情報保護學會論文誌
第 22 卷 第 5 號, 2012. 10

소스코드 기반의 정밀도 높은 실행 흐름 보호 기법*

이 종 협,† 김 용 승‡
한국교통대학교

Precise control flow protection based on source code*

JongHyup Lee,† Yong Seung Kim‡
Korea National University of Transportation

요 약

기존의 Control Flow Integrity(CFI)와 Control Flow Locking(CFL) 기법은 프로그램이 개발자의 의도대로만 실행되도록 강제하여 실행 흐름의 무결성을 제공하고 안전한 프로그램 실행을 보장한다. 하지만, 함수 호출 문맥을 인지하는 보호 기법을 제공하지 않아 정밀도가 떨어지고 이를 악용한 공격을 허용하는 취약점을 가지고 있다. 본 논문에서는 이러한 문제점들을 해결하는 Source-code CFI(SCFI) 시스템을 제안한다. 제안한 시스템에서는 정밀도 높은 CFI 기능을 제공하여 프로그램의 안전성을 높인다.

ABSTRACT

Control Flow Integrity(CFI) and Control Flow Locking(CFL) prevent unintended execution of software and provide integrity in control flow. Attackers, however, can still hijack program controls since CFI and CFL does not support fine-granularity, context-sensitive protection. In this paper, we propose a new CFI scheme, Source-code CFI(SCFI), to overcome the problems. SCFI provides context-sensitive locking for control flow. Thus, the proposed approach protects software against the attacks on the previous CFI and CFL schemes and improves safety.

Keywords: Software Security, Control Flow Integrity

I. 서 론

소프트웨어 보안에서의 안전성이란 소프트웨어가 개발자의 의도대로만 실행되는 것을 의미한다. 하지만 실제 소프트웨어는 추상화 단계의 소스코드에서 시작되어 컴파일이라는 과정을 거쳐 기계가 인식할 수 있는 바이너리(binary) 형태의 실행파일이 되고, 이 실행파일은 사용자의 수만큼이나 다양한 컴퓨팅 환경에서 실행된다. 따라서 안전한 소프트웨어의 실행이란 이러한 복잡한 단계와 상황에 있는 실제 실행파일이 개발당시의 의도를 정확하게 반영하게 해야 한다는 어려움을 가지게 된다.

소프트웨어 보안 공격은 프로그램 개발상의 실수에 집중한다. 가장 대표적인 취약점은 메모리의 변수에 할당되어 있는 공간을 넘어서 비정상적인 메모리 쓰기 수행하는 버퍼 오버플로우(buffer overflow) 공격이나 printf등에서 사용하는 포맷스트링(format string)에 대한 직접적인 접근을 통한 공격들이 있다. 이러한 공격의 목표는 크게 두 가지로 구별할 수 있다. 첫 번째는 저수준 프로그램의 stack frame에서 볼 수 있는 복귀 주소(return address)나 저장된 frame pointer와 같은 프로그램의 제어(control)와 관련된 제어 데이터의 변조이고 두 번째는 프

접수일(2012년 9월 10일), 수정일(2012년 10월 16일), 게재확정일(2012년 10월 16일)
* 이 논문은 2012년도 한국교통대학교 교내학술연구비의 지원을 받아 수행한 연구임
† 주저자, jhlee@ut.ac.kr
‡ 교신저자, ys@ut.ac.kr

로그램에서 사용하는 데이터에 대한 변조이다. 이 중에서 제어 데이터에 대한 변조는 공격자가 수행중인 소프트웨어의 제어 권한을 가로채(control hijacking) 프로그램 중 원하는 코드나 자신이 제공하는 코드를 수행할 수 있게 한다는 점에서 더 위험성이 크다.

소프트웨어 공격에 대한 방어 방법은 다양한 측면에서 연구되어 왔다. 공격의 예방과 탐지 측면에서 strcpy와 같이 버퍼 오버플로우 공격에 대해 안전하지 않은 함수를 더 안전한 형태의 함수로 바꾸거나 stack canary와 같이 버퍼 오버플로우가 발생하는 것을 미리 감지하여 공격이 발생하더라도 프로그램의 제어를 빼앗기지 않도록 하는 기법들이 제안되어 사용되어왔지만, 여전히 이를 우회하는 공격들도 함께 발달하고 있다. 이와 다른 측면으로 실효성 있는 공격을 막기 위하여 공격자가 자신이 침투시킨 코드를 실행시키지 못하도록 하기 위해 주소 공간을 랜덤화 하는 Address space layout randomization(ASLR) 기법[1]과 같은 방어법도 이용되고 있지만, 소프트웨어 코드 자체를 재사용하여 공격하는 코드 재사용 공격 등에 취약점을 보이고 있다. 이러한 문제점들을 해결하고 프로그램 제어권 유지 강화를 통해 근본적인 소프트웨어 안전성 유지를 위한 방법으로 연구된 대표적인 기술이 Software Fault Isolation(SFI,[2,3])와 CFI[4,5]이다. 특히 CFI의 경우는 정적인 분석(static analysis)을 통해 합당한 프로그램 제어권 이동을 미리 분석하고, 프로그램 수행 과정 동안 제어권 이동이 침해되지 않는다는 무결성을 유지해 준다는 점에서 소프트웨어의 안전성을 보장하기 위한 중요한 조건을 만족시키는 효과적인 접근방법이다.

대표적인 CFI에 대한 연구[4]는 저수준의 바이너리 코드를 대상으로 프로그램에서 의도된 제어 이동을 미리 분석하여 표시해 두었다가 프로그램 실행시에 프로그램의 제어가 변하는 시점마다 표시를 확인하여 의도된 제어의 이동이 맞는지를 확인하는 방식이다. 하지만 CFI[4]가 이용하는 바이너리 코드에 대한 정적 분석은 저수준 코드 분석에서 발생하는 어려움 때문에, 효율성과 적용가능성을 고려한 프로그램 제어 잠금(CFL) 기법[5]이 제안되었다. CFL 기법은 소스코드의 단계에서 적용될 수 있어 대상 플랫폼에 상관없이(platform independent) 적용될 수 있다는 측면에서 장점을 가지지만, 기법의 단순화 과정을 통하여 세밀한(fine granularity) 제어를 제공하지 않아 이를 이용한 공격이 가능하다는 한계점을 가지고 있다.

