# 안전하고 효율적인 Code Reuse Attack 탐지를 위한 ARM 프로세서의 두 가지 명령어 세트를 고려한 Meta-data 생성 기술

허인구*, 한상준*, 이진용*, 백윤흥*
*서울대학교 전기정보공학부
e-mail : igheo@sor.snu.ac.kr, sjhan@sor.snu.ac.kr, jylee@sor.snu.ac.kr, ypaek@sor.snu.ac.kr

# A Meta-data Generation Technique for Efficient and Secure Code Reuse Attack Detection with a Consideration on Two Types of Instruction Set

Ingeo Heo*, Sangjun Han*, Jinyong Lee* and Yunheung Paek*
*Dept. of Electrical and Computer Engineering, Seoul National University

**요 약**

Code reuse attack (CRA)는 기존의 코드 내에서 필요한 코드 조각들 (gadgets)을 모아 indirect branch 명령어들로 잇는 방식으로 공격자가 원하는 악성 프로그램을 구성할 수 있는 강력한 공격 방법이다. 공격자는 자신의 코드를 대상 시스템에 심는 대신 기존의 코드를 이용하기 때문에, 대부분의 범용 운영체제 (OS)가 강제하는 W^X protection 을 무력화할 수 있다. 이러한 CRA 에 대응하기 위하여 다수의 연구들에서 branch 의 trace 를 분석하여 CRA 고유의 특성을 찾아내는 Signature 기반 탐지 기술을 제안하였다. 본 논문에서는 ARM 프로세서 상에서의 CRA 를 대응하기 위한 Signature 기반 탐지 기술을 효율적으로 도울 수 있는 binary 분석 및 meta-data 생성 기술을 제안한다. 특히, 본 논문은 우리의 이전 논문에서 고려 되지 못했던 ARM 의 두 가지 명령어 세트의 특성을 고려하여, 공격자가 어느 명령어 세트를 이용하여 CRA 를 시도하더라도 막아낼 수 있도록 meta-data 를 두 가지 mode 에 대해서 생성하였다. 실험 결과, meta-data 는 본래 바이너리 코드 대비 20.8% 정도의 크기 증가를 일으키는 것으로 나타났다.

## 1. 서론

근래 들어, Code Reuse Attack (CRA)이라는 새로운 공격 기술이 많은 컴퓨터 시스템들을 위협 하고 있다. 이러한 CRA 는 크게 초기 형태인 return-oriented programming (ROP)와 이를 변형한 형태인 jump-oriented programming (JOP) 으로 나뉠 수 있다. CRA 는 공통적으로, 아주 적은 3~4 개 이하의 명령어로만 구성된 gadget 이라는 코드 블록을 return 혹은 indirect branch 명령어로 연결하여, 원하는 악성 행위를 수행한다. 특히 이 gadget 은 공격자에 의해 새롭게 만들어진 코드가 아니라, 이미 시스템 상에서 수행되고 있는 다른 프로그램들의 코드 내에서 탐색하여 얻어낸 것이므로, W^X protection 혹은 DEP [1]와 같이 data 영역의 값을 코드 실행의 목적으로 사용하는 것을 방어하는 기술들로 방어가 불가능하다.

이러한 CRA 는 ARM 프로세서를 표준으로 사용하고 있는 모바일 기기들에서도 빈번하게 나타나고 있다. 이를 방어하기 위하여 몇몇 연구가 소개된 바 있으나 [2][3], return 과 관련된 CRA 의 특징에만 집중했던 해당 기술들은 모두 indirect jump 명령어를 이용해 gadget 을 연결하는 JOP 공격을 방어할 수 없었다.

최근 들어 제안된 signature-based detection 방식 [4]은 이러한 ROP/JOP 를 모두 탐지할 수 있는 방어 기법이다. 이 기법은 각 gadget 의 길이가 보통의 프로그램에서 branch 의 사이사이에서 실행되는 명령어의 개수에 비해 현저히 작다는 사실에 주목하였다. 즉, 특정 개수 이하의 명령어를 실행하는 gadget 을 몇 개 이상 연속적으로 발견할 경우, 이를 CRA 로 감지하는 것이다. 이러한 방어 기법의 유효성은 이후의 다른 논문들에서도 유효함이 밝혀진 바 있다 [5].

