Code Reuse Attack 의 탐지를 위한 Meta-data 생성 및 압축 기술

황동일*, 허인구*, 이진용*, 이하윤*, 백윤흥*
*서울대학교 전기정보공학부 및 반도체공동연구소
e-mail : {dihwang, igheo, jylee, hyyi}@sor.snu.ac.kr, ypaek@snu.ac.kr

A Meta-data Generation and Compression Technique for Code Reuse Attack Detection

Dongil Hwang*, Ingoo Heo*, Jinyong Lee*, Hayoon Yi* and Yunheung Paek*
* Dept. of Electrical and Computer Engineering and Inter-University Semiconductor Research Center (ISRC), Seoul National University

요 약

근래 들어 모바일 기기의 시스템을 장악하여 사용자의 기밀 정보를 빼내는 악성 행위의 한 방법으로 Code Reuse Attack (CRA) 이 널리 사용되고 있다. 이와 같은 CRA 를 막기 위하여 call-return 이 일어날 때마다 이들 address 를 비교해 보는 shadow stack 과 branch 에 대한 몇 가지 규칙을 두어 CRA 를 탐지하는 branch regulation 과 같은 방식이 연구되었다. 우리는 shadow stack 과 branch regulation 을 종합하여 여러 종류의 CRA 를 적은 성능 오버헤드로 탐지할 수 있는 CRA Detection System 을 만들고자 한다. 이를 위하여 반드시 선행 되어야 할 연구인 바이너리 파일 분석과 meta-data 생성 및 압축 기술을 제안한다. 실험 결과 생성된 meta-data 는 압축 기술을 적용하기 전보다 1/2 에서 1/3 가량으로 그 크기가 줄어들었으며 CRA Detection System 의 탐지가 정상적으로 동작하는 것 또한 확인할 수 있었다.

1. 서론

근래 들어 스마트폰이나 태블릿 PC 와 같은 모바일 기기의 보급률이 폭발적으로 증가한 만큼 이들이 주로 사용하는 운영체제인 구글의 안드로이드 혹은 애플의 iOS 등을 대상으로 하는 다양한 보안 위협 역시 증가하는 추세이다. 백신 개발 회사인 Kaspersky Lab. 과 국제형사경찰기구 (INTERPOL)가 2014 년 공동 조사 및 발표한 report 에 따르면 [1], 2014 년 3 월에는 2013 년 8 월에 비하여 한달 간 모바일 공격 건수가 무려 10 배 가량 증가하였으며 이러한 공격의 59.06% 가 사용자에게 금전적 피해를 입힐 수 있는 것이었기 때문에 사태가 심각하다.

과거에 가장 흔한 공격법 중 하나는 code injection 이라 불리는 것으로 buffer overflow 를 사용하여 사용자 system 의 stack 에 공격 코드를 삽입한 뒤, function return address 를 삽입된 공격 코드의 시작점으로 조작하여 공격자가 원하는 악성 코드를 수행하는 것이었다 [2]. 하지만 이러한 공격 방식은 한 memory 영역을 쓰기 가능 (writable) 혹은 실행 가능 (executable) 중 한 가지 권한만 가질 수 있도록 함으로써 stack 과 같은 data 영역의 실행을 막는 방법으로 비교적 쉽게 무력화시킬 수 있었다 (W⊕X protection).

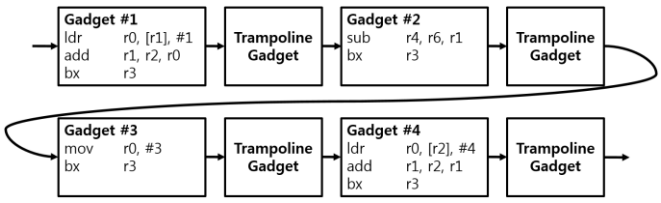
그래서 등장한 새로운 공격 방식이 code reuse attack (CRA)으로 오늘날 특히 위협적인 것으로 알려져 있다. CRA 는 크게 예전 방식인 return-oriented programming (ROP)와 이를 보완하기 위해 만들어진 jump-oriented programming (JOP)로 나눌 수 있다. ROP 와 JOP 모두 공통적으로 그림 1 과 같이 gadget 이라고 불리는 3~4 개 이하의 명령어로 이루어진 작은 코드 블록을 branch 와 jump 로 연결하여 원하는 악성 행위를 수행하는 방식으로 이루어진다. 이러한 방식의 공격은 기존의 시스템 위에 존재하는 실행 가능 영역의 코드를 재활용하기 때문에, 위에서 언급한 W⊕X protection 을 사용하여 data 영역의 실행을 막는 것으로는 방어가 불가능하여 매우 위협적이다.

