Journal of The Korea Institute of Information Security & Cryptology
VOL.30, NO.5, Oct. 2020
ISSN 1598-3986(Print)
ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2020.30.5.881

안드로이드 애플리케이션 환경에서 CFI 우회 공격기법 연구

이 주 엽,[1†] 최 형 기[2‡]
[1,2]성균관대학교(대학원생, 교수)

A Study of Attacks to Bypass CFI on Android Application Environment

Ju-yeop Lee,[1†] Hyoung-kee Choi[2‡]
[1,2]Sungkyunkwan University(Graduate student, Professor)

요 약

CFI(Control Flow Integrity)는 제어 흐름을 검증해 프로그램을 보호하는 기법이다. 안드로이드 환경에서 애플리케이션 보호를 위해 LLVM Clang 컴파일러가 지원하는 CFI 기법인 IFCC(Indirect Function Call Checks)와 SCS(Shadow Call Stack)이 도입되었다. IFCC가 함수 호출, SCS이 함수 복귀 시 제어 흐름을 보호한다. 본 논문에서는 IFCC, SCS을 적용한 애플리케이션 환경에서 CFI 우회 공격기법을 제안한다. 사용자 애플리케이션에 IFCC, SCS을 적용하여도 애플리케이션 메모리 내 IFCC, SCS으로 보호되지 않은 코드 영역이 다수 존재하는 것을 확인하였다. 해당 코드 영역에서 공격을 위한 코드를 실행해 1) IFCC로 보호된 함수 우회 호출 기법, 2) SCS 우회를 통한 복귀 주소 변조 기법을 구성한다. 안드로이드10 QP1A. 191005.007.A3 환경에서 IFCC, SCS으로 보호되지 않은 코드 영역을 식별하고 개념 증명(proof-of-concept) 공격을 구현해 IFCC, SCS이 적용된 환경에서 제어 흐름 변조가 가능함을 보인다.

ABSTRACT

CFI(Control Flow Integrity) is a mitigation mechanism that protects programs by verifying control flows. IFCC(Indirect Function Call Checks) and SCS(Shadow Call Stack), CFI supported by LLVM Clang compiler, were introduced to protect applications in Android. IFCC protects function calls and SCS protects function returns. In this paper, we propose attacks to bypass CFI on the application environment with IFCC and SCS. Even if IFCC and SCS were applied to user applications, it was confirmed that there were many code segments not protected by IFCC and SCS in the application memory. We execute code in CFI unprotected segments to construct 1) bypassing IFCC to call a protected function, 2) modulating return address via SCS bypass. We identify code segments not protected by IFCC and SCS in Android10 QP1A. 191005.007.A3. We also implement proof-of-concept exploits to demonstrate that modulation of control flow is possible in an environment where IFCC and SCS are applied.

Keywords: Android, CFI(Control Flow Integrity), IFCC(Indirect Function Call Checks), SCS(Shadow Call Stack), Bypass

I. 서 론

제어 흐름 변조 기법은 프로그램을 공격하기 위해 사용되는 주요 기법 중 하나이다. 공격자는 메모리 오염(memory corruption) 취약점으로 함수 포인터나 복귀 주소(return address)를 덮어써 프로그램의 제어 흐름을 변조해 악의적인 기능을 수행한다 〔1〕, 〔2〕. 공격으로부터 프로그램을 보호하기 위해 CFI(Control Flow Integrity)〔2〕, 〔3〕,

Received(08. 18. 2020), Accepted(08. 31. 2020)
† 주저자, daf198@g.skku.edu
‡ 교신저자, meosery@g.skku.edu(Corresponding author)

ASLR(Address Space Layout Randomization)〔4〕, RELRO(RELocation Read-Only)〔5〕 등 보호기법이 개발되었다. 이 중 CFI는 컴파일 시 제어 흐름 그래프(control flow graph)를 작성하고 프로그램 실행 시 제어 흐름을 비교해 검증하는 기법이다. 제어 흐름이 변경되는 주요 분기는 함수 호출과 복귀(return)이다〔6〕. 함수 호출을 보호하는 순방향(forward-edge) CFI와 함수 복귀를 보호하는 역방향(backward-edge) CFI가 검증된 제어 흐름만 허용해 프로그램을 보호한다〔6〕. 최근 주요 운영체제의 개발도구에 CFI가 통합, 지원되어 보호하는 환경의 범위를 넓혀가고 있다〔6〕, 〔7〕. 특히 시장 점유율이 85%에 달하는 주요 모바일 운영체제인 안드로이드 환경에서 제어 흐름을 보호하기 위해 LLVM Clang 컴파일러가 지원하는 CFI가 도입되었다〔8〕, 〔9〕, 〔10〕. 안드로이드8부터 순방향 CFI인 IFCC(Indirect Function Call Checks), 안드로이드10부터 역방향 CFI인 SCS(Shadow Call Stack)을 지원한다. 즉 안드로이드10부터 개발자는 애플리케이션에 IFCC, SCS을 적용해 순방향, 역방향 제어 흐름을 보호할 수 있다.

본 논문에서는 IFCC, SCS을 모두 적용한 안드로이드 애플리케이션 환경에서 CFI를 우회할 수 있는 취약점을 보인다. CFI를 우회하기 위해 IFCC, SCS으로 보호되지 않은 코드 영역을 사용하며 해당 영역을 CFI 비보호 영역으로 명명한다. 사용자 애플리케이션에 IFCC, SCS을 적용하여도 애플리케이션 메모리에 CFI 비보호 영역이 다수 존재하는 것을 확인하였다. CFI 비보호 영역에서 공격을 위한 코드를 실행하여 IFCC, SCS을 우회, 제어 흐름을 변조한다. 안드로이드10 QP1A. 191005.007.A3 환경에서 CFI 비보호 영역을 안드로이드 시스템 라이브러리, 시스템 코드에서 식별하고 개념 증명(proof-of-concept) 공격을 구성해 순방향, 역방향 CFI가 적용된 환경에서 제어 흐름 변조가 가능함을 증명한다. 또한 CFI 우회 공격기법의 실효성을 논의하고 CFI 적용 환경의 문제점을 보완하기 위한 아이디어를 제시한다.

II. 배 경

CFI의 기본적인 동작 구조를 알아보고 안드로이드에 적용된 LLVM Clang 컴파일러가 지원하는 CFI 기법인 IFCC와 SCS을 분석한다. 안드로이드 환경에서 CFI를 통해 보호받을 수 있는 코드의 범위에 대해 설명한다.

2.1 CFI 개요

CFI는 제어 흐름 변조 공격을 막기 위한 보호기법의 일종이다. 제어 흐름이 변경되는 주요 분기는 함수 호출과 이전 함수로 복귀이다. CFI는 함수 호출 분기를 보호하는 순방향 CFI와 함수 복귀 분기를 보호하는 역방향 CFI로 구분한다. 순방향 CFI는 호출하는 목적지 주소가 런타임에 결정되는 간접 호출을 보호한다. 이 기법은 코드 컴파일 시 코드 내 간접 호출되는 모든 목적지 주소를 수집해 제어 흐름 그래프로 작성한다. 실행 시 각 간접 호출에서 호출하는 목적지 주소를 제어 흐름 그래프와 비교해 사전에 정의된 주소인지 검증한다. Fig.1.에서 caller 함수에서 callee 함수를 호출하는 제어 흐름이 순방향이다. 순방향 CFI를 적용하면 공격자가 callee 함수의 포인터를 attack 함수 주소로 변조하여도 호출 시 목적지 주소를 검증하여 사전에 정의되지 않은 attack 함수 호출을 허용하지 않는다.