우리의 이전 연구 [6]에서는 이러한 signature 기반 CRA 탐지 기술을 모바일에 적용하기 위한 선행 연구로서, gadget 의 크기를 빠르게 계산할 수 있는 정적 분석 기술을 제안하였다. Signature 기반 탐지 기술에서는, 매 번 indirect branch 가 수행될 때마다, 다음 branch 가 일어나기 전까지 수행 된 명령어의 개수, 즉 gadget 의 길이를 분석하여야 한다. 하지만, runtime

에 수행된 명령어의 개수를 그 때마다 동적으로 계산할 경우, binary 를 runtime 에 다시 읽거나, 수행된 명령어의 주소들을 비교해야 하는 등, 상당한 계산을 요구하게 된다. 이러한 계산을 줄이기 위하여, 이전 연구에서는 각 명령어의 주소 별로, 해당 지점으로 control flow 가 이동 되었을 경우에, 다음 branch 지점까지 수행되는 명령어의 개수를 meta-data 형태로 관리하였다. 이 meta-data 는 결국 CRA 가 수행하게 되는 개별 gadget 들의 길이가 되므로, runtime 에 이 meta-data 를 indirect branch 마다 읽기만 하면, gadget 의 길이를 빠른 시간 안에 얻어낼 수 있다.

하지만, 이전 연구에서 고려하지 못하였던 것은, 우리가 대상으로 한 ARM 프로세서가 두 가지 명령어 세트 모드인 ARM 과 THUMB 모드를 지원한다는 사실이다. ARM 은 기본적인 32-bit 명령어 세트인 ARM 모드 이외에도, 16-bit/32-bit 가 모두 가능한 THUMB 모드를 동시에 지원한다. 하지만 이전 연구에서는 ARM 모드에 대한 meta-data 만을 생성하였기 때문에, 공격자가 자신의 gadget 을 THUMB 모드를 가정해 수집할 경우, 우리의 meta-data 가 올바른 정보를 줄 수 없다는 문제가 있었다. 특히 최근의 연구들에 의하면 [7], ARM 모드로 compile 된 바이너리 코드를 강제로 THUMB 모드로 읽음으로써, 프로그래머가 의도하지 않았던 명령어를 찾아내 공격에 필요한 gadget 으로 사용하는 것이 가능한 것으로 밝혀졌다.

따라서, 본 논문에서는 meta-data 를 ARM/THUMB 두 가지 명령어 모드를 모두 가정하여 생성하는 기술을 제안한다. Signature 기반 CRA 탐지를 수행할 시에, 하나의 indirect branch 가 수행될 경우, 현재의 명령어 세트 모드에 따라 해당하는 meta-data 에 접근한다. 즉, 공격자가 ARM 모드가 아닌 THUMB 모드를 이용하여 gadget 을 실행하더라도, THUMB 모드를 가정하여 생성된 meta-data 를 사용하므로, 제대로 된 gadget 길이를 얻을 수 있다. 실험 결과, 이렇게 생성된 meta-data 는 기존 바이너리 코드의 20.8% 정도의 크기를 가지는 것으로 나타났다. 하지만, 이러한 meta-data 를 정적 분석을 통해 미리 생성해둠으로써, runtime 에 필요한 계산 및 오버헤드를 줄일 수 있다.

## 2. Code Reuse Attack 과 Signature 기반 탐지 기술

그림 1 은 기존의 코드들을 재활용하여, JOP 스타일의 CRA 가 수행될 때의 프로그램 실행 흐름을 나타내고 있다. 공격자가 원하는 코드를 수행해 줄 gadget 들이 연결되어 있고, gadget 사이 사이에는 jump 할 주소를 업데이트 해주는 역할을 하는 trampoline gadget 들이 위치하고 있다. CRA 는 자신이 원하는 코드를 찾기 위해, 기존의 라이브러리나 다른 프로그램의 코드에서 필요한 gadget 들을 찾아 낸다. 하지만 일반적으로, 자신이 원하는 동작을 할 수 있는 gadget 의 수는 매우 적기 때문에, 필요한 동작 별로 매우 짧은 길이의 primitive 한 gadget 들을 찾아 놓고, 이를 수 십 개 이상 연결하여 수행한다. 이는 자신이 만든 코드를 수행하는 방식이 아니라, 기존 코드를 조합하여 만들어야 하기 때문에 생기는 불편함이다. 따라서, CRA 가 사용하는 gadget 은 add, sub, load, store 등, 범용적으로 쓰일 수 있는 명령어 1~2 개와, gadget 간의 연결을 위해 필요한 indirect branch 하나로 조합되는 것이 일반적이다. Gadget 을 조합하는 기술이 뛰어날 경우, 하나의 gadget 에서 여러 개의 명령어를 한꺼번에 배치하여 사용하는 것이 가능하지만, 이는 수행되는 명령어들끼리 원치 않는 side effect 를 일으켜 제어를 힘들게 만들 가능성이 높기 때문에 매우 어렵다. 이 때문에, 일반적인 프로그램과 달리 CRA 는 그림 1 과 같이 매우 적은 수의 명령어 만을 수행한 뒤 지속적으로 indirect branch 를 수행하는 방식으로 동작할 수 밖에 없다.