따라서 본 논문에서는 기존의 소스코드 기반의 CFL 기법이 가지는 문제점을 극복하기 위해 함수 호출의 문맥을 인지(context-sensitive)할 수 있는 프로그램 제어 잠금 기법인 SCFI를 제안한다. SCFI는 소스코드를 대상으로 함수 호출 그래프(call graph)를 분석하고, 분석된 호출 관계를 바탕으로 가능한 프로그램 제어의 변화를 미리 계산한다. 계산된 프로그램 제어에 따라 의도된 프로그램 제어 변화를 확인 할 수 있는 'lock 값'을 설정한다. 이 때 같은 함수에 대한 호출이더라도 호출되는 위치마다 별개의 lock 값을 설정하고 호출된 함수는 이 정보를 유지함으로써 함수 호출마다 다른 lock 값으로 프로그램 제어의 안전성을 보장하게 한다. [그림 1]은 기존 기법과 제안하는 기법의 동작과정의 차이를 간략히 보여준다. 특히 제안하는 SCFI는 CIL[6]을 기반으로 자동화된 시스템으로 개발되어 입력받은 소스코드에서 SCFI이 적용된 실행파일을 자동으로 얻을 수 있다.

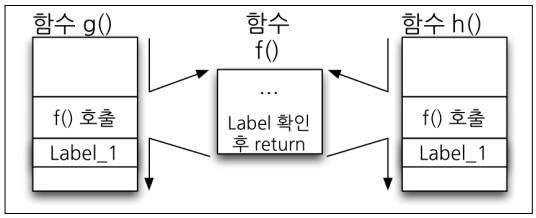
본 논문은 다음과 같이 구성된다. 2절에서는 기존의 CFI와 CFL 기법에 대해서 설명하고 기존 기법들이 가지는 한계점을 이용한 공격 방법을 보여준다. 3절에서는 제안하는 SCFI 기법에 대해 자세히 설명한다. 4절에서는 구현 과정과 실험 결과를 통해서 SCFI의 효과를 분석하고 5절에서는 SCFI 시스템 구성과정에서 보안과 관련하여 고려할 사항에 대하여 토의하고 향후 연구 방향을 제시한 후 6절에서 끝맺는다.

II. 기존의 프로그램 제어권한 보장 기법들

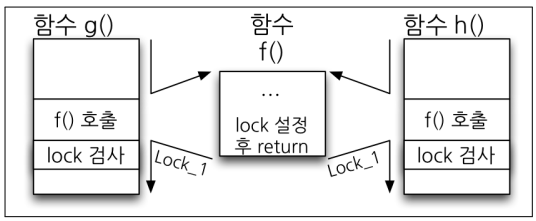
2.1 프로그램 제어 무결성 보장 기법 (control flow integrity)

변조되지 않은 프로그램 흐름 제어는 프로그램의 안전한 동작을 위한 필수적인 조건이다. 프로그램의 제어는 일반적으로는 순차적으로 흘러가지만, 함수를 호출하거나 호출된 함수에서 복귀(retrun)하는 경우에 특정 위치로의 프로그램 제어의 이동(transfer)이 생기게 된다. 이러한 프로그램 제어 이동의 순간에서 프로그램 제어가 보호 받지 못한다면 소프트웨어의 안전성에 직접적인 영향을 받게 되는데, 버퍼 오버플로우와 같은 공격 또한 이러한 프로그램의 제어권 이동의 순간을 목표로 프로그램의 제어권을 공격자가 가져오기 위해 일어나는 공격이라 할 수 있다.

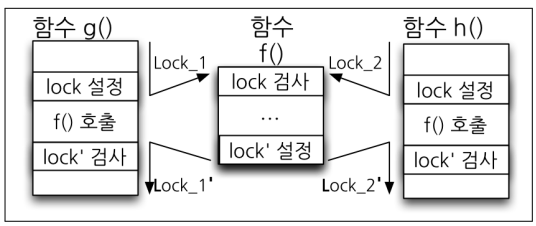
CFI는 프로그램 제어권 이동시의 안전과 무결성을 보장하기 위하여 제안되었다[4]. CFI는 프로그램이



(a) CFI



(b) CFL



(c) SCFI

〔그림 1〕 기존의 CFI와 CFL의 동작과정과 제안하는 SCFI 기법의 동작과정

수행되는 동안 프로그램의 제어권이 이동될 때마다 이미 알려진 유효한 위치로만 이동될 수 있도록 강제하고 이에 대한 무결성을 보장하는 것을 목표로 한다. 이러한 목적을 달성하기 위하여, CFI는 프로그램의 수행 전에는 정적인 분석을 통해 프로그램을 분석하고 이를 바탕으로 프로그램을 수정하여 실행되는 동안 분석된 대로 프로그램이 의도된 제어권 이동만을 수행하도록 검사한다. 동작 과정은 〔그림 1〕(a)와 같으며 자세한 내용은 다음과 같다.

1. 저수준 프로그램에 대한 정적인 분석 과정을 통하여 프로그램의 제어 흐름을 나타내는 CFG(Control Flow Graph)를 구성하여 함수 복귀와 같이 프로그램 제어 이동이 발생하는 곳을 파악한다.
2. 파악된 프로그램 제어 이동에 대하여 label를 할당한다. label를 할당할 때에는 중복될 수 있는 프로그램 제어 이동들도 같은 값을 가지도록 한다. 예를 들어, 〔그림 1〕(a)와 같이 어떠한 함수 f()가 g()와 h() 모두에게서 불리게 된다면 함수 f()의 return은 자신을 부른 함수에 따라 함수 g()내에 있는 호출 위치(call site) 바로 다음 명령어(instruction)나 h() 함수 내에 있는 호출 위치 다음 명령어로 갈 수 있기 때문에 f() → g()와 f() → h()의 두 제어 이동에 대하여 같은 label를 부여한다.
3. 바이너리 프로그램에 대한 수정 방법(binary instrumentation)을 이용하여 제어 이동 이전과 이후에 해당하는 프로그램 코드를 수정한다. 1) 제어 이동이 다다르게 될 위치의 코드에 할당된 label 값을 설정한다. 2) 제어 이동이 일어나기 이전에는 자신이 제어권을 이동하려는 곳에 자신에게 맞는 label가 있는 지를 확인하는 코드를 추가한다. 그림의 예에서, 함수 f()가 return 이후 g()와 h()로 돌아갈 위치(함수 호출 다음 명령어)에 label값을 기록하고, 함수 f()는 return을 통해 제어 이동을 하기 이전에 자신이 이동할 곳의 코드에 정해진 label의 값이 있는 지를 확인하고 정당한 경우에만 return 명령을 수행하게 된다.

