효율적인 Code Reuse Attack 탐지를 위한 Meta-data 생성 기술

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

A Meta-data Generation Technique for Efficient Code Reuse Attack Detection

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

요 약

최근 들어, 모바일 기기의 시스템을 장악하여 중요 정보를 빼내는 등의 악성 행위를 위해 Code Reuse Attack (CRA) 이 널리 사용되고 있다. 이러한 CRA 를 막기 위한 방법으로 branch 의 trace 를 분석하여 CRA 고유의 특성을 찾아내는 Signature 기반 탐지 기술이 있다. 이러한 탐지 기술을 효율적으로 지원하기 위하여, 본 논문에서는 ARM 프로세서용 바이너리를 분석하여, signature 분석을 위해 필수적으로 분석되어야 하는 gadget 의 크기를 빠르게 접근할 수 있는 meta-data 를 생성하는 기술을 제안한다. 이러한 meta-data 를 활용하는 방식은 gadget 의 크기를 계산하는 추가적인 코드의 수행을 제거해 주므로, 더욱 효율적으로 CRA 를 탐지할 수 있도록 도와준다. 실험 결과, 이러한 meta-data 는 본래의 바이너리 코드 대비 9 % 만의 크기 증가를 일으키는 것으로 나타났다.

1. 서론

최근 들어, 애플의 iOS 혹은 구글의 안드로이드 기기를 대상으로 하는 다양한 공격들이 증가하여, 이에 따른 보안 위협 역시 증가하는 추세이다. 최근의 한 report 에 따르면 [1], 2012 년과 2013 년 사이 안드로이드를 대상으로 하는 malware 는 무려 614%나 증가했다. 이러한 malware 들은 다양한 공격 방법을 이용하여 모바일 기기 내의 중요 정보를 유출하거나, 사용자가 의도하는 동작들을 방해하는데, 이런 공격 방식 중에서도 code reuse attack (CRA)는 특히 위협적인 것으로 알려져 있다.

CRA 는 크게 예전 방식인 return-oriented programming (ROP)와 이를 보완하여 고안된 jump-oriented programming (JOP)로 나뉠 수 있다. 이러한 CRA 는 공통적으로, 아주 적은 3~4 개 이하의 명령어로만 이루어진 gadget 이라는 코드 블록을 branch 또는 jump 로 연결하여, 원하는 악성 행위를 수행하고자 한다. 이러한 방식의 공격은 기존에 시스템 상에 존재하는 코드를 재활용하기 때문에, data execution prevention (DEP)과 같이 삽입된 코드의 실행을 방지하는 방식의 방어 기법들을 무력화할 수 있어서 매우 위협적이다.

모바일 기기 상에서의 이러한 CRA 를 방어하기 위하여, 몇몇 연구[2][3]가 기존에 소개된 바 있다. 이 중 mocfi [2]는 shadow stack 이라는 기술을 활용하여, 기기 상에서 수행되는 프로그램의 control flow 를 지속적으로 감시하여, CRA 를 방어하고자 하였다. 하지만, mocfi 가 사용한 shadow stack 기술은, 프로그램이 실행하는 call-return 의 짝을 비교하여 control flow 를 감시하기 때문에, ROP 가 아닌 JOP 스타일의 공격에는 쉽게 무력화될 수 밖에 없다. 또한, shadow stack 기술은 매번 call-return 이 일어날 때마다 추가적인 코드를 수행하여야 했기 때문에, 굉장한 성능 저하를 일으켰다 (최대 4.87 배). 그러므로 mocfi 와 같은 접근 방식은 현재의 모바일 기기에 그대로 적용되기가 힘들다.