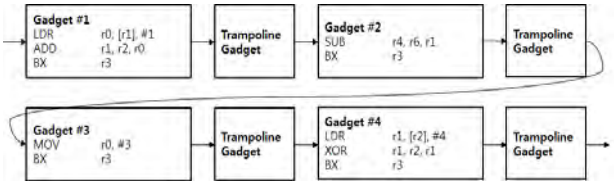


**그림 1 CRA 공격 예제**

Signature 기반 탐지는 이러한 특성을 관찰함으로써 CRA 를 탐지하는 기술이다. 그림 1 의 예제에서 trampoline gadget 의 크기를 4 로 가정할 때(일반적으로 trampoline 역시 code 를 재활용하는 것이므로 크기가 크지 않다.) 각각 2-4-1-4-1-4-2-4 의 크기를 가지는 코드 block 들이 수행되었고, 이들은 모두 BX 라고 하는 ARM 의 indirect 명령어로 연결되었다. 이러한 형태의 수행 흐름은 일반적인 프로그램에서는 흔하지 않기 때문에, 탐지 도구가 이런 수행 흐름 상의 특성을 발견하면, CRA 라 판단하여 해당 프로그램이 악성 행위에 활용되고 있음을 감지할 수 있다.

이 예제에서 보듯이, signature 기반 탐지 기술은 각 indirect branch 별로 수행하는 명령어의 개수를 분석해야 하므로, 명령어의 개수 혹은 gadget 의 크기를 지속적으로 계산하여야 한다. 이는 일반적으로 추가적인 계산 코드의 수행을 요구하지만, 본 논문에서 제시하고 있는 meta-data 방식을 사용하면, 별도의 계산 코드 없이 빠르게 gadget 의 크기를 얻어낼 수 있다.

## 3. ARM 의 두 가지 명령어 세트: ARM/THUMB

앞서 언급한 바와 같이 ARM 프로세서에는 일반적으로 2 가지 종류의 명령어 세트 모드가 지원된다. 첫 번째는 32-bit 로 동작하는 ARM 모드이고, 두 번째는 경우에 따라 16-bit/32-bit 로 동작하는 THUMB 모드이다. 두 가지 모드를 구분하는 방법은 프로세서 내에 위치한 current program status register (CPSR)의 T-bit 를 확인하는 것이다. 이 bit 가 0 이면 프로세서는 ARM 모드로 동작하고 1 이면 THUMB 모드로 동작한다. 프로세서의 모드를 설정하는 방법은 간단하다. “blx”나 “bx” 같은 indirect branch 명령어를 실행할 시에 target 으로 사용되는 주소의 마지막 bit 인 bit[0]에 스위치하고 싶은 모드에 해당하는 값을 쓰는 것이다. 예를 들어, THUMB 모드로 target 주소를 읽고 싶다면 bit[0]를 1 로 설정한 채 branch 를 하면 된다. 따라서, CRA 를

시도하는 공격자가 ARM 모드로 컴파일 된 바이너리를 THUMB 모드로 읽고 싶다면, 이러한 방법을 사용하게 된다.

## 4. 두 가지 명령어 세트를 지원하는 meta-data 생성

3 장에서 설명된 예에서처럼 공격자가 명령어 세트 모드를 변형하여 CRA 를 시도하는 것에 대비하기 위해 본 논문에서는 같은 주소에 대해서도 현재의 T-bit 이 0 인 경우와 1 인 경우를 모두 가정하여 ARM 모드용 meta-data 와 THUMB 모드용 meta-data 를 모두 생성하는 방식을 사용한다.