CFI는 문제가 될만한 프로그램 제어 이동의 경우를 미리 찾아서 방어한다. 하지만, CFI에는 몇 가지 문제점이 있다. 첫째, CFI는 제어 이동 제한의 정밀도(granularity)가 떨어진다. CFI는 함수 문맥을 고려하지 않아(context insensitive) 어느 함수가 호출했는지에 대한 정보가 없어 자신을 호출하는 call site라면 어디로든 이동할 수 있게 된다. 즉, 위의 예처럼 g()와 h()가 f()를 호출하는 경우, 의도된 제어 이동은 g() → f() → g()이거나 h() → f() → h() 뿐이지만, g() → f() → h()와 같은 의도되지 않은 이동을 허용할 수 있다. 특히 함수 포인터와 같은 간접 호출(indirect call)의 경우 어떠한 함수도 호출될 수 있어 제어 이동을 강제하는 label 마저도 특정하지 못하기 때문에, CFI의 효과가 무력화 되는 문제가 발생하게 된다. 둘째, CFI는 효과를 높이기 위해 바이너리 코드 상태의 소프트웨어를 대상으로 정적 분석을 수행하고 있지만, 바이너리 코드의 정적 분석을 통한 완벽한(complete) CFG를 얻기가 쉽지 않다. 바이너리 코드를 대상으로 한 정적 분석 과정에서 필수적인 abstract interpretation 기법을 향상시키기 위한 연구들이 진행되어 왔지만〔7〕, 여전히 완벽한 결과를 얻기 위해 많은 노력이 필요한 상황이다. 완벽하지 못한 CFG는 그만큼 CFI의 빈틈을 만들어 공격

당할 가능성이 높아진다. 또한 바이너리 코드를 대상으로 한 기법은 calling convention과 같은 플랫폼과 밀접한 문제들이 함께 다루어져야 하기 때문에 다른 플랫폼으로의 적용이 쉽지 않다.

2.2 프로그램 제어 잠금 기법 (control flow locking)

CFI의 성능 저하 개선을 위해 소스코드에서도 적용이 가능한 프로그램 잠금 기법인 CFL이 제안되었다[5]. CFL은 소스코드를 대상으로도 적용이 가능하기 때문에 플랫폼과 상관없이 사용 가능하다. CFL은 프로그램이 실행되기 이전에 프로그램을 분석하고 제어의 이동을 제한한다는 측면에서 CFI와 유사하지만, 소스코드를 대상으로 한 프로그램이 분석이 가능하기 때문에 제어 이동에 대한 분석 결과가 더 정확하고 프로그램 자체에 label를 넣는 것이 아니라 별도의 lock 변수를 정의하고 제어 이동을 특정 위치로만 제한하는 lock 값을 lock 변수에 저장하여 확인한다는 점에서 차이를 가진다([그림 1](b) 참조). 특히 CFL은 저수준 바이너리 코드에서만 발생할 수 있는 프로그램의 구조 자체를 벗어나는 비정상적인 제어 이동을 방지하기 위하여 SFI 기법[2]과 동시에 적용되는 것을 가정하고 있다. 따라서 소스코드 수준에서 드러나는 구조적인 제어 이동의 문제점을 방지함으로써 소프트웨어의 보안성을 높이게 된다.

하지만 CFL에서도 CFI가 가지는 동일한 정밀성의 문제를 가지게 된다. 간접 함수에 대해서는 함수 호출이나 return의 제어 이동을 제한할 수 없으며, 직접 호출의 경우에는 자신이 호출되는 어느 위치로라도 이동할 수 있어 CFL을 우회하는 공격이 가능하다. [그림 2]의 예는 이러한 한계점 때문에 발생할 수 있는 문제를 보여주고 있다.

2.3 기존 기법이 가지는 문제점의 예

[그림 2]의 예제 시나리오에서 `main()` 함수는 입력을 받아 `vuln_func()` 함수를 호출하여 이를 확인하고 `authenticate()` 함수를 통해 인증 후, 다시 `vuln_func()` 함수를 이용하여 두 번째 입력을 검증하고 중요한 작업에 해당하는 `critical_ops()`를 실행한다. 하지만 `vuln_func()`은 4번 라인의 strcpy를 이용해 입력을 buf에 복사하는 과정에서 버퍼 오버플로우 취약점을 가지고 있어 이를 통해 공격자가 프로

```
1  int main(char ** argv)
2  {
3    ...
4    vuln_func(argv[1])
5    ...
6    if (!authenticate(argv[1])) {
7      fprintf(stderr, "Authentication fails!");
8      exit(1);
9    }
10   vuln_func(argv[2]);
11   /* Critical operations */
12   critical_ops();
     ...
```

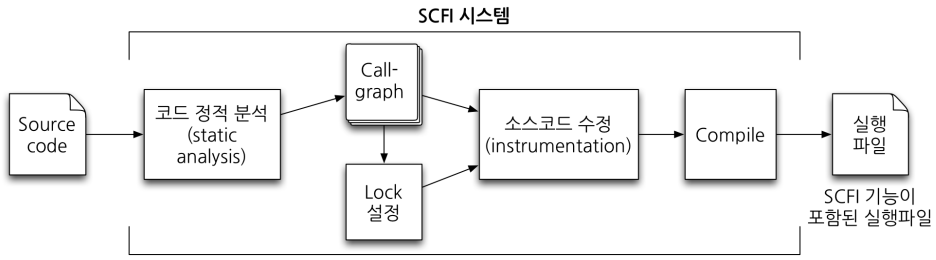
(a) main 함수

```
1  void vuln_func(char * str)
2  {
3    char buf[8];
4    strcpy(buf, str); /* Buffer overflow */
5    if (!verify_1(buf)) exit(1);
6  }
```