스택에 저장되는 복귀 주소를 조작해 제어 흐름을 변조하는 공격들은 역방향 CFI가 방어한다. 이 기법은 복귀 주소 관리를 위한 특별 영역을 메모리에 할당하고 함수 호출 시 복귀 주소를 스택에 저장하는 동시에 특별 영역에도 저장한다. 함수 실행 후 복귀 시에 스택 내 주소와 특별 영역 내 주소를 비교해서 복귀 주소를 검증한다. Fig.1.에서 callee 함수 실행 후 caller 코드로 복귀가 역방향 제어 흐름의 예시이다. 공격자가 스택에 저장된 caller 함수로의 복

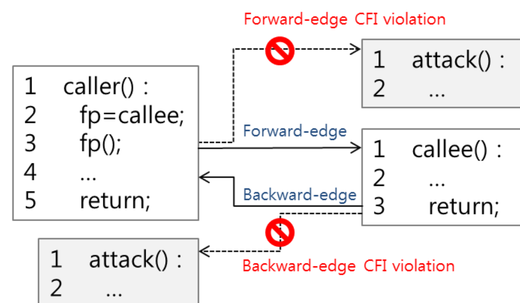


Fig. 1. Examples of forward and backward-edge. CFI provides verification of the destination address in each branch.

귀 주소를 임의로 변조하여도 특별 영역에 저장된 주소와 검증하므로 제어 흐름 변조를 방어할 수 있다.

2.2 LLVM Clang CFI 분석

안드로이드에 도입된 CFI 기법은 순방향 CFI 기법인 IFCC(Cross-DSO 모드[11])와 역방향 CFI 기법인 SCS이다. 해당 기법들의 보호 디자인을 분석한다.

2.2.1 Clang IFCC

IFCC는 코드 내 간접 호출을 통한 제어 흐름 변조를 보호한다. IFCC를 적용해 컴파일 시에 프로그램 내 간접 호출되는 함수들의 정보를 수집해 점프 테이블을 생성한다. 점프 테이블의 각 항목(entry)은 8바이트 단위로 정렬되며 간접 호출 목적지 주소로 점프하는 명령으로 작성된다. 프로그램 내 간접 호출되는 주소는 점프 테이블 내의 해당 목적지로 점프하는 항목의 주소로 변경된다. 간접 호출 시 항목의 주소를 검증해 주소가 점프 테이블 영역 내부이면 정상 호출되고 외부이면 CFI 위반으로 판단한다. 공격자가 간접 호출의 목적지 주소를 변경하여도 점프 테이블에 정의된 함수만 호출할 수 있어 제어 흐름의 변조를 방지한다.

동적 라이브러리를 사용하는 프로그램의 제어 흐름을 보호하기 위해 다른 방식의 보호 절차가 필요하다. 동적 라이브러리는 프로그램 실행 중 메모리에 로드된다. 동적 라이브러리를 참조하는 프로그램은 컴파일 시 라이브러리의 함수 정보를 알 수 없어 IFCC를 적용할 수 없다. IFCC Cross-DSO 모드는 동적 라이브러리를 사용하는 프로그램에서 라이브러리 함수의 제어 흐름을 보호하기 위해 개발된 IFCC 기법의 일종이다[11].

동적 라이브러리의 제어 흐름을 보호하기 위해 라이브러리를 참조하는 코드 영역에 IFCC Cross-DSO 모드를 적용해야 한다. 함수를 호출하는 코드 영역에서 동적 라이브러리 내 함수 정보를 확인할 수 없기 때문에 함수를 검증하기 위해 검증 요청 함수(__cfi_slowpath)를 호출한다. 검증 요청 함수는 간접 호출 목적지 함수가 포함된 동적 라이브러리의 자체 검증 함수(__cfi_check)의 주소를 계산한다. 간접 호출 목적지 함수를 구분하기 위한 식별자와 목적지 주소를 매개변수로 전달해 검증 함수를 호출한다.

동적 라이브러리 컴파일 시 IFCC Cross-DSO 모드를 적용하면 검증 함수가 생성된다. 검증 함수는 해당 동적 라이브러리 내 함수들의 주소와 식별자 정보를 가진다. 검증 함수가 호출되면 전달받은 식별자와 목적지 주소를 비교해 검증하고 결과를 반환한다. 목적지 주소와 식별자가 모두 같을 경우 CFI 검증을 통과하고 이외 경우 CFI 위반으로 처리한다. IFCC Cross-DSO 모드가 적용되지 않은 동적 라이브러리는 검증 함수가 존재하지 않아 검증 정보를 제공할 수 없다. IFCC Cross-DSO 모드가 적용된 간접 호출 코드에서 IFCC Cross-DSO 모드가 적용되지 않은 동적 라이브러리의 함수를 호출하면 검증 요청 함수에서 검증 함수를 호출하지 않고 복귀해 검증이 생략된다. 즉 정상적으로 IFCC Cross-DSO 모드로 제어 흐름을 보호하기 위해서는 간접 호출 코드 영역과 간접 호출 목적지 코드 영역 모두에 IFCC Cross-DSO 모드가 적용되어야 한다.

Fig.2.는 동적 라이브러리 libshared.so 내 dummy 함수를 간접 호출 시 IFCC Cross-DSO 모드 검증 절차의 예시이다. 동적 라이브러리 libshared.so 내 dummy 함수를 간접 호출 시 검증 요청 함수를 실행한다(Fig.2. ①). 검증 요청 함

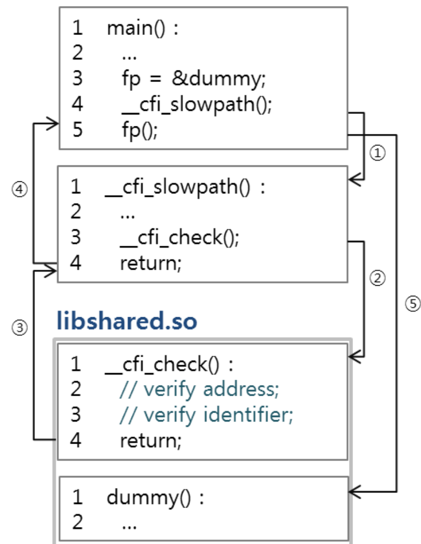


Fig. 2. When IFCC Cross-DSO mode is applied, the destination address and identifier are compared to verify.

수에서 libshared.so의 검증 함수의 주소를 계산해 호출하고 목적지 주소와 식별자를 전달한다(Fig.2. ②). libshared.so 내 검증 함수에서 두 값을 비교해 검증한 결과를 반환하고(Fig.2. ③,④) 검증을 정상 통과하면 간접 호출이 실행된다(Fig.2. ⑤). 간접 호출 전 검증을 진행함으로써 동적 라이브러리의 함수 호출 시 제어 흐름을 보호한다.