우리는 이전 연구에서 항상 32-bit 로 동작하는 ARM 모드에 대한 meta-data 생성에 대하여 소개한 바 있다. ARM 모드에 대해서는 1 차적으로 바이너리 파일에서 추출한 text section (명령어가 위치한 영역)을 가장 마지막 주소부터 순차적으로 32-bit 씩 decoding 해 간다. 이 때 해당 32-bit 명령어가 indirect branch 명령어일 경우, meta-data 를 0 으로 저장하고, 다음 매 32-bit 명령어마다 1 씩 증가시킨 값을 meta-data 로 저장한다. 그리고 다시 indirect branch 명령어를 만날 경우, 다시 0 으로 초기화한 값을 meta-data 로 저장한다. 이는 다시 말해, 해당 위치로 branch 가 되었을 경우, 다음 indirect branch 까지 수행되는 명령어의 개수를 표시하는 것이다. Runtime 에 control flow 가 해당 위치로 이동하였을 시에, 다음 branch 를 만나기 전까지는 control flow 가 변할 수 없으므로, 이러한 방법을 통해, branch target address 에 대응하는 gadget 의 길이를 meta-data 로 남겨둘 수 있는 것이다.

이번 연구에서는 THUMB 모드용 meta-data 또한 생성하는데, 이 때 ARM 모드용 meta-data 를 생성할 때와 같이 전체 명령어를 역순으로 읽어오는 방식을 사용할 수 없다. ARM 모드에서는 명령어가 모두 32-bit 으로 고정되어 있지만, THUMB 모드에서는 16-bit/32-bit 명령어가 혼용되어서 쓰이기 때문이다. THUMB 모드에서 명령어가 16-bit THUMB 명령어인지 32-bit THUMB 명령어인지 구분하기 위해서는 첫 5-bit 을 확인하면 되는데, 첫 5-bit 이 “11101”, “11110”, 또는 “11111”이면 32-bit THUMB 명령어임을 알 수 있다. 문제는 저장된 명령어에 역순으로 접근하게 되면, 지금 읽고 있는 32-bit 의 첫 16-bit 이 한 명령어의 시작부분인지, 아니면 그 이전의 32-bit THUMB 명령어의 뒷부분의 16-bit 인지 알 수 없다는 점이다. 이를 알려면 그 다음으로 읽을 32-bit (text section 상에서는 그 이전에 위치하는 32-bit)의 뒷부분의 16-bit 을 읽어야 하는데, 이 역시 한 명령어의 시작부분인지, 아니면 그 이전의 32-bit THUMB 명령어의 뒷부분의 16-bit 인지 알려면 그 다음으로 읽을 명령어 정보를 필요로 한다.

결국 어떤 한 명령어가 16-bit 인지 32-bit 인지를 구분하기 위해서는, text section 의 시작 주소부터 순차적으로 전체 명령어에 접근해야만 한다. 하지만 이 경우, 현재 명령어를 기준으로 다음에 수행될 branch 의 위치를 미리 알지 못하기 때문에, meta-data 를 생성하는 데 어려움이 따른다. 이를 해결하기 위해, 이번 연구에서는 우선 전체 명령어를 순차적으로 읽어서 16-bit/32-bit 명령어를 모두 구분하여 표시해놓은 다음에, 다시 전체 명령어를 역순으로 읽어서 meta-data 를 계산하는 방식을 제안한다. 그림 2 는 이러한 방식으로 meta-data 를 계산하는 과정과 그 결과의 예를 나타내고 있다.