(b) vuln_func 함수

[그림 2] 기존 기법에서 보호되지 않는 프로그램의 예

그램의 제어를 가로챌 수 있다. SFI가 적용이 되는 가정에서도 CFI나 CFL 기능이 제공되지 않는다면 공격자는 입력 값을 조정하여 첫 번째 `vuln_func()`에서 복귀하는 순간에 프로그램 내에 있는 아무 함수나 수행할 수 있게 된다. 이 때 CFI와 CFL을 적용하면 공격자의 공격을 제한할 수 있지만, 문맥을 인지하는 정밀도가 떨어지기 때문에 제한적인 공격이 가능하다. 예제의 `main()` 함수에서 `vuln_func()`은 `authentication()` 함수를 사이에 두고 두 번 호출되지만, 기존의 기법에서는 두 호출을 구분하지 않아 같은 lock 값을 설정하게 된다. 정상적인 상황에서 첫 번째 호출이 끝나고 나면 프로그램의 제어는 `main()`의 5번째 라인으로 가야하지만, 공격자의 `vuln_func()`에 대한 공격을 통해서 첫 번째 호출이 끝나고 두 번째 호출이 끝나고 돌아가야 할 `main()`의 11번 라인으로 제어를 옮길 수 있다. 특히 기존 기법에서 두 호출에 대한 lock이 같기 때문에 이러한 비정상적인 제어 이동이 탐지되지 않는다. 이렇게 공격자는 `authenticate()`의 실행을 우회하여 인증과정 없이 `critical_ops()`를 실행시킬 수 있다. (4.2절에서 이러한 공격에 대한 실험과 결과를 더 자세히 설명한다.)



〔그림 3〕 SCFI 시스템 동작 과정

III. SCFI : 제안하는 CFI 기법

3.1 개요 및 시스템 모델

앞서 설명한 CFI의 문제점들을 해결하기 위하여 SCFI 기법을 제안한다. SCFI는 소스코드를 대상으로 하는 CFL을 바탕으로 하여 함수 문맥을 인지하고 모든 제어 이동에 대하여 유일한 lock을 제공하는 문맥을 인지하는 특성을 가진다. 〔그림 1〕(c)와 같이 SCFI는 함수 호출이 일어나기 전에 함수 호출마다 유일한 lock 값을 설정한다. 즉 함수 f()가 g()와 h()에서 호출된다고 하더라도 g()에서 f()를 호출할 때의 lock 값과 h()에서 호출하는 lock 값이 다르게 된다(〔그림 1〕(c)에서 Lock_1과 Lock_2). 또한 f() 함수에서 복귀하는 경우에도 호출되었을 때 사용한 lock값을 가공한 새 lock 값을 설정하여 사용하므로 (Lock_1′과 Lock_2′) 문맥을 인지하여 함수 호출과 복귀 모두에 대한 lock 설정이 가능하다. 따라서 SCFI는 프로그램 제어에 대한 기존의 공격들뿐만 아니라 2.3절에서 우려하는 CFI와 CFL에 대한 공격들도 방어하는 높은 정밀도의 보호기법을 제공한다.

기존의 CFL과 같이 SCFI는 저수준의 비정상적인 제어 이동을 방지하는 SFI 기능과 함께 이용하는 것을 가정한다. SFI는 프로그램이 이동이 함수 시작이나 함수 호출이 일어난 곳과 같이 소스코드 수준에서 인지할 수 있는 제어만이 일어나는 것을 보장한다. 또한 공격자의 능력은 버퍼 오버플로우 같은 공격을 통해서 소프트웨어의 제어 데이터(예를 들어 stack frame에 저장되어 있는 복귀 주소)에 대한 모든 공격이 가능함을 가정한다.

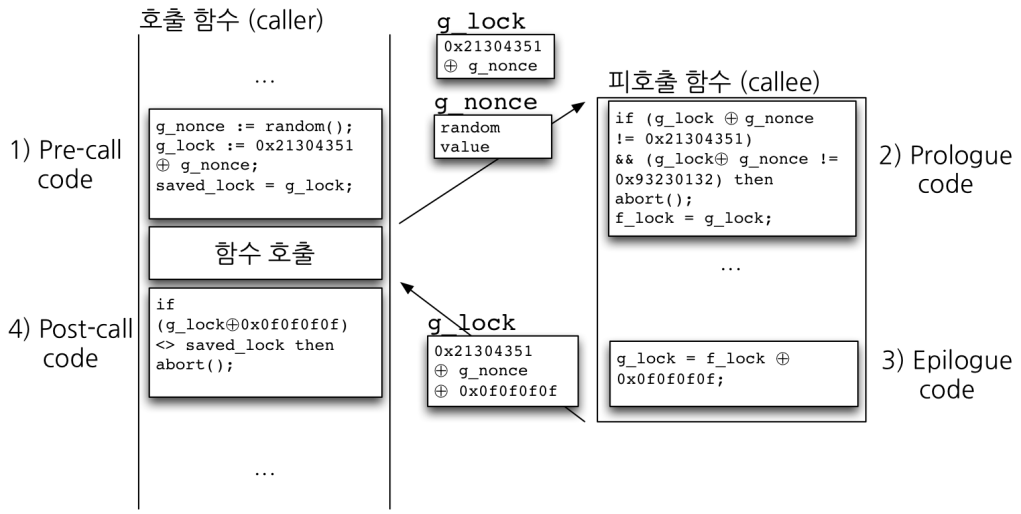
3.2 SCFI 시스템 실행 과정

제안하는 SCFI 시스템의 실행 과정은 〔그림 3〕에 나타나 있다. 개발된 시스템은 소스코드를 입력받아서 정적 분석을 실시하여 call graph와 같은 정보를 수집하고 lock 값을 설정한다. 분석된 정보를 바탕으로 소스코드를 직접 SCFI 기능을 사용하도록 수정 후 컴파일하여 최종적으로 SCFI에 의해서 보호받는 실행파일을 얻을 수 있다. 이러한 과정은 C코드 분석 시스템인 CIL〔6〕 시스템을 기반으로 개발되어 전체의 과정이 자동으로 수행된다. 구현과 관련된 자세한 내용은 4절에서 설명하고 있으며 각 단계에 대한 설명은 다음과 같다.