2.2.2 Clang SCS

SCS은 Clang 7 버전부터 적용되어 현재 ARM64 아키텍처 환경만 지원한다. SCS은 복귀 주소를 일반 스택과 별도로 지정한 쉐도우 스택(shadow stack)에 각각 저장한다. 쉐도우 스택은 공격으로부터 복귀 주소를 보호하기 위한 메모리 내 8192바이트 크기의 특별 영역이다. SCS이 활성화된 함수들은 쉐도우 스택에 저장한 주소로 복귀해 역방향 제어 흐름을 보호한다. 쉐도우 스택의 주소는 특수 목적으로 사용하는 X18 레지스터로 관리한다〔12〕. X18 레지스터는 호출 규약 상 스택에 저장되지 않아 스택이 유출되어도 쉐도우 스택 주소가 노출되지 않는다. 또한 SCS 관련 함수 이외에 X18 레지스터를 조작하는 코드가 존재하지 않아 코드 재사용 공격(code reuse attack)〔13〕으로 X18 레지스터를 변조하기 어렵다.

Fig.3.는 SCS이 적용된 ARM64용 어셈블리 코드 예시이다. ARM64 환경에서 복귀 주소는 X30 레지스터에 저장된다. Fig.3.의 3행과 같이 복귀 주소를 X18 레지스터가 가리키는 쉐도우 스택에 저장한다. 호환성을 위해서 Fig.3.의 4행과 같이 복귀 주소를 스택에 저장하고, 함수 실행 후 스택에 저장된 복귀 주소를 X30 레지스터에 복구한다(Fig.3.의 6행). 최종으로 쉐도우 스택에 있는 복귀 주소를 X30 레지스터로 복구하고 함수를 종료한다(Fig.3.

```
1  Function() :
2      SUB  SP, SP, #0x20
3      STR  X30, [X18], #8
4      STP  X29, X30, [SP,#0x10+var_s0]
5      …
6      LDP  X29, X30, [SP,#0x10+var_s0]
7      LDR  X30, [X18, #-8]!
8      RET
```

Fig. 3. When SCS is applied, codes are inserted to store and recover the return address in the shadow stack.

의 7행). 스택 버퍼 오버플로(stack buffer overflow)와 같은 취약점이 발생해 스택의 복귀 주소를 임의의 값으로 덮어써도 쉐도우 스택에 보관된 주소로 복귀해서 역방향 제어 흐름은 보호된다. SCS을 적용하지 않고 코드를 컴파일하면 Fig.3.의 3, 7행은 생략된다.

2.3 안드로이드 환경 상 CFI 보호 범위

제어 흐름 변조 공격은 주로 네이티브 코드(C, C++)에서 발생하는 메모리 오염 취약점을 이용한다〔1〕. CFI는 제어 흐름 변조를 막기 위해 네이티브 코드로 작성된 코드 영역을 보호한다. 애플리케이션 실행 시 메모리에는 다수의 코드 영역이 로드된다. 기본적으로 애플리케이션 코드와 안드로이드에서 제공하는 시스템 라이브러리, 애플리케이션 개발자가 작성한 사용자 라이브러리가 로드된다. 추가로 애플리케이션에 공통으로 필요한 자원들을 사전에 로드해놓는 Zygote〔14〕 코드를 포함해 안드로이드의 기능을 지원하는 시스템 코드 영역들이 메모리에 적재되어 복합적으로 사용된다. 이 중 CFI의 주요 보호 대상은 네이티브 코드로 작성되며 사용자 입력을 처리하여 취약점이 발생할 수 있는 시스템 라이브러리와 사용자 라이브러리이다. 개발자가 IFCC, SCS을 활성화해 애플리케이션을 제작하면 사용자 라이브러리에 순방향, 역방향 CFI가 적용되어 함수 호출, 복귀 시 제어 흐름을 보호한다.

III. CFI 우회 공격기법 구성

IFCC(Cross-DSO 모드)와 SCS을 적용한 애플리케이션 환경에서 CFI를 우회해 제어 흐름을 변조할 수 있는 취약점을 탐색한다. 순방향, 역방향 제어 흐름이 보호받는 환경에서 보안 취약점을 탐색하였으며 취약점을 기반으로 IFCC(Cross-DSO 모드), SCS을 우회하는 공격기법을 제안한다. 편의를 위해 IFCC Cross-DSO모드는 이하 IFCC로 표기한다.

3.1 CFI 우회 취약점 탐색 조건

IFCC와 SCS을 모두 적용해 제작한 사용자 애플리케이션 환경에서 CFI 우회 취약점을 탐색한다. 애플리케이션에 포함된 코드 중 개발자가 네이티브 코드로 기능을 구현해 사용하는 사용자 라이브러리가

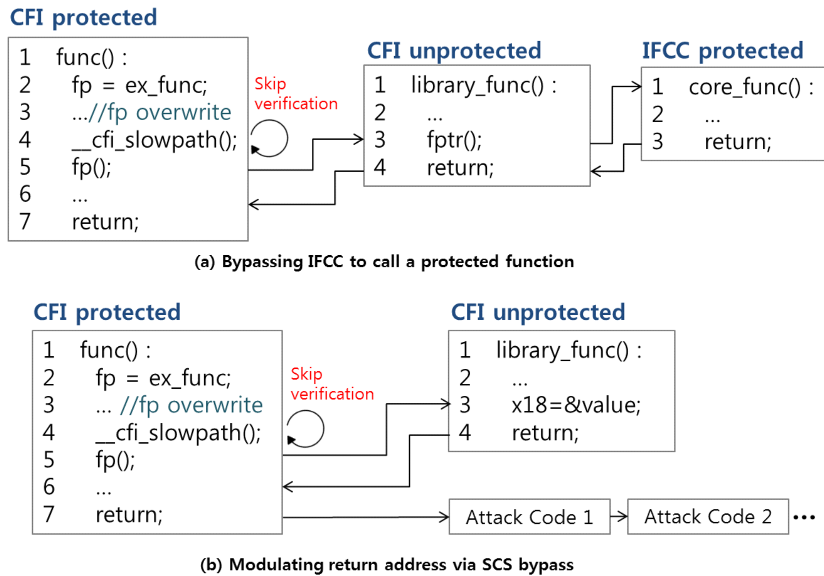


Fig. 4. We propose attacks to bypass IFCC, SCS using code gadgets in CFI unprotected segments.

IFCC, SCS으로 보호된다. CFI 우회 취약점 탐색 시 IFCC, SCS으로 보호된 코드에 메모리 오염 취약점이 있어 함수 포인터를 변조해 간접 호출의 목적지 주소를 조작할 수 있다고 가정한다. 메모리 오염 취약점은 버퍼 오버플로, 포맷 스트링 버그(format string bug) 등 다양한 버그로 발생할 수 있어 일반적인 가정이다[15], [16]. 코드가 IFCC, SCS으로 정상 보호되면 간접 호출을 조작해도 제어 흐름을 임의로 변조할 수 없다. 이 같은 환경에서 CFI를 우회해 제어 흐름을 변조할 수 있는 취약점을 탐색한다.