Shadow stack 방식의 한계를 뛰어넘기 위하여 DROP [4]와 같은 연구에서는 branch 의 history 를 분석하여 CRA 를 탐지하는 기술들을 제안하였다. 기본적인 아이디어는, CRA 가 매우 작은 크기의 gadget 들을 특정 수 이상 이어야만 원하는 악성 행위를 일으킬 수 있다는 것에 착안하여, indirect jump 로 이어지는 gadget 의 크기와, 연결된 길이를 signature 로 삼아 탐지하는 것이다. 이러한 방식은 이후의 다른 연구들에도 유효함이 밝혀진 바 있다 [5].

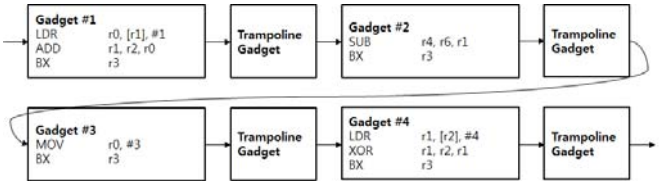
본 논문에서는 이러한 signature 기반 CRA 탐지 기술을 모바일에 적용하기 위한 선행 연구로서, gadget 의 크기를 빠르게 계산할 수 있는 정적 분석 기술을 제안한다. Signature 기반 탐지 기술에서는, 매 번 indirect branch 가 수행될 때마다, 다음 branch 가 일어나기 전까지 수행 된 명령어의 개수, 즉 gadget 의 크기를 분석하여야 한다. 하지만, runtime 에 수행된 명령

어의 개수를 그 때마다 동적으로 계산할 경우, binary 를 runtime 에 다시 읽거나, 수행된 명령어의 주소들을 비교해야 하는 등, 상당한 계산량을 요구하게 된다. 이러한 계산을 줄이기 위하여, 본 연구에서는 각 명령어의 주소 별로, 해당 지점으로 branch 가 되었을 경우에 다음 branch 지점까지 수행되는 명령어의 개수를 meta-data 형태로 관리한다. 이 meta-data 는 결국 CRA 가 수행하게 되는 gadget 의 크기가 되므로, runtime 에 이 meta-data 를 indirect branch 가 일어날 때마다 읽으면, gadget 의 크기를 빠른 시간 안에 얻어낼 수 있다. 우리의 실험에 의하면, 이러한 방식의 정적 분석 기술은 프로그램 바이너리의 크기를 6 가지 종류 벤치마크에 대하여 평균 9% 정도 증가시키는 것으로 나타났다. 하지만, 잠재적으로 일어날 수 있는 runtime 오버헤드를 크게 줄일 수 있으므로, 이러한 방식은 CRA 를 모바일 기기 상에서 효율적으로 방어하는 데 있어 매우 유용한 기술이 될 것이다.

2. Code Reuse Attack 과 Signature 기반 탐지 기술

이 장에서는, 이해를 돕기 위하여 CRA 의 공격 방식과, 이러한 CRA 가 가지는 독특한 branch 특성을 이용하여 공격을 탐지하는 Signature 기반 탐지 기술에 대하여 설명하도록 하겠다. 최근 들어, ROP 보다는 JOP 스타일의 공격이 대중화되고 있으므로, JOP 를 기준으로 설명하도록 하겠다.

아래 그림 1 은 기존의 코드들을 재활용하여, JOP 스타일의 CRA 가 수행될 때의 프로그램 실행 흐름이다. 공격자가 원하는 코드를 수행해 줄 gadget 들이 연결되어 있고, gadget 사이 사이에는 jump 할 주소를 업데이트 해주는 역할을 하는 trampoline gadget 들이 위치하고 있다. 일반적으로 CRA 는 자신이 원하는 code 를 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 작은 크기의 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 활용하여, indirect branch 가 일어날 때마다 사이사이 수행되는 명령어의 개수를 분석하여 CRA 를 탐지한다. 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 블락들이 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 볼 수 있듯이, 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 잇는 meta-data 방식을 사용하면, 별도의 계산 코드 없이 빠르게 gadget 의 크기를 얻어낼 수 있다