1. 입력: SCFI 시스템은 프로그램의 소스 코드를 입력받아 작업을 시작한다.
2. 정적 프로그램 분석: 정적 분석을 통하여 소스코드 내에 있는 모든 제어권 이동을 찾아낸다. 특히 함수의 호출 관계를 나타내는 call graph를 생성하여 함수들 사이의 제어 이동을 분석한다.
3. Lock 설정: Call graph를 참조하여 분석된 제어 이동을 대상으로 lock 값을 설정한다. Lock 값은 call graph내의 모든 함수 호출에 대하여 개별적으로 유일한 값을 설정하여 문맥을 인지하는 특성을 가지도록 한다.
4. 소스코드 수정: 할당된 lock값을 설정하고 제어 이동 전후에 검사하는 코드를 입력받은 소스코드에 추가한다. 검사 코드는 prologue check, epilogue check, pre-call check, post-call check의 네 종류의 코드가 있으며 각 코드를 주어진 위치에 추가한다. 각 검사 코드에 대해서는 이후에 설명한다.
5. 컴파일: 수정된 소스코드를 컴파일하여 SCFI 기능을 지원하는 실행파일을 생성한다.

3.3 문맥을 인지하는(Context-sensitive) Lock 설정

문맥을 인지하는 lock이 가능하게 하기 위해서는 하나의 함수에 대한 호출이더라도 호출한 함수에 따라



〔그림 4〕 SCFI 시스템의 상세 동작과 자동적으로 주입된 코드의 의사 코드(psuedo code)

서 호출할 때와 return을 통해 복귀할 때 다른 lock에 의해서 보호 받아야 한다. 이를 위해 우선 call graph의 함수 호출에 해당하는 제어 이동(call graph에서는 함수 호출 edge)에 유일한 lock 값을 설정한다(유일한 lock값을 설정하기 위해서는 의사난수발생기를 이용한다.). Return 명령을 통한 복귀 과정에서의 제어 이동에 대해서는 함수 호출을 위해 할당 받은 lock 값에 특정한 연산을 통해 가공한 값을 사용한다. 본 논문에서는 함수 호출 과정에서 설정된 lock 값에 상수값 0x0f0f0f0f을 XOR하여 사용한다.

프로그램 수행 중에 제어 이동이 일어나기 직전 lock 값이 설정되는 변수는 공격자의 공격이 닿지 않는 위치 (예를 들어, 다른 segment, heap, 또는 global)에 저장되어 안전하게 지켜진다. 하지만 미리 계산된 lock 값은 프로그램 수정 과정에서 코드에 주입이 되기 때문에 공격자가 소프트웨어 바이너리 역공학(reverse engineering) 기법을 이용하여 소프트웨어 코드 분석을 통해 lock 값을 알아낼 수 있다. 이러한 상황을 대비하여 프로그램 수행 중 lock 변수에 lock 값을 설정할 때 nonce의 역할을 하는 난수를 생성하여 추가해 사용한다. 생성된 nonce는 XOR로 lock 값에 연산하여 적용한다(lock ⊕ nonce). 즉, lock 값은 프로그램의 정적 분석과정에서 미리 정해져서 변하지 않는 값이지만(소스코드 자체에 정해진 값으로 추가), nonce는 프로그램의 실행 중에 생성되는 난수 값이다. 특히 난수를 추가하여 추가된 난수가 명시되지 않은 간접 호출과 같은 lock 값을 특정하여 정할 수 없어 0으로 놓아야 하는 경우에도 적용되어 호출 후 복귀에 해당하는 제어 이동의 안전성을 보장할 수 있게 된다. 따라서 nonce를 통하여 문맥을 인지하는 lock 검사가 항상 가능하게 된다.

Lock 값을 저장하는 lock 변수와 실행 중에 생성된 nonce 값을 저장하는 nonce 변수는 사용자의 일차적인 공격 범위에서 벗어나고 함수 호출 관계와 상관없이 접근이 가능해야 하므로 global 변수를 이용한다. 공격자의 공격 범위가 늘어난 경우도 고려하여 lock 변수와 nonce 변수를 heap이나 별도의 section에 위치하는 방법을 적용할 수도 있다.

3.4 주입 코드

정적 분석 단계가 끝나고 나면 SCFI 시스템에서는 프로그램 소스코드의 각 위치에 주어진 lock 값을 설정하고 검사하는 코드가 자동으로 주입된다. 추가되는 코드는 크게 4 종류로 함수의 시작에 주입되는 prologue code, 함수가 종료 바로 앞에 주입되는 epilogue code, 함수 호출 직전에 주입되는 pre-call code, 그리고 함수 호출 직후에 주입되는 post-call code가 있다. 다음은 4 종류의 주입 코드에 대한 설명이다.

1) pre-call code (호출자, caller): 다른 함수를 호출하기 직전에 수행된다. 함수를 호출하기 위해서 할당된 lock값에 난수를 XOR하여 lock 변수에 값을 저장하고 nonce 변수에 난수를 저

장한다. Lock 변수는 제어 이동이 있을 때마다 재사용되므로 함수가 호출에서 돌아온 이후의 값을 검사하기 위하여 현재 lock 변수에 저장된 값을 자신의 호출용 지역(local) 저장소에 저장한다.