3.2 CFI 비보호 영역을 이용한 CFI 우회 공격기법 제안

안드로이드 애플리케이션 메모리에는 다양한 코드 영역이 로드되어 복합적으로 사용된다. 제어 흐름 보호를 위해 IFCC, SCS을 적용하여도 애플리케이션 메모리 내 IFCC, SCS으로 보호되지 않은 CFI 비보호 영역이 존재하는 것을 확인하였다. 메모리 내 CFI 비보호 영역이 있으면 CFI 우회 공격에 활용할 수 있다. CFI 비보호 영역의 특징은 다음과 같다. 1) IFCC로 보호된 코드에서 IFCC로 보호되지 않은 코드를 목적지로 간접 호출 시 검증 절차를 생략하고 목적지 주소를 호출한다. 2) IFCC로 보호되지 않은 코드는 간접 호출의 검증이 없어 목적지의 IFCC 보호 여부와 관계 없이 간접 호출할 수 있다. 3) SCS으로 보호되지 않은 코드는 복귀 주소를 스택에서 읽어 사용하므로 스택을 조작해 역방향에서 제어 흐름을 변조할 수 있다. 이 같은 특징들을 가진 CFI 비보호 영역에서 특정 기능을 수행하고 복귀하는 일련의 코드 조각인 코드 가젯들을 실행해 CFI 우회 공격기법을 구성한다.

CFI 우회 공격기법의 상세 절차는 Fig.4.와 같다. Fig.4. (a)는 IFCC로 보호된 함수 우회 호출 기법이다. IFCC로 보호된 코드에서 간접 호출 시 목적지 주소와 식별자를 검증한다. 공격자가 간접 호출을 변조해 IFCC로 보호된 임의의 함수를 호출하면 식별자가 일치하지 않아 검증에 실패한다. 이 때 CFI 비보호 영역에서 간접 호출하는 코드 가젯을 거쳐 IFCC로 보호된 임의의 함수를 호출할 수 있다. 검증을 생략해 호출 가능한 CFI 비보호 영역에는 간접 호출의 검증 절차가 없어 간접 호출의 목적지를 조작해 IFCC로 보호된 함수를 호출할 수 있다. 이 우회 기법으로 공격자는 IFCC로 보호되지 않은 함수뿐 아니라 IFCC로 보호된 함수까지 제약 없이 호출할 수 있다.

Fig.4. (b)는 SCS 우회를 통한 복귀 주소 변조 기법이다. SCS 적용 시 X18 레지스터를 이용해 쉐도우 스택 주소를 관리한다. X18 레지스터의 값을

Table 1. A number of indirect call gadgets and X18 modulation gadgets were identified in CFI unprotected segments. The system libraries display only four, considering the quantity of gadgets.

	System codes		System libraries			
	boot.oat	app_process64	libhwui.so	libandroid_runtime.so	libfrsdk.so	libc.so
Indirect call gadget	48859	92	18814	10902	9644	835
X18 modulation gadget	-	-	-	2	-	1

조작하면 쉐도우 스택 대신 다른 메모리 영역에서 복귀 주소를 읽도록 할 수 있다. CFI 비보호 영역에서 X18 레지스터의 값을 변조하는 코드 가젯을 탐색해 간접 호출한다. CFI 비보호 영역 내에 있는 코드 가젯은 간접 호출 시 검증 절차가 생략되어 정상 호출된다. X18 레지스터의 값을 공격자가 조작 가능한 메모리 주소로 설정하고 해당 메모리에 실행할 주소를 저장한다. 이후 SCS으로 보호된 코드에서 X18 레지스터를 참조해 복귀하면 공격자가 저장한 주소가 실행된다. 결과적으로 SCS을 우회해 복귀 주소를 변조할 수 있다. 제안한 두 가지 CFI 우회 공격기법을 통해 IFCC, SCS이 모두 적용된 코드에서 순방향, 역방향 제어 흐름을 변조할 수 있다.

3.3 안드로이드10 환경 상 우회 공격 가능성 평가

3.2에서 제안하는 CFI 우회 공격기법을 수행하기 위해 애플리케이션 메모리 내 CFI 비보호 영역이 있고 해당 영역에 공격을 위한 코드 가젯이 존재해야 한다. 우회 공격 가능성 확인을 위해 ARM64 기반 Pixel XL 기기의 최신 버전 안드로이드10 QP1A. 191005.007.A3 환경에서 CFI 비보호 영역 탐색을 진행하였다. 대상 장비 파일 시스템에서 파일 추출 후 역어셈블리 도구(objdump, ROPgadget[17])로 CFI 적용 시 생성되는 함수 및 명령을 탐색해 CFI 비보호 영역을 식별하였다. 분석 결과 시스템 라이브러리와 애플리케이션에 공통적으로 로드되는 시스템 코드에서 CFI 비보호 영역을 다수 확인하였다. 시스템 라이브러리에 IFCC는 안드로이드8부터, SCS은 안드로이드10부터 적용되어 일부 라이브러리를 보호한다. 탐색한 시스템 라이브러리 570개 중 IFCC로 보호된 라이브러리 118개, SCS으로 보호된 라이브러리 3개, IFCC, SCS으로 모두 보호된 라이브러리 2개를 확인하였다. 즉 570개 중 123개를 제외한 447개 시스템 라이브러리가 애플리케이션 메모리에 로드되면 CFI 비보호 영역으로 활용할 수 있다. 또 실행 권한을 가진 시스템 코드 중 boot.oat 영역과 app_process64 영역, 두 개 코드 영역이 CFI 비보호 영역임을 확인하였다.

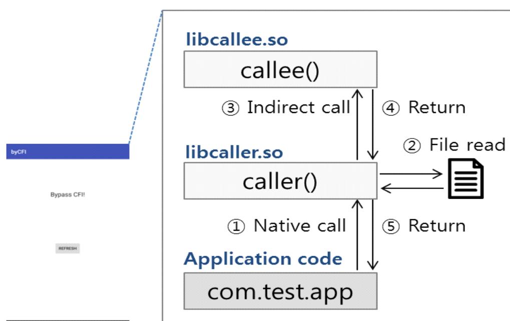


Fig. 5. A test application consists of an application codes written in java and two user libraries written in C.

CFI 우회 공격기법에 필수로 필요한 코드 가젯은 X18 변조 가젯과 간접 호출 가젯이다. X18 변조 가젯은 MOV, LDR 등 명령으로 X18 레지스터의 값을 변경할 수 있는 코드 가젯으로 선정한다. 간접 호출 가젯은 레지스터를 이용해 목적지 주소를 호출하는 BLR, BR 명령을 사용하는 코드 가젯으로 선정한다. 역어셈블리 도구를 이용해 코드 가젯의 수량을 식별하는 스크립트를 작성하여 실행, 분석한 결과 CFI 비보호 영역에서 Table 1.과 같은 수량을 확인하였다. 간접 호출 가젯은 한 개 영역에 최대 48859개, 평균 640개로 코드 영역 전반에 다수 분포해 공격 환경에 알맞은 코드 가젯을 선택해 사용할 수 있다. X18 변조 가젯은 총 3개로 수량은 적지만 메모리에 기본으로 적재되는 시스템 라이브러리(libandroid_runtime.so, libc.so) 내에 있어 범용적으로 활용할 수 있다. 최신 안드로이드10 환경에서 CFI 우회 공격기법을 구성할 수 있는 CFI 비보호 영역과 공격 코드 가젯의 수량을 확인하였다.