3. ARM Instruction Set Architecture (ISA)상의 Branch 명령어들

본 연구에서는 ARM 프로세서를 기반으로 한 모바일 기기를 가정하고 있으므로, meta-data 를 생성하기 위해서는 ARM 의 바이너리 코드를 분석해야 한다. 4 장에서 본격적인 설계 방식을 설명하기 이전에, 이 장에서는 ARM 의 다양한 branch 명령어들에 대하여 설명하도록 하겠다.

x86 시스템과 달리, ARM ISA 에서는 다양한 명령어들이 branch 의 용도로 사용될 수 있다. 일반적인 direct branch 의 경우 다른 프로세서들과 마찬가지로 다양한 condition code 가 붙은 BEQ, BNE 등과 같은 정해진 명령어가 사용된다. 하지만 indirect branch 의 경우, ARM 은 r15 즉 PC 에 직접 값을 넣는 방식으로도 control flow 를 바꿀 수 있기 때문에, 다양한 indirect branch 구현이 가능하다.

일반적으로 가장 많이 쓰이는 indirect branch 는 B, BL, BLX 와 같은 명령어와 r3 와 같은 일반 레지스터를 조합시키는 것이다. 이 경우 r3 에 있는 값을 주소로 하여, indirect branch 가 가능하다. 하지만 **MOV r15, r3** 와 같이 PC (r15)에 직접 값을 써서 jump 를 하는 것도 가능하다. 혹은, **LDR r15, [r1], #4** 와 같이 메모리에 있는 값을 읽어와 바로 쓰는 것이 가능하다. 대표적인 것은 context switch 를 수행할 때 LDM 과 같은 다중 load 명령어를 이용해, 스택에 있는 return 주소

를 PC 에 쓰는 방식이다. 우리는 이러한 ARM ISA 의 특성을 고려하여, 우리의 정적 분석 기술이 다양한 형태의 branch 를 모두 처리할 수 있도록 구현하였다.

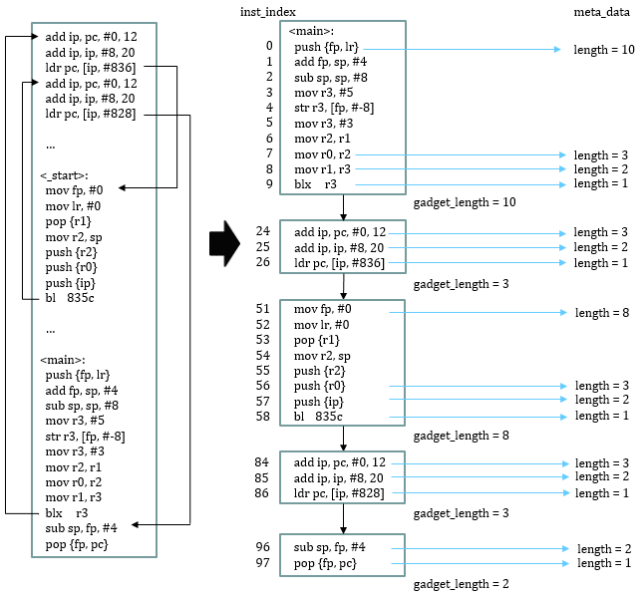
4. Meta-data 생성 및 CRA 탐지 구현

CRA 탐지에 필요한 meta-data, 즉 gadget 의 크기에 대한 정보는 runtime 시 수행될 명령어들의 주소별로 생성해주어야 하고, runtime 시 수행되는 명령어들은 바이너리 파일의 text section 에 있으므로, 정적 분석 시 바이너리 파일 전체를 분석할 필요없이 text section 만 분석하면 된다. **–objcopy –j .text –O elf32-littlearm** 옵션을 이용하여 text section 만을 담고 잇는 새로운 바이너리 파일을 생성하였고, 이 바이너리 파일에 대해 정적 분석을 시행하였다.