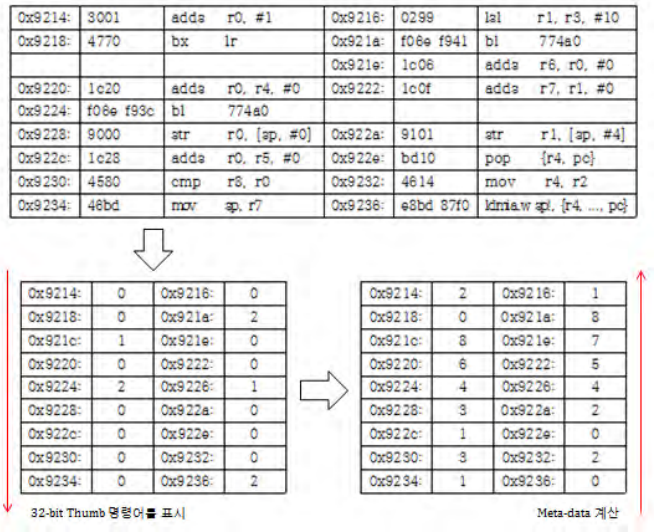


**그림 2 meta-data 를 계산하는 과정과 결과**

우선 전체 명령어를 시작 주소부터 순차적으로 읽으면서 16-bit THUMB 명령어면 0 으로, 32-bit THUMB 명령어면 앞쪽 16-bit 에는 2 로, 뒤쪽 16-bit 에는 1 로 표시해둔다. 그리고 다시 전체 명령어를 역순으로 읽으면서, 표시해둔 값이 0 이면 16-bit THUMB indirect branch 명령어인지 아닌지 확인해서 meta-data 를 계산하고, 1 이면 바로 이전의 meta-data 값에 1 을 더한 값을 저장하고, 2 이면 32-bit THUMB indirect branch 명령어인지 아닌지 확인해서 meta-data 를 계산한다. 그림 2 의 오른쪽 하단 그림은 이러한 과정의 결과를 보여주고 있다.

그림 2 에서의 예제 명령어들은 runtime 시 THUMB 모드로 동작하고, meta-data 역시 THUMB 모드용 meta-data 만을 보여주고 있지만, 실제로는 같은 주소에 대해서 ARM 모드용 meta-data 또한 계산한다. 예를 들어 0x9214 번지에는 ‘3001’이, 0x9216 번지에는 ‘0299’가 저장되어 있는데, 이를 ARM 모드 명령어인 것처럼 ‘30010299’ (mul r1, r9, r2)로 해석해서 그에 대한 meta-data 또한 계산한다. 반대로 runtime 시 ARM 모드로 동작하는 명령어에 대해서도 THUMB 모드용 meta-data 를 계산하여 저장한다. 즉, 어떤 번지에 저장되어 있는 명령어에 대해서 runtime 시에 어떤 모드로 실행되는가에 상관없이 두 가지 모드로 모두 해석해서 meta-data 를 계산한다. 이와 같이 하는 이유는 앞서 설명한대로 원래의 명령어가 runtime 시에 ARM 모드로 실행이 되게끔 만들어졌다 하더라도, 이를 CRA 공격자가 bit[0]를 1 로 설정하고, THUMB 모드용 명령어로 사용할 수 있기 때문이다.

이렇게 저장된 meta-data 사용 방식은 그림 3 과 같

다. 여러 논문에서 제안하듯이 대부분의 signature 기반 탐지 기술들은 branch trace 를 입력으로 한다. 다시 말해 indirect branch 명령어가 위치한 주소와, 그 명령어가 실행되었을 때 target 하는 주소가 CRA 탐지 도구에 주어진다. 이 중 우리의 meta-data 는 branch 의 target 주소를 입력으로 가정한다. 이 주소가 탐지 도구에 전달 되면 탐지 도구는 이 주소의 bit[0]를 확인하여 0 이면 ARM 모드용 meta-data 에, 1 이면 THUMB 모드용 meta-data 에 접근하여 gadget 길이를 얻는다.

| 모드 | 번지 | | | Meta-data | Branch Target 주소 |
|---|---|---|---|---|---|
| ARM | 0x9214: | 3001 0299 | mul r1, r9, r2 | 11 | 0x9214일 때 접근 |
| THUMB | 0x9214: | 3001 | adds r0, #1 | 2 | 0x9215일 때 접근 |
| | 0x9216: | 0299 | lsl r1, r3, #10 | 1 | 0x9217일 때 접근 |

**그림 3 meta-data 사용 방식**

## 5. 실험

본 논문에서 제안한 meta-data 생성 기술은 Linux 3.8 kernel 환경 하에서 이루어졌고, ARM 바이너리 파일은 Cortex-A9 Processor, ARMv7-A architecture 을 기준으로 하였다. ARM 바이너리 파일 생성을 위해서는 Xilinx ARM cross compiler 를 사용하였다.

이전 연구에서는 ARM 모드만을 가정했기 때문에 32-bit 명령어 하나당 1 개의 meta-data 만을 생성하였지만, 이번 연구에서는 THUMB 모드까지 고려하기 때문에 32-bit 명령어 하나당 3 개의 meta-data 를 생성하였다. 각 meta-data 는 gadget 의 길이를 나타내는데, signature 기반 탐지 기술에서는 보통 gadget 의 길이가 두 자릿수를 넘어가면 길이를 정확히 아는 것은 필요 없다. 따라서 각 meta-data 마다 4-bit 을 할당하였고, 길이가 15 가 넘어가면 15 로 saturate 하였다. 결국 32-bit 명령어 하나당 12bit 의 meta-data 가 생성되는 셈이다. 그림 4 는 SPEC benchmark 에 있는 10 가지 종류의 벤치마크에 대해서 meta-data 생성으로 인한 바이너리 파일 크기 증가를 나타내고 있다. 평균적으로 약 20.8% 정도 파일 크기가 증가한 것을 알 수 있다.