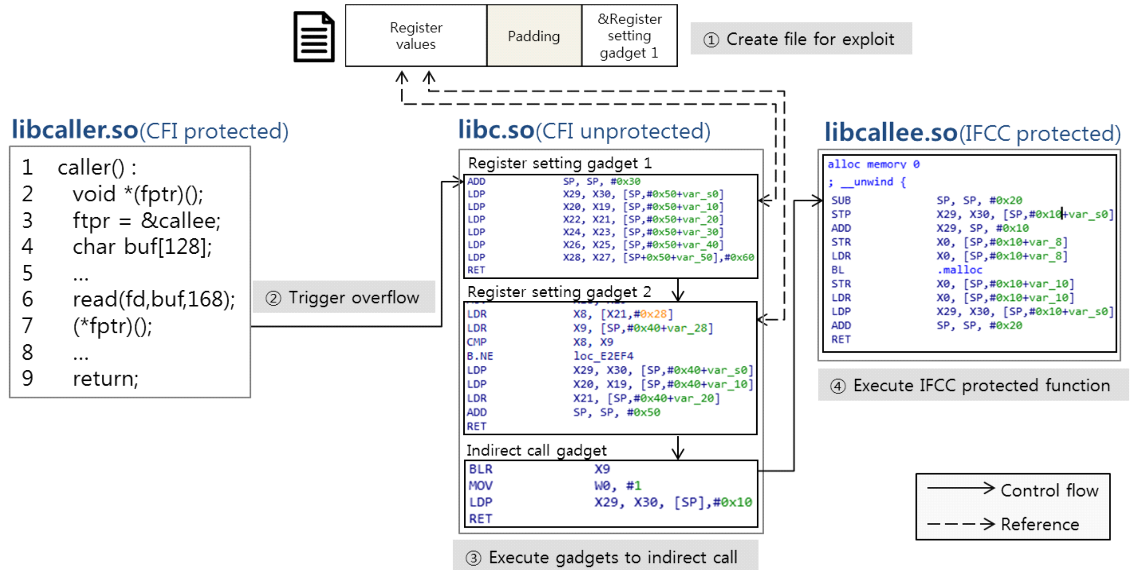


Fig. 6. Indirect call gadgets in CFI unprotected segment allow calls to IFCC protected functions by bypassing verification.

IV. 개념 증명 공격 구현

IFCC, SCS을 적용한 테스트 애플리케이션을 제작하고 이를 대상으로 개념 증명 공격을 진행해 실제 환경에서 CFI 우회 공격이 가능함을 보인다.

4.1 환경 상세

개념 증명 공격은 3.3에서 기술한 환경에서 진행한다. 테스트 애플리케이션의 상세 구조는 Fig.5.와 같다. 테스트 애플리케이션 코드는 자바로 작성하며 C로 작성한 두 개의 사용자 라이브러리(libcaller.so, libcallee.so)를 포함한다. 사용자 라이브러리에 IFCC, SCS을 모두 적용해 라이브러리 함수의 순방향, 역방향에서 제어 흐름을 보호한다. 사용자 라이브러리는 C 또는 C++로 작성되어야 하고 IFCC, SCS을 활성화해 컴파일해야 하므로 전체 소스코드가 필요하다. 또한 테스트를 위해 코드 내 간접 호출을 변조할 수 있는 취약점이 있어야 한다. 기존에 출시된 애플리케이션들로 위 조건들을 충족하기 어려워 자체 애플리케이션을 제작해 테스트하였다. 애플리케이션 실행 시 화면에 버튼이 위치하며 버튼을 누르면 libcaller.so 라이브러리에 정의된 함수 caller가 호출된다(Fig.5. ①). caller는 특정 파일을 읽어 화면에 출력 후(Fig.5. ②) libcallee.so 내 함수 callee를 간접 호출한다(Fig.5. ③). callee는 일련의 기능을 수행하고 복귀해 애플리케이션 코드로 제어 흐름이 되돌아간다(Fig.5. ④,⑤).

4.1.1 보유 취약점

테스트 애플리케이션은 스택 버퍼 오버플로 취약점을 가지고 있다. libcaller.so 내 함수 caller에서 libcallee.so의 함수 callee를 간접 호출하기 전 특정 파일을 읽기 위해 read 함수를 호출한다. 이때 파일 내용을 저장하는 버퍼의 크기보다 파일로부터 큰 크기를 읽어 스택 버퍼 오버플로가 발생한다. caller의 스택에는 파일 내용을 저장하는 버퍼와 callee의 주소가 저장된 함수 포인터가 연속적으로 위치해 오버플로가 발생하면 함수 포인터가 변조된다. 즉 스택 버퍼 오버플로를 통해 간접 호출의 목적지 주소를 조작할 수 있다.

4.1.2 적용 보호기법

라이브러리에 적용되는 보호기법은 스택 및 힙에 실행 권한을 없애는 DEP(Data Execution

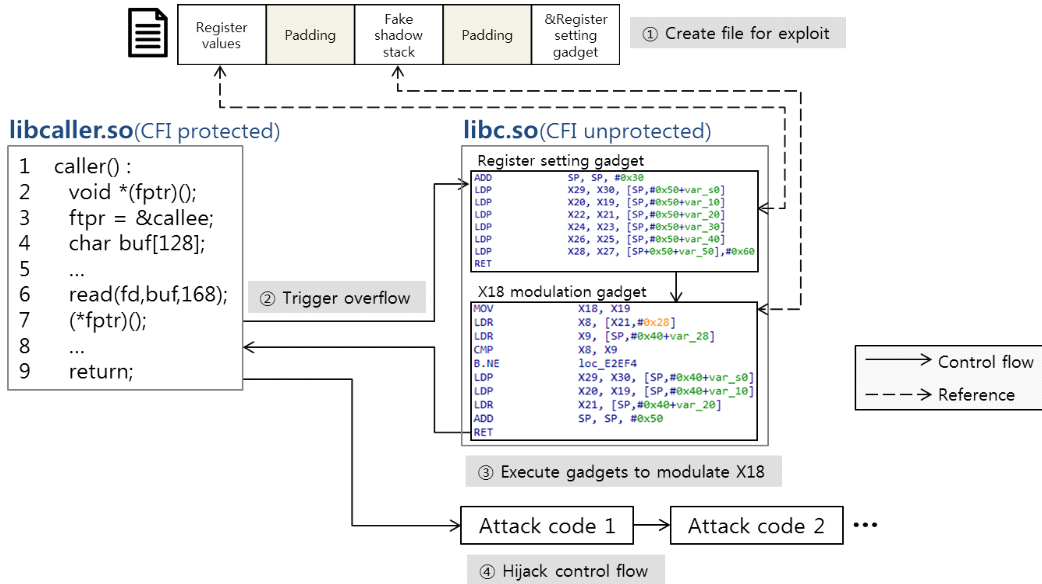


Fig. 7. The return address can be modulated by bypassing SCS using X18 modulation gadget.

Prevention)[13], GOT(Global Offset Table)에 쓰기 권한을 부여하지 않는 Full RELRO[5], IFCC(Cross-DSO 모드), SCS이다. 메모리 영역의 주소를 랜덤하게 변경하는 ASLR은 메모리 유출(memory leak) 취약점으로 우회할 수 있다[18]. 본 개념 증명 공격에서는 간접 호출의 목적지를 조작해 메모리를 유출하는 일련의 코드 가젯을 실행, ASLR을 우회할 수 있어 테스트 애플리케이션에 적용하지 않았다. 안드로이드 환경에서 ASLR 우회에 대한 자세한 내용은 5.1.1에서 기술한다.