각 명령어에 대응되는 meta-data 를 계산할 때는 그 명령어와 가장 가까우면서 아직 수행되지 않은 branch 명령어가 기준이 되어, 그 명령어가 수행되고 나서부터 다음 branch 명령어가 올 때까지의 명령어 개수를 저장한다. 이 때, runtime 시 명령어가 실행될 순서대로 meta-data 를 계산하고 저장하는 것이 아니라, 명령어가 저장된 주소의 역순으로 가면서 저장하는 것이 편리하다. 즉, 역순으로 가면서 branch 명령어가 오면 1 을 저장하고, 그 바로 위의 명령어에 대해서는 2 를 저장하고, 그 위의 명령어에 대해서는 3 을 저장하는 식으로 또 다른 branch 명령어가 올 때까지 값을 1 씩 증가시키면서 저장하고, 또 다른 branch 명령어가 오면 다시 1 을 저장한다. 그림 2 는 이러한 meta-data 을 계산 및 저장하는 pseudo code 를 나타내고 있다. 이렇게 계산된 meta-data 는 바이너리 파일에 같이 저장된 뒤 runtime 시 사용된다. gadget 의 길이가 256 을 넘어가는 일이 없기 때문에, 각 meta-data 마다 unsigned char 자료형을 통해 1byte 를 할당하였다. 32-bit 명령어 하나당 1byte(=8bit)의 meta-data 가 생성되는 셈이다. 기존 바이너리 파일 대비 어느 정도 파일 크기가 증가하였는지를 실험한 5 장에서는 이 점을 이용하여 실험을 진행하였다.

```
struct meta_data {
   unsigned char gadget_length;
}
while(1) {
   meta_data *m = (struct meta_data*) malloc(sizeof(struct
                        meta_data) * total_number_of_inst);
   if (inst belongs to branch inst.) {
      length = 1;
   }
   else {
      length++;
   }
   m[inst_index].length = length;
   inst = prev_inst;
   if (inst == finish of program)
      break;
}
```

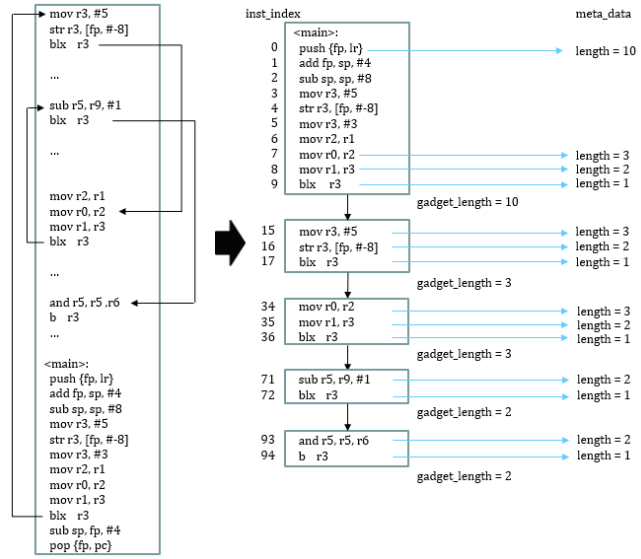
(그림 2) meta-data 를 계산 및 저장하는 pseudo code