2) prologue code (피호출자, callee): 함수가 호출된 직후 자신이 정당하게 호출되었는지를 확인한다. Lock 변수에 저장된 값과 nonce 변수에 저장된 값을 XOR하여 자신에게 할당된 lock 값 중에 하나와 일치하는지 확인한다. (함수는 한번 이상의 호출을 당할 수 있으므로 자신에 해당되는 lock 값을 하나 이상 가진다.) 값이 일치하면 Lock 변수의 값을 자신의 피호출용 지역 저장소에 저장하고 일치하지 않는 경우 프로그램 실행을 중단한다.
3) epilogue code (피호출자, callee): Return을 통해 함수가 종료되기 전에 안전한 제어이동을 준비한다. 자신의 피호출용 지역 저장소에 저장되어있는 lock 값에 0x0f0f0f0f를 XOR하여 lock 변수에 할당하고 return을 수행한다.
4) post-call code (호출자, caller): 호출한 함수로부터 정당하게 제어 이동을 받았는지 확인한다. Lock 변수를 통해 넘겨받은 lock 값에 0x0f0f0f0f를 XOR하여 자신의 호출용 지역 저장소에 저장되어있는 lock 값과 동일한지 확인한다. 두 값이 일치한다면 정상적인 제어 이동을 한 것이므로 lock 변수와 지역 저장소를 초기화하고 일치하지 않으면 프로그램 수행을 중단한다.

〔그림 4〕는 SCI 시스템의 상세 동작 과정에의 예이다. 호출 함수에서 함수 호출이 있기 전에 pre-call code가 수행된다. 그림의 예에서 함수 호출에 대한 lock 값으로 0x21304351이 설정되었다. pre-call code에서는 난수를 생성하여 설정된 lock 값에 XOR하여 global 변수인 g_lock에 저장되고 생성된 난수는 역시 global 변수인 g_nonce에 저장된다. g_lock과 g_nonce는 제어 이동이 일어날 때 마다 재사용되므로, 호출 이후에 현재의 lock 값을 이용해 적법한 제어 이동을 확인하기 위해서 현재의 lock 값을 자신의 호출용 지역 저장소인 saved_lock변수에 저장한다. 함수 호출이 일어나고 난 후 피호출함수에서는 이 값을 검사하는 prologue code가 수행된다. global 변수 g_lock에 저장되어 있는 값에 다시 g_nonce를 XOR하여 해당 함수를 위해 해당 함수 호출을 위해서 정해져 있는 lock값 0x21304351과 비교한다. (여기에서는 피호출함수가 프로그램 내에서 두 번 호출되기 때문에 두 lock값 0x21304351과 0x9323013에 대해서 모두 비교하고 있다.) 동일한 결과를 얻을 수 있으면 g_lock 변수의 값을 함수의 피호출용 지역 저장소인 f_lock에 저장한다. 피호출함수가 모두 수행되고 원래의 함수로 return하기 이전에 f_lock에 저장되어 있는 값을 0x0f0f0f0f를 XOR하여 다시 g_lock으로 옮김으로써 lock을 호출 함수로 전달한다. 호출 함수에서는 g_lock 변수에 다시 0x0f0f0f0f를 XOR하고 이 결과가 저장했던 saved_lock과 같은지를 비교하여 제어 이동이 올바르게 이루어 졌음을 확인한다.

IV. 구현 및 보안 고려사항

4.1 구현

SCFI는 CIL 시스템〔6〕을 기반으로 자동화된 시스템을 구현하였다. CIL 시스템은 C언어 코드를 대상으로 한 분석 및 수정 시스템으로써 소스코드를 분석이 용이한 CIL이라고 불리는 중간 언어(intermediate language)로 변환하고 중간 언어에 대한 분석 플랫폼을 제공한다. 중간 언어 상태에서 분석되고 수정된 프로그램은 다시 C언어의 소스코드로 변환 후 컴파일되어 원하는 기능이 추가된 실행파일을 얻을 수 있다. 이를 이용해 SCFI는 CIL로 변환된 소스코드를 대상으로 프로그램 구성을 분석하고 lock 값을 생성한 후 SCFI 기능에 해당하는 코드를 생성 후 주입하여 기존의 소스코드와 함께 동작할 수 있도록 조정한다. 수정된 CIL 코드는 다시 C 언어 코드로 복원 후에 컴파일 된다. 특히 기존에 제안된 CFL 처럼 실제적인 동작 방식이 모호한 문제점을 피하기 위하여, 이론적으로 잘 정의되어 있는 CIL을 바탕으로 한 구현을 통해 명확한 동작 과정을 정립하였다. 전체적인 동작과정은 3.2절의 내용을 따르고 있지만, 시스템 구현 과정에서 다음과 같은 사항들이 고려되어야 한다.

Call graph 생성과 lock 값의 설정: SCFI에서 사용할 lock 값을 할당하기 위해서는 call graph 생성과 lock 값의 설정이라는 두 단계의 정적 분석이 필요하다. 우선 소스코드 내의 함수 호출 관계를 모두 찾아서 호출 관계도인 call graph를 완성한다. 현재

SCFI 시스템은 컴파일이 가능한 소스코드 내의 함수들을 보호 대상으로 삼고 있기 때문에 call graph 분석과 함께 해당되는 함수가 소스코드 내에서 정의된 함수인지 외부 함수(라이브러리 함수)인지도 함께 판단한다. Lock 설정은 이러한 정보를 모두 이용하여 할당되어야 하므로 별도의 단계로 진행된다. Call graph에서 분석된 결과를 대상으로 모든 함수 호출을 구분하여 외부 함수에 대한 호출이 아닌 경우 유일한 lock값을 생성하여 할당한다. Lock 값 생성시에는 의사 난수 발생기를 이용한다.

생성 코드 주입: Lock 값이 설정된 이후에는 3.4절에 설명된 4가지 코드를 소스코드에 주입한다. 각 코드는 포함되어야 하는 위치에 따라서 다른 방식으로 주입된다. Prologue code는 함수의 맨처음에 오되 local 변수들이 선언된 이후에 와야 하므로 CIL에서 정의된 function descriptor의 function body 사이에 주입한다. 이와 반대로 epilogue code는 피호출함수가 수행을 마치고 호출함수로 돌아가는 모든 상황에서 실행되어야 하므로, 함수 내부의 모든 `Return` statement를 찾아 epilogue code가 포함된 return으로 대체한다. Pre-call code와 post-call code의 경우에는 statement내의 lock 값이 정의된 모든 `Call` instruction을 찾아서 함수 호출 전에는 pre-call code가 호출 후에는 post-call code가 수행되는 새로운 블록(block)을 생성하여 대체하는 방식으로 코드를 주입한다.