4.2 공격 절차

4.2.1 IFCC로 보호된 함수 우회 호출 기법

IFCC로 보호된 함수 우회 호출 기법의 전체 제어 흐름은 Fig.6.와 같다. 간접 호출로 메모리를 유출해 메모리 주소를 파악하는 절차는 생략한다. 먼저 테스트 애플리케이션이 읽는 파일을 대체해 공격에 사용할 파일을 생성한다(Fig.6. ①). 스택에 저장되는 파일 내용에 공격에 필요한 값들을 입력해 코드 가젯에서 활용한다. 본 공격에서는 코드 가젯에서 레지스터에 설정할 값들과 함수 포인터를 덮어써 호출할 코드 가젯의 주소를 버퍼의 크기를 고려해 파일로 작성한다. 애플리케이션 화면에서 버튼을 누르면 caller 함수에서 *read(fd,buf,168)*가 호출되어 40바이트 스택 버퍼 오버플로가 발생한다(Fig.6. ②). 파일의 내용이 버퍼 *buf*와 연속적으로 위치한 함수 포인터 *fptr*까지 덮어써 함수 포인터의 값이 코드 가젯의 주소 *&Register Setting Gadget 1*으로 변조된다. 이후 간접 호출 시 CFI 비보호 영역에서 간접 호출을 하기 위한 가젯들이 실행된다(Fig.6. ③). CFI 비보호 영역의 코드 가젯은 복귀 주소를 스택에서 읽어 사용하므로 스택에 호출할 주소를 배치해 코드 가젯을 연속으로 실행할 수 있다. 본 공격에서 사용한 간접 호출 가젯은 X9 레지스터가 가진 주소를 호출한다. 간접 호출 가젯을 실행하기 전 레지스터를 설정하는 가젯들을 먼저 호출해 *LDR X9, [SP,#0x40+var_28]*에서 스택에 저장된 값으로 X9 레지스터를 설정한다. 이 때 X9 레지스터의 값은 공격자가 스택에 저장한 IFCC로 보호된 함수 주소 *&alloc_memory_0*으로 설정된다. 최종적으로 간접 호출 가젯에서 *BLR X9*가 실행되어 검증 절차 없이 IFCC로 보호된 함수가 정상 호출된다(Fig.6. ④). 공격자는 간접 호출 가젯을 이용해 목적지 주소의 IFCC 보호 여부와 관계 없이 호출할 수 있다.

4.2.2 SCS 우회를 통한 복귀 주소 변조 기법

SCS 우회를 통한 복귀 주소 변조 기법의 전체적

인 제어 흐름은 Fig.7.과 같다. 본 공격에서는 공격용 파일의 내용을 코드 가젯에서 레지스터에 설정할 값들과 실행할 복귀 주소들, 함수 포인터를 덮어써 호출할 코드 가젯의 주소로 구성한다. 특히 실행할 복귀 주소들이 저장되는 스택 영역을 쉐도우 스택을 대신하는 가짜 쉐도우 스택으로 활용한다. 4.2.1. 기법과 동일하게 스택 버퍼 오버플로를 발생시켜 함수 포인터를 코드 가젯의 주소 *&Register Setting Gadget*으로 설정한다. 이후 간접 호출로 X18 변조를 위한 코드 가젯들이 실행된다(Fig.7. ③). X18 변조 가젯을 실행하기 전 코드 가젯의 형태에 따라 필요한 레지스터 값들을 사전에 설정해야 한다. 본 공격에서 사용한 X18 변조 가젯을 사용하기 위해서 X19, X21 레지스터를 설정해야 한다. 이를 위해 레지스터를 설정하는 코드 가젯을 먼저 호출해 *LDP X20, X19, [SP,#0x50+var_10], LDP X22, X21, [SP,#0x50+var_20]*에서 X19, X21 레지스터를 공격자가 조작한 값으로 설정한다. 이 때 X19 레지스터 값은 가짜 쉐도우 스택의 주소 *&Fake shadow stack*으로 설정된다. X18 변조 가젯에서 *MOV X18, X19*가 실행되면 최종적으로 X18 레지스터는 가짜 쉐도우 스택의 주소로 설정된다. SCS으로 보호된 코드에서 X18 레지스터를 참조해 복귀하면 가짜 쉐도우 스택에 저장된 주소로 복귀한다(Fig.7. ④). 공격자는 가짜 쉐도우 스택에 실행할 주소를 순차적으로 배치해 복귀 주소로 사용할 수 있다.

개념 증명 공격으로 보인 두 가지 기법으로 IFCC, SCS으로 보호된 코드 영역에서 순방향, 역방향 제어 흐름 변조가 가능한 것을 보였다. 두 가지 기법에서 사용한 코드 가젯은 모두 시스템 라이브러리 libc.so에서 탐색하였다. libc.so는 대다수 애플리케이션에 기본으로 로드되어 타 애플리케이션을 대상으로 공격을 진행하여도 코드 가젯을 충분히 활용할 수 있다.

V. 논 의

본 논문에서 제안한 CFI 우회 공격기법의 실효성을 분석한다. 또한 CFI가 적용된 안드로이드 환경의 보안 취약점에 대응해 제어 흐름 보호를 강화하기 위한 아이디어를 제시한다.

5.1 CFI 우회 공격기법 실효성 분석

5.1.1 안드로이드 ASLR 우회 가능성

공격자는 CFI를 우회해 제어 흐름을 조작하기 위해 ASLR을 우회해야 한다. 안드로이드 환경에서 ASLR은 제한적으로 적용되어 메모리 유출 취약점이 있으면 ASLR을 우회할 수 있다[19]. 안드로이드에서는 애플리케이션에 공통으로 필요한 자원을 메모리에 로드 해놓은 Zygote 프로세스를 사전에 생성하고 추후 애플리케이션 실행 시 이 프로세스를 포크(fork)해 사용한다. Zygote 프로세스 생성 시 ASLR이 적용되어 메모리 주소가 변동되지만 이후 실행되는 프로세스들은 모두 Zygote 프로세스와 동일한 메모리 주소를 갖는다. 즉 애플리케이션이 재시작되어도 메모리 주소가 변동되지 않는다. 애플리케이션에 메모리 유출 취약점이 있으면 공격자는 반복적으로 메모리 유출을 시도해 메모리 주소를 파악, ASLR을 우회할 수 있다.

5.1.2 CFI 비보호 영역 유지 가능성

CFI 우회 공격기법을 수행하기 위해서 애플리케이션 메모리 내 CFI 비보호 영역이 존재해야 한다. 현재 시스템 라이브러리 중 일부에 IFCC, SCS이 적용되어 있다. 시스템 라이브러리에 CFI를 적용한 기간을 고려하면 전체 라이브러리에 IFCC, SCS이 패치되는 것은 상당 시간이 소요될 것으로 예상된다. 시스템 라이브러리 전체에 CFI가 적용되어도 시스템 코드 중 CFI 비보호 영역을 활용할 수 있다. 시스템 코드에 IFCC, SCS이 적용될 수 있지만 적용 시 성능 저하가 발생한다[11]. 모든 애플리케이션에서 공통으로 사용되는 시스템 코드는 전체 성능에 영향을 줄 수 있어 즉각적인 IFCC, SCS 적용은 어려울 것으로 판단된다. 추가로 CFI로 보호되지 않은 사용자 라이브러리를 사용할 수 있다. IFCC, SCS을 적용하기 위해서는 라이브러리를 다시 컴파일해야 해, 소스코드를 보유하지 못한 경우 CFI를 적용할 수 없다. 또한 재사용이 많은 라이브러리 특성 상 CFI로 보호되지 않은 사용자 라이브러리가 사용될 확률이 높다. 결론적으로 단기간 내에 모든 코드 영역에 CFI가 패치되기 어려워 CFI 우회 공격기법을 활용할 수 있을 것으로 보인다.

