포함되어 컴파일러가 제어 흐름 보호 코드를 모두 생성한다. 이로 인해 NVFCC 와 같이 확장적인 보호기능이나 새로운 버전을 개발할 시 컴파일러의 수정만을 통해 구현할 수 있다. 하지만 CFG 의 경우 컴파일러의 수정은 물론 시스템 모듈을 수정해야 하기 때문에 개발비용이 크다. 또한 매 함수단위로 검사하기 때문에, 성능에 치명적인 영향을 끼치는 CFI 의 특성상 확장적인 보호기능을 개발할 시 생기는 모듈과 실행 프로그램의 통신이 성능에 영향을 끼쳐 개발난이도를 크게 높인다.

3.3 보안(Security)의 한계

CFG 는 LLVM CFI 와 다르게 CFG Bitmap 을 통해 제어 흐름의 변조를 확인하기 때문에 CFG Bitmap 을 변조하는 것으로 CFG 를 우회할 위험이 있다. 공격자가 CFG Bitmap 의 베이스 주소(base address)를 획득할 수 있고 메모리 오염 등의 취약점이 존재할 시 CFG 우회가 가능하게 된다.

4. Control Flow Guard 의 한계에 대한 대응책 제안

CFG 의 한계점은 보안의 문제를 제외하고는 외부 모듈의 사용과 추가적인 메모리접근에 인해 발생한다. 두가지 모두 실행 프로그램 내부에 구현한다면 문제가 해결된다.

4.1 CFI Check 의 인라인(inline)화

첫번째 방법은 CFG 의 제어 흐름 변조 확인 로직을 LLVM CFI 와 같이 컴파일러 지원을 통해 실행 프로그램 내부에 인라인(inline)으로 삽입하는 방법이다. 이를 통해 외부 모듈실행을 제거함으로써 성능을 향상시킨다. 또한 OS 의존성을 제거하고 보호기능 추가에 대한 확장성을 증대한다. 3.2 에서 언급한 것과 같이 NVFCC 와 같은 기능을 추가할 시 외부 모듈을 사용하지 않는다면 설계난이도가 줄어들고 성능향상을 도모할 수 있다.

4.2 CFG Bitmap 의 인라인(inline)화

두번째 방법은 CFG Bitmap 에 저장하는 주소를 LLVM CFI 와 같이 컴파일러에서 파악하여 실행 프로그램 내부에 인라인으로 삽입하는 방법이다. CFG Bitmap 은 런타임 시간에 생성하기 때문에 이 시간을 컴파일 시간으로 옮겨 런타임 시간이 빨라진다[4]. 또한 메모리 접근을 줄임으로써 CFI Check 로직의 성능이 증진된다. 그리고 CFG Bitmap 을 운용하지 않음으로써 CFG Bitmap 을 탈취할 가능성을 제거해 보안을 향상시킨다

5. 결론

본 논문에서는 Windows CFG 의 한계점을 LLVM IFCC 와 비교하여 탐색하였고 이에 대한 대응책을 제안하였다. 1) 메모리 접근과 외부 모듈 호출로 인한 성능 감소, 2) 시스템 모듈 의존성에 인한 기능확장의 어려움을 3) CFG Bitmap 에 의한 보안 한계점에 대해 알아보았다. 이에 대한 대응책으로 CFI Check 와 CFG Bitmap 의 인라인화를 제안하였다. 본 논문은 Windows CFG 의 LLVM IFCC 에 대한 상대적인 약점을 보였다. 본 연구 결과는 가장 높은 시장 점유율을 가진 Windows 의 CFI 기술을 발전하는 데에 도움이 될 것으로 판단된다.

Acknowledgement

본 논문은 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임 (No. NRF-2020R1A2C1012708).

참고문헌

[1] A. Biondo, M. Conti and D. Lain, "Back to the epilogue: evading control flow guard via unaligned targets," Proc. of the Network and Distributed Systems Security Symposium, Feb. 2018.

[2] Microsoft documentation, Control Flow Guard for platform security Available at “https://docs.microsoft.com/en-us/windows/win32/secbp/control-flow-guard”

[3] Clang documentation, Control Flow Integrity Design Documentation “https://releases.llvm.org/10.0.0/tools/clang/docs/ControlFlowIntegrityDesign.html”

[4] Jack Tang and Trend Micro Threat Solution Team. 2015. Exploring control flow guard in windows 10. Available at “http://blog.trendmicro.com/ trendlabs-security-intelligence/ exploring-control-flow-guard-in-windows-10”