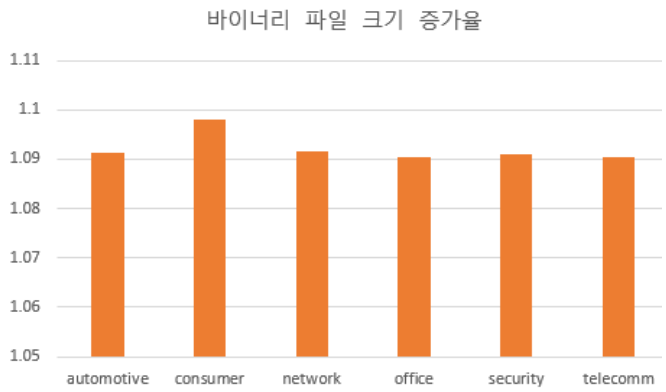
(그림 3) 정상적인 control flow 에서의 gadget 흐름

그림 3 과 그림 4 는 각각 정상적인 control flow 에서의 gadget 흐름과 CRA 가 발생하였을 때의 gadget 흐름의 예를 나타내고 있다. 그림 3 의 왼쪽 부분은 text section 만 담고 있는 바이너리 파일을 나타내며, 오른쪽 부분은 gadget 의 흐름과 저장되어 있는 meta-data 를 나타낸다. meta-data 는 바이너리 파일의 제일 밑인 pop {fp,pc}부터 제일 위의 add ip,pc, #0, 12 순으로 계산 및 저장되며, runtime 시 blx r3 을 통해 indirect jump 를 하면 그 다음 gadget 의 길이는 meta-data 를 통해 3 임을 바로 알 수 있게 된다. indirect branch 가 연속으로 오지 않고, 매우 짧은 길이의 gadget 들이 연속으로 오지 않으므로 정상적인 control flow 로 간주할 수 있다.

반면, 그림 4 에서는 indirect branch 가 연속으로 오며, 매우 짧은 길이의 gadget 이 연속되는 수행 흐름상의 특성을 보이고 있으므로, CRA 가 발생하였다고



(그림 4) CRA 가 발생하였을 때의 gadget 흐름



(그림 5) Meta-data 로 인한 파일 크기 오버헤드

판단할 수 있다.

5. 실험

본 논문에서 제안한 정적 분석을 위한 meta-data 생성 기술과 그와 관련한 실험은 Linux 3.8 kernel 환경하에서 이루어졌고, ARM 바이너리 생성을 위해서 Xilinx ARM cross compiler 를 사용하였다. 컴파일러 버전은 gcc version 4.6.3 이고, Linux 서버 사양은 Intel Core i5-2400 CPU @ 3.10GHz 이다.

그림 5 는 Mibench 에 있는 6 가지 종류의 벤치마크에 대해서 meta-data 생성으로 인한 바이너리 파일 크기 증가를 나타내고 있다. 가로축은 Mibench 에 있는 6 가지 종류의 벤치마크를 나타내고 있고, 세로축은 기존 바이너리 파일 크기 대비 Meta-data 생성으로 인한 바이너리 파일 크기의 증가율을 나타내고 있다. 평균적으로 약 9% 정도로 파일 크기가 증가한 것을 알 수 있다.

6. 결론

본 연구에서 제안한 정적 meta-data 생성 기술은 기존의 연구와 비교해 볼 때, 약 9% 정도의 파일 크기 증가가 있었지만 runtime 시의 상당한 성능 저하를 피할 수 있었다. 메모리 크기에 제약이 따르는 모바일 플랫폼의 특성상, 9%의 파일 크기 오버헤드는 아주 무시 못할 수준이긴 하지만, 성능 저하 없이 모바일 플랫폼에서도 CRA 를 효과적으로 막아낼 수 있다는 점이 이 연구의 의의이다. 또한, 모바일 플랫폼에 장착되는 메모리 역시 그 크기가 지속적으로 증가하고 있으므로, 바이너리 파일 크기 증가에 의한 오버헤드에서 좀 더 자유로워질 수 있다.

Acknowledgement

본 연구는 교육과학기술부/한국과학재단 우수연구센터 육성사업(과제번호 2012-0000470), 중소기업청에서 지원하는 2012 년도 산학연공동기술개발사업(No. C0019562), 미래창조과학부 및 한국산업기술평가관리원의 산업융합원천기술개발사업(정보통신) [No. 10047212, 1kB 이하 암호문 간의 연산을 지원하는 동형 암호 원천 기술 개발 및 응용 연구] 및 IDEC 의 지원을 받아 수행하였습니다.

참고문헌

[1] Suarez-Tangil, Guillermo, et al. "Evolution, detection and analysis of malware for smart devices." (2013): 1-27.
[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.