5.1.3 CFI 우회 공격기법 활용 방안

취약점 및 적용 보호기법을 고려한 공격 환경에 따라 제어 흐름을 탈취, 공격을 진행하는데 CFI 우회 공격기법을 활용할 수 있다. 예를 들어, 공격자가 간접 호출을 조작할 수 있고 특정 레지스터를 변조할 수 있는 경우 IFCC로 보호된 함수 우회 호출 기법으로 복귀 주소 변조 없이 순방향 제어 흐름만 이용해 IFCC로 보호된 함수를 호출할 수 있다. 공격을 진행하며 필수로 호출해야 하는 함수에 SCS이 적용된 경우 SCS 우회를 통한 복귀 주소 변조 기법으로 함수를 실행하고 복귀 시 공격자의 제어 흐름을 이어갈 수 있다. 또 SCS으로 보호된 코드에서 복귀 주소 변조가 가능해 SCS으로 보호된 코드를 코드 가젯으로 활용 가능하다. 이외에도 공격 시나리오에 따라 두 가지 기법을 조합하여 CFI를 우회해 제어 흐름 변조 공격을 진행하는데 활용할 수 있다.

5.2 보안 취약점 대응방안

5.2.1 IFCC, SCS 중 한가지 기법 우선 적용

본 논문에서 제시하는 우회 공격기법은 IFCC, SCS이 적용되어 있지 않은 코드 영역의 코드 가젯을 활용한다. 코드 영역에 SCS이 적용되면 코드 가젯 실행 후 복귀 시 쉐도우 스택에 저장된 복귀 주소로 복귀한다. 간접 호출로 코드 가젯을 실행하여도 복귀 주소를 변조하지 못해 코드 가젯을 연속적으로 실행할 수 없다. 코드 영역에 IFCC가 적용되면 간접 호출 시 검증 절차가 추가되어 코드 가젯을 호출할 수 없다. 즉 코드 영역에 IFCC 혹은 SCS 한가지 기법만 우선 적용되어도 본 논문에서 제시하는 CFI 우회 공격기법을 포함한 제어 흐름 변조 공격들의 성공률을 낮출 수 있다.

5.2.2 함수 주소 정렬을 이용한 코드 가젯 제한

윈도우 CFG(Control Flow Guard)[6]의 보호 디자인을 Clang CFI 기법에 적용할 수 있다. CFG를 적용해 컴파일하면 함수의 시작 주소를 0x10 정렬이 맞도록 생성한다. 프로그램 실행 중 호출하는 주소의 0x10 정렬 여부를 검증해 올바른 함수 주소인지 확인한다. Clang IFCC에 유사한 보호 디자인을 적용해 공격 코드 가젯의 수량을 줄일 수 있다. 코드 가젯은 주로 함수 중반부터 복귀까지 코드를 사용한다. 코드에서 간접 호출 시 정렬된 함수 시작 주소만 호출하도록 강제하면 함수 중반에 존재하는 코드 가젯은 호출할 수 없다. 예를 들어 함수 시작 주소를 0x100 단위로 정렬하고 간접 호출 시 목적지 주소의 하위 두 자리를 0으로 설정해 호출할 수 있다. 이 방식을 적용하면 공격자는 함수들의 시작 주소 또는 0x100 단위로 정렬된 코드 가젯만 호출할 수 있어 호출 가능한 코드 가젯의 수량을 감소시킬 수 있다. 이 기법으로 코드 가젯을 제한해 추가적인 공격 진행을 억제할 수 있지만 컴파일러의 수정이 필요하다.

VI. 관련 연구

CFI는 제어 흐름 보호를 위해 런타임에 제어 흐름을 검증해야 해 성능 저하가 발생한다. 이로 인해 CFI는 성능을 고려해 완화된 검증 조건을 가진 coarse-grained CFI와 성능 저하를 감수하고 사전에 허용된 주소만 호출하는 fine-grained CFI로 분류된다[6]. 초기 coarse-grained CFI가 다수 발표되었으나 최근 보안의 중요성이 부각됨에 따라 fine-grained CFI가 주를 이루고 있다[3], [19]. 공격자들은 CFI를 우회하기 위해 보호기법 디자인 취약점을 탐색하고 동작 절차 상 발생 가능한 문제점을 분석한다. CFI 우회를 위해 본 논문에서 활용한 CFI 비보호 영역을 이용한 공격 이외에 다양한 방식의 공격기법들이 제안되었다.

먼저 완화된 검증 조건을 만족하는 코드 가젯을 CFI 우회에 이용할 수 있다. Carlini의 연구팀은 최근 호출 기록으로 제어 흐름을 검증하는 CFI를 대상으로 연구를 진행하였다[20]. CFI 우회를 위해 호출 기록을 은닉할 수 있는 코드 가젯의 형태를 식별해 공격을 구성하였다. Davi의 연구팀은 기존에 발표된 다섯 개의 coarse-grained CFI 기법을 합쳐 가장 제한된 조건을 가진 CFI 기법을 구현하고 이를 우회할 수 있음을 보였다[21]. 제한된 조건을 우회하기 위해 필요한 가젯을 여섯 종으로 분류하고 코드 재사용 공격을 수행해 CFI를 우회하였다. 이 기법들은 완화된 제어 흐름 검증 조건을 가진 CFI 기법들을 우회할 수 있지만 검증 시 사전에 호출 가능한 목적지 주소만 허용하는 fine-grained CFI 방식은 우회할 수 없다는 한계가 존재한다.

CFI 동작을 위해 작성한 제어 흐름 그래프의 범

위 내에서 공격을 수행하는 기법들도 제안되었다. Carlini의 연구팀은 제어 흐름에 직접 관여하지 않는 데이터(non-control-data)를 이용해 허용된 제어 흐름 내에서 공격자의 코드를 실행하는 기법을 보였다[13]. Evans의 연구팀은 제어 흐름 그래프 생성 시 불완전한 포인터 분석을 이용해 CFI를 우회하였다[22]. 포인터 분석의 유연성으로 인해 잘못된 간접 호출의 목적지 주소를 허용하고 이를 활용해 우회 공격한다. 이 공격들로 최신 fine-grained CFI 기법들이 우회될 수 있음을 보였다.

또한 CFI가 적용된 환경을 이용해 우회 공격을 구성할 수 있다. Conti의 연구팀은 CFI가 적용된 아키텍처의 호출 규약을 이용한 공격을 제시하였다[19]. 호출 규약 상 스택에 저장되는 레지스터 값을 변조해 CFI 검증 절차를 우회하였다. 스택에 저장된 레지스터 값을 유출하여 쉐도우 스택의 주소를 파악해 역방향 CFI를 우회하는 기법도 제안하였다. Biondo의 연구팀은 윈도우 환경에서 CFI 보호 디자인이 지켜지지 않은 라이브러리를 확인하고 해당 영역에서 공격을 위한 코드 가젯을 실행해 성공적으로 CFI를 우회하였다[6]. 이 기법들은 특정 환경의 고유 특성에 의존해 CFI가 적용되는 운영체제 및 아키텍처가 변경되면 활용할 수 없다는 제약이 있다. 제시한 세 가지 방식 이외에도 다양한 취약점을 기반으로 CFI를 우회하는 기법들이 제안되었다.