4.2 실험 및 보안성

SCFI의 결과를 확인하기 위하여 2.3절의 〔그림 2〕의 프로그램을 대상으로 실험을 수행하였다. 〔그림 2〕에서 볼 수 있듯이 버퍼 오버플로우의 약점을 가지고 있는 함수 `vuln_func()`이 `authenticate()` 함수를 전후로 하여 실행되고 정상적으로 인증이 가능한 입력이라면 `critical_ops()`란 함수가 실행된다. 공격자의 목적은 `vuln_func()`의 취약점을 이용하여 `critical_ops()` 함수를 실행시키는 것으로써, SFI가 적용된다는 가정하에서 1) `vuln_func()`의 복귀 주소를 `critical_ops()` 함수의 주소로 덮어쓰거나 2) 2.3절에서 설명한 방식으로 처음 호출되는 `vuln_func()` 함수의 복귀 주소를 `authenticate()` 함수 뒤에 호출되는 `vuln_func()`의 복귀 주소로 덮어쓰는 두 가지 공격 방법이 있다.

〔그림 5〕는 이러한 두 가지 공격을 CFI나 CFL 기능이 전혀 없는 경우, 기존의 CFL 기법이 적용된 경우, 그리고 제안하는 SCFI가 적용된 경우의 세가지 경우에 대하여 수행했을 때의 예이다. 각 경우마다 생성된 바이너리 실행파일이 조금씩 다르기 때문에 공격에 사용되는 payload는 조금씩 달라지지만 모두

공격 1	`$ ./s `python -c 'print "A" * 20 + "\x33\x86\x04\x08"'` a` **`This is critical_ops()`** **`Segmentation fault`**
공격 2	`$ ./s `python -c 'print "A" * 20 + "\x14\x86\x04\x08"'` a` **`This is critical_ops()`** **`Segmentation fault`**

(a) CFI나 CFL이 적용되지 않는 경우. 두 공격이 모두 성공한다.
(critical_op()의 주소: 0x8048533, 두 번째 vuln_func()의 retrun 주소: 0x8048614)

공격 1	`$ ./s_cfl `python -c 'print "A" * 20 + "\xbb\x85\x04\x08"'` a` **`Error: control flow violation`** **`Aborted`**
공격 2	`$ ./s_cfl `python -c 'print "A" * 20 + "\xa7\x87\x04\x08"'` a` **`This is critical_ops()`** **`Segmentation fault`**

(b) 기존 CFL이 적용된 경우. 공격1은 실패하지만 공격2는 성공한다.
(critical_op()의 주소: 0x80485bb, 두 번째 vuln_func()의 retrun 주소: 0x80486a7)

공격 1	`$ ./s_scfi `python -c 'print "A" * 28 + "\x3a\x86\x04\x08"'` a` **`Error: control flow violation`** **`Aborted`**
공격 2	`$ ./s_scfi `python -c 'print "A" * 28 + "\x52\x88\x04\x08"'` a` **`Error: control flow violation`** **`Aborted`**

(c) 제안하는 SCFI가 적용된 경우. 모든 공격을 방어한다.
(critical_op()의 주소: 0x804863a, 두 번째 vuln_func()의 retrun 주소: 0x8048852)

〔그림 5〕 CFL과 SCFI 적용에 따른 프로그램의 두 가지 공격에 대한 방어 결과

vuln_func()의 버퍼 오버플로우 취약점을 이용하여 첫 번째 공격에서는 critical_ops() 함수의 주소를, 그리고 두 번째 공격에서는 authenticate()이후에 위치한 vuln_func()의 복귀 주소를, 처음으로 호출되는 vuln_func()의 복귀 주소에 덮어쓰는 공격을 수행하였다. (critical_ops() 함수는 "This is critical_ops()"란 문자열을 출력하므로 해당 문자열이 출력되었으면 공격이 성공된 것으로 가정한다.)

〔그림 5〕에서 볼 수 있듯이 CFI나 CFL이 적용되지 않는 경우에는 두 가지 공격이 모두 성공하여 critical_ops() 함수가 실행된 것을 볼 수 있다. CFL이 적용된 경우 강제적으로 critical_ops() 함수를 실행하려는 공격은 실패하였지만, 2.3 절에 설명된 것 같이 문맥을 인지하지 못하는 특성을 이용한 공격인 두 번째 공격은 성공하는 것을 볼 수 있다. 하지만 제안하는 SCFI 기법의 경우에는 이러한 모든 공격이 실패하는 것을 확인할 수 있다.

〔표 1〕 SCFI 적용 후 프로그램의 크기 변화

원본 프로그램 크기 (bytes)	SCFI 적용 후 크기 (bytes)	증가율
7479	7635	2.77 %

이와 함께 〔표 1〕에서 원본 프로그램과 SCFI 프로그램을 적용한 후의 실행파일의 크기를 비교하였다. SCFI를 적용한 후 실행파일의 크기는 2.77% 정도로 적은 양이 증가한 것으로 나타났다. 즉 SCFI는 적은 양의 코드 증가를 통하여 프로그램의 안전성을 비약적으로 높일 수 있다.

V. 토의 및 향후 연구 방향

5.1 다중 Lock 값들의 비교

SCFI에서 어떠한 내부 함수가 한번 이상 호출되게 되게 되면 해당 함수에 대한 lock값이 하나 이상 설정되어 다중의 lock 값을 가지게 된다. 이러한 경우 함수의 prologue code에서 현재의 lock 값과 자신에게 할당된 다중의 lock 값들을 비교해야 한다. Lock 값을 비교하는 방법은 다중의 lock들을 하나의 배열로 만들어 저장하고 for문과 같은 루프를 통해서 비교하는 방법이 코드양의 증가가 적다는 점에서는 유리하지만, lock 값이 데이터 영역에 저장된다는 측면에서 혹시라도 모를 공격의 가능성을 배제하기 위하여 바람직하지 않다. 다른 방법은 if-then-else 체인을 이용하여 다중 lock 값을 하나씩 비교하는 방식이다. 이 방식은 lock 값들이 직접 코드에 상수 값으로 명시되기 때문에 공격에서 더 안전하지만 코드의 양이 더 증가한다는 단점을 가지고 있다. SCFI에서는 더 나은 안전성을 위하여 두 번째 방법을 사용한다.