## 6. 결론 및 향후 과제

본 연구에서 제안한 meta-data 생성 기술은 우리의 이전 연구와 비교했을 때 바이너리 파일 크기 증가율이 9%에서 20.8%로 많이 증가하였지만, 이전 연구에서 고려하지 않았던 THUMB 모드까지 다룸으로써 THUMB 모드 명령어를 포함하는 바이너리 파일에 대해서도 meta-data 를 생성할 수 있다는 장점이 있다. 특히 CRA 를 시도하는 공격자는 자신이 원하는 명령어를 수월하게 찾기 위해 Encoding 이 짧은 THUMB 모드 명령어를 많이 사용한다는 점에서, 이 기술로 더 많은 CRA 를 detection 할 수 있을 것이라 기대된다.

향후 이루어져야 할 과제로, 본 연구에서 제안한 정적 분석 기술을 접목하여 Signature 기반 탐지로 CRA 를 탐지했을 시, 긍정 오류(false positive)와 부정 오류(false negative)가 어느 정도 발생하는 지에 대해서 분석할 필요가 있다.

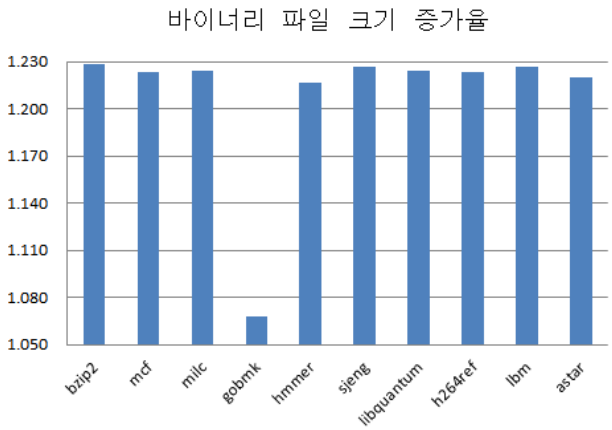


**그림 4 Meta-data 로 인한 파일 크기 오버헤드**

### Acknowledgement

본 연구는 미래창조과학부/한국연구재단 우수연구센터 육성사업 (과제번호 NRF-2008-0062609), 중소기업청에서 지원하는 2014 년도 산학연협력 기술개발사업 (No. C0218072), 2014 년도 두뇌한국 21 플러스사업, 미래창조과학부 및 한국산업기술평가관리원의 산업융합원천기술개발사업(정보통신) [No. 10047212, 1kB 이하 암호문 간의 연산을 지원하는 동형 암호 원천 기술 개발 및 응용 연구] 및 IDEC 의 EDA Tool 의 지원을 받아 수행하였습니다.

### 참고문헌

[1] Andersen, S., and V. Abella. "Data Execution Prevention. Changes to Functionality in Microsoft Windows XP Service Pack 2, Part 3: Memory Protection Technologies." (2004).

[2] Davi, Lucas, et al. "MoCFI: A framework to mitigate control-flow attacks on smartphones." Symposium on Network and Distributed System Security (NDSS). 2012.

[3] Huang, ZhiJun, Tao Zheng, and Jia Liu. "A dynamic detective method against ROP attack on ARM platform." Software Engineering for Embedded Systems (SEES), 2012 2nd International Workshop on. IEEE, 2012.

[4] Chen, Ping, et al. "DROP: Detecting return-oriented programming malicious code." Information Systems Security. Springer Berlin Heidelberg, 2009. 163-177.

[5] Pappas, Vasilis, Michalis Polychronakis, and Angelos D. Keromytis. "Transparent ROP exploit mitigation using indirect branch tracing." Proceedings of the 22nd USENIX Conference on Security. 2013.

[6] 한상준, 허인구, 백윤흥, “효율적인 Code Reuse Attack 탐지를 위한 Meta-data 생성 기술”, 정보처리학회 춘계학술 발표대회, 2014.

[7] Checkoway, Stephen, et al. "Return-oriented programming without returns." Proceedings of the 17th ACM conference on Computer and communications security. ACM, 2010.