본 논문에서 제시한 공격기법은 CFI가 적용되는 환경을 이용한 우회 공격들과 유사하다. 안드로이드 환경에 특정되는 공격기법이지만 기존에 연구되지 않은 ARM64 아키텍처 안드로이드 기반에 IFCC Cross-DSO 모드, SCS을 모두 적용한 환경에서 CFI 우회 공격기법을 연구한 점에서 의의가 있다.

VII. 결 론

본 논문에서는 Clang IFCC(Cross-DSO 모드)와 SCS으로 보호된 안드로이드 애플리케이션 환경에서 취약점을 탐색하고 CFI 우회 공격기법을 제안하였다. IFCC, SCS을 적용한 애플리케이션 메모리에서 CFI 비보호 영역을 탐색해 공격에 활용하였다. CFI 비보호 영역에서 코드 가젯을 실행해 1) IFCC로 보호된 함수 우회 호출 기법, 2) SCS 우회를 통한 복귀 주소 변조 기법을 구성하였다. 1) IFCC로 보호된 함수 우회 호출 기법은 검증 절차가 없는 CFI 비보호 영역의 간접 호출을 이용해 IFCC로 보호 받는 함수를 제약 없이 호출한다. 2) SCS 우회를 통한 복귀 주소 변조 기법은 쉐도우 스택의 주소를 관리하는 레지스터를 조작해 복귀 주소를 변조한다. 두 가지 기법으로 순방향, 역방향 CFI로 보호되는 코드에서 CFI를 우회해 제어 흐름을 변조한다. 안드로이드10 환경 상 CFI 비보호 영역을 탐색, 코드 가젯의 수량을 확인하고 개념 증명 공격을 보였다. 또한 CFI 우회 공격기법의 실효성을 분석하고 CFI 적용 환경의 보안 취약점에 대응해 제어 흐름 보호를 강화할 수 있는 방안을 아이디어 수준에서 제시하였다. 안드로이드 환경에서 IFCC, SCS은 새롭게 도입되어 점차 보호 범위를 확대해 나가고 있다. 본 논문의 우회 공격기법을 통해 시스템의 일부 코드 영역에 CFI를 적용하는 것은 보호 성능이 저하되거나 보호절차가 우회될 수 있다는 것을 보였다. 본 연구 결과는 향후 안드로이드 환경에 순방향, 역방향 CFI가 확대 적용, 발전하는 데에 도움이 될 것으로 판단된다.

References

[1] L. Szekeres, M. Payer and T. Wei, "Sok: eternal war in memory," Proc. of the IEEE Symposium on Security and Privacy, pp. 48-62, May. 2013.

[2] N. Burow, S.A. Carr and J. Nash, "Control-flow integrity: precision, security, and performance," ACM Computing Surveys, pp. 1-33, Apr. 2017.

[3] M. Abadi, M. Budiu and J. Ligatti, "Control-flow integrity principles, implementations, and applications," ACM Transactions on Information and System Security, pp. 1-40, Nov. 2009.

[4] J. Seo, B. Lee and S.M. Shih, "SGX-shield: enabling address space layout randomization for SGX programs," Proc. of the Network and Distributed Systems Security Symposium, Feb. 2017.

[5] S. Jeong, J. Hwang and H. Kwon, "A CFI countermeasure against GOT overwrite attacks," IEEE Access, pp.

36267-36280, Feb. 2020.

[6] 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.

[7] C. Tice, T. Roeder and P. Collingbourne, "Enforcing forward-edge control-flow integrity in {GCC} & {LLVM}," Proc. of 23rd USENIX Security Symposium, pp. 941-955, Aug. 2014.

[8] T. Nguyen, J. Mcdonald and W. Glisson, "Detecting repackaged android applications using perceptual hashing," Proc. of the 53rd Hawaii International Conference on System Sciences, pp. 6641-6650, Jan. 2020.

[9] E.M. Koruyeh, S.H.A. Shirazi and K.N. Khasawneh, "SPECCFI: mitigating spectre attacks using CFI informed speculation," Proc. of the IEEE Symposium on Security and Privacy, pp. 39-53, May. 2020.

[10] AOSP, "ShadowCallStack," https://source.android.com/security/enhancements/enhancements10, Dec. 2019.

[11] Clang 12 documentation, "Cross-DSO," https://clang.llvm.org/docs/ControlFlowIntegrityDesign.html, Dec. 2019.

[12] Clang 12 documentation, "ShadowCall Stack," https://clang.llvm.org/docs/ShadowCallStack.html, Dec. 2019.

[13] N. Carlini, A. Barresi and M. Payer, "Control-flow bending: on the effectiveness of control-flow integrity," Proc. of 24th USENIX Security Symposium, pp. 161-176, Aug. 2015.

[14] V. Parikh and P. Mateti, "ASLR and ROP attack mitigations for ARM-based android devices," International Symposium on Security in Computing and Communication, pp. 350-363, Nov. 2017.

[15] T. Saito, R. Watanabe and S. Kondo, "A survey of prevention/mitigation against memory corruption attacks," 19th International Conference on Network-Based Information Systems, pp. 500-505, Sep. 2016.

[16] J. Xu, P. Ning and C. Kil, "Automatic diagnosis and response to memory corruption vulnerabilities," Proc. of the 12th ACM conference on Computer and communications security, pp. 223-234, Nov. 2005.

[17] ROPgadget, "gadget," http://shell-storm.org/project/ROPgadget, Dec. 2019.

[18] B. Lee, L. Lu and T. Wang, "From zygote to morula: fortifying weakened aslr on android," Proc. of the IEEE Symposium on Security and Privacy, pp. 424-439, May. 2014.

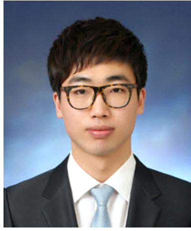
[19] M. Conti, S. Crane and L. Davi, "Losing control: on the effectiveness of control-flow integrity under stack attacks," Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 952-963, Oct. 2015.

[20] N. Carlini and D. Wagner, "{ROP} is still dangerous: breaking modern defenses," Proc. of 23rd USENIX Security Symposium, pp. 385-399, Aug. 2014.

[21] L. Davi, A.R. Sadeghi and D. Lehman, "Stitching the gadgets: on the ineffectiveness of coarse-grained control-flow integrity protection," Proc. of 23rd USENIX Security Symposium, pp. 401-416, Aug. 2014.

[22] I. Evans, F. Long and U. Otgonbaatar, "Control jujutsu: on the weaknesses of fine-grained control flow integrity," Proc. of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 901-913, Oct. 2015.

〈저자소개〉



이 주 엽 (Ju-yeop Lee) 정회원
2014년 2월: 성균관대학교 컴퓨터공학과 졸업
2019년 3월~현재: 성균관대학교 소프트웨어학과 석사과정
〈관심분야〉 정보보호, 시스템 보안

최 형 기 (Hyoung-kee Choi) 정회원
1992년 2월: 성균관대학교 전자공학과 졸업
1996년 2월: Polytechnic University in Brooklyn, NY 석사
2001년 2월: Georgia Institute of Technology in Atlanta, GA 박사
2001년 1월~2004년 12월: Lancope 근무
2004년 3월~현재: 성균관대학교 소프트웨어대학 교수
〈관심분야〉 네트워크 보안, 리버스 엔지니어링