하지만, if-then-else의 체인을 이용한 방법은 어떠한 함수가 호출되는 횟수에 비례하는 O(n)의 검색 비용을 필요로 하기 때문에 빈번하게 사용되는 함수에게는 비효율적이다. 호출되는 횟수가 많은 함수에 대해서는 부가적인 코드양의 증가를 감수하고 이진 검색(binary search)형태로 lock 값을 검색하는 코드를 주입하거나, 정밀도 저하를 감수하고 다중 lock 값을 하나로 병합(merge)하여 사용되는 lock 값의 개수를 줄이는 방법을 적용할 수 있다.

5.2 간접 호출 함수와 호출되지 않는 함수

SCFI에서 사용하는 call graph의 정확도를 저하시키는 주요 원인은 간접 호출 함수 이다. 함수 포인터등을 이용한 간접 호출 함수의 경우에는 함수 타입이 허용하는 범위에서 어떠한 함수도 호출할 수 있기 때문에 정확한 호출 관계를 분석하기 어려워진다. 따라서 SCFI에서는 프로그램 내에서 어떠한 함수의 주소가 사용되는 경우 해당 함수를 '간접 호출 가능 함수'라고 가정을 하고, 간접 호출이 있는 경우에는 lock값은 설정하지 않고(lock = 0) nonce만 설정함으로써 호출-피호출 관계를 유지하는 방법을 사용한다. 이러한 방법은 어떤 함수가 한번 호출이 되고 나면 꼭 자신을 호출한 함수로 복귀하도록 강제하는 보호 효과는 여전히 가지고 있지만, 공격자가 함수 포인터 자체를 덮어씀으로써 불리지 말아야할 다른 간접 호출 가능 함수를 호출하는 공격은 방어가 어렵다. 특히 프로그램내에 어떠한 함수로부터라도 직접적으로 호출이 되지 않는 함수가 있는 경우, 이러한 함수를 간접 호출 가능 함수로 구분지어야 하는지 아니면 보안상의 목적으로 실행되지 않도록 해야 하는지에 대한 여부를 결정하기 어렵다. 따라서 이러한 간접 호출 함수에서 발생할 수 있는 문제를 해소하기 위하여 간접 호출 가능 함수에 대한 지시어를 추가하고 간접 함수 호출에 대한 call graph의 정확도를 높이는 후속 연구를 현재 진행 중이다.

VI. 결론

기존의 CFI와 CFL 기법은 프로그램 수정을 통하여 프로그램 내부에서 적법한 제어 이동만 일어날 수 있도록 강제한다는 측면에서 많은 공격들로부터 소프트웨어를 방어할 수 있는 좋은 방법을 제시하였다. 하지만 기존의 기법이 함수 호출 문맥을 고려하지 않는 한계점 때문에 방어의 정밀도가 떨어져 제한적인 공격이 가능한 문제점을 가지고 있었다. 본 논문에서는 이러한 문제들을 해결한 SCFI 기법을 제안한다. SCFI 시스템은 문맥을 인지한 lock을 설정하는 향상된 정밀도의 방어 기법을 자동으로 프로그램에 주입하여 프로그램의 안전성을 확보한다.

참고문헌

〔1〕 Address space layout randomization, http://pax.grsecurity.net/docs/aslr.txt

〔2〕 R. Wahbe, S. Lucco, T. Anderson, and S. Graham, "Efficient software-based fault isolation," Proceedings of the 14th ACM symposium on Operating systems principles (SOSP), pp. 203-216, Dec. 1993.

〔3〕 S. McCamant and G. Morrisett. "Efficient, verifiable binary sandboxing for a CISC architecture," MIT Technical Report MIT-CSAIL-TR-2005-030, MIT, May. 2005.

〔4〕 M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications," ACM Transactions on Information and System Security (TISSEC), vol. 13, no. 1, pp. 1-40, Oct. 2009.

〔5〕 T. Bletsch, X. Jiang, and V. Freeh, "Mitigating code-reuse attacks with control-flow locking," Proceedings of the 27th Annual Computer Security Applications Conference, pp. 353 - 362, Dec. 2011.

〔6〕 G. Necula, S. McPeak, and S. Rahul, "CIL: Intermediate language and tools for analysis and transformation of C programs," Proceedings of International Conference on Compiler Construction, pp. 213-228, Jan. 2002.

〔7〕 이종혁, "단편화된 실행파일을 위한 데이터 구조 역공학 기법," 한국정보보호학회논문지, 22(3), pp. 615-619, 2012년 6월.

〈著者紹介〉

이 종 협 (JongHyup Lee) 종신회원
2002년 2월: 연세대학교 기계전자공학부 졸업
2004년 2월: 연세대학교 컴퓨터과학과 석사
2009년 8월: 연세대학교 컴퓨터과학과 박사
2009년~2012년: Carnegie Mellon University, CyLab, Postdoc. 연구원
2012년 3월~현재: 한국교통대학교 소프트웨어학과 조교수
〈관심분야〉 소프트웨어 보안, 네트워크 보안

김 용 승 (Yong Seung Kim) 일반회원
1980년: 숭실대학교 전자계산학과 학사
1983년: 숭실대학원 전자계산학과 석사
1998년: 숭실대학원 전자계산학과 박사
1980년~1983년: 충북대학교 전자계산소 조교
1983년~1991년: 충주공업전문대학 전자계산과 부교수
1991년~2006년: 청주과학대학 컴퓨터과학과 교수
2006년 ~현재: 한국교통대학교 소프트웨어학과 교수
〈관심분야〉 컴퓨터 시스템, 소프트웨어 개발, 네트워크, 프로그램 로직