이러한 모바일 기기 상에서의 CRA 를 방어하기 위한 방법 [3][4][5]이 여러 가지 측면에서 연구된 바 있다. 이 중 MoCFI[4]에서 도입된 shadow stack 은 프로그램이 실행되는 도중에 함수의 call 과 return 의 짝을 비교하여 control flow 를 감시하기 때문에 return instruction 만을 사용하여 gadget 을 연결하는 ROP 공격은 완벽하게 막을 수 있지만 indirect jump 를 사용한 JOP 스타일의 공격에는 쉽게 무력화되고 만다. 또한 shadow stack 의 비교를 위해 call 과 return 이 일어날 때 마다 추가적인 instrumented 코드를 수행하기 때문

에 성능 저하가 상당히 큰 편이다 (최대 4.87 배). 따라서 MoCFI 와 같은 접근 방식을 현재의 모바일 기기에 그대로 적용하기에는 어려움이 따른다.

Shadow stack 으로는 탐지할 수 없는 JOP 공격을 막기 위해 등장한 방법이 branch regulation (BR)이다 [5]. BR 은 call 할 시에는 항상 어떠한 function 의 entry 로만 이동해야 하며, return 명령어는 call 명령어가 실행될 때 저장된 주소로만 이동해야 하고 indirect jump 는 반드시 같은 function 내부의 주소 혹은 다른 function 의 entry 주소로만 이동할 수 있다는 세 가지 규칙을 포함한다.

본 논문에서는 앞서 언급한 shadow stack 과 branch regulation 기술을 종합하여 우리가 새롭게 고안해낸 CRA Detection System 을 간단히 소개하고, 이를 설계하기 위해 반드시 선행되어야 할 연구인 meta-data 생성 및 압축 기술을 제안한다. 실험 결과, 생성된 meta-data 를 이용하여 CRA Detection System 이 올바르게 CRA 를 감지하는 것을 확인하였으며 meta-data 의 압축 기술을 적용하여 그 크기를 대략 1/2-1/3 가량으로 줄일 수 있는 것으로 나타났다.



(그림 1) Code Reuse Attack 의 예시

2. Code Reuse Attack 과 CRA Detection System

우리는 ROP 공격을 방어할 수 있는 shadow stack 과 더 포괄적인 개념인 JOP 공격을 방어할 수 있는 branch regulation 을 통합하여 모든 종류의 CRA 를 탐지할 수 있으면서도 성능 오버헤드가 크지 않은 CRA Detection System 을 만들고자 하였다. 우리가 고안한 CRA detecting system 의 대략적인 구조는 그림 2 와 같이 나타낼 수 있다.

CRA Detection System 은 크게 프로그램이 실제 수행되는 runtime 에 관여하는 부분인 runtime detection process 와 이 과정에 앞서 반드시 필요한 meta-data 를 생성하기 위한 부분인 offline binary analysis 로 나눌 수 있다. Runtime detection process 는 Zynq-7000 보드 위에 실제로 구현한 하드웨어에서 수행되며 크게 branch time handler (BTH)와 shadow stack 그리고 indirect branch bounds checker (IBBC)로 이루어져 있다.

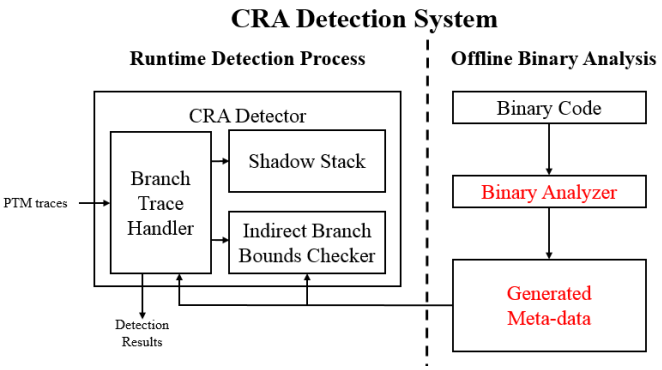
우리의 시스템에서 BTH 는 PTM 으로부터 프로그램 트레이스를 전달받아와 분석한 뒤 shadow stack 과 IBBC 으로 전달한다. PTM (Program Trace Macrocell) 이란 ARM® CoreSight 에서 생산되는 실시간 추적 모듈로써 ARM processor 에 포함되어있다. PTM 은 프로세서 위에서 돌아가고 있는 프로그램 instruction 의 여러 가지 정보들을 추적하여 다른 모듈에 제공해주는 역할을 한다.

CRA Detection System 에서 기본적으로 ROP 를 사용한 공격은 MoCFI 에서 제안한 shadow stack 과 같은 방식으로 감지한다. 다만 MoCFI 에서 바이너리 코드를 추가하여 return address 를 shadow stack 에 복사 및 저장한 것과는 달리 CRA Detection System 에서는 프로그램이 수행되는 동시에 하드웨어상으로 PTM 에서 받아온 정보를 BTH 를 통하여 분석하고 그를 바탕으로 return address 정보를 shadow stack 에 저장하기 때문에 이와 관련한 시간 오버헤드 문제를 해결할 수 있다.

Shadow stack 을 사용하면 branch regulation 의 세 가지 규칙 중 return 시에는 항상 call 명령어가 실행될 때 저장된 주소로만 이동해야 한다는 규칙이 위반되는 상황을 탐지할 수 있다. 그러므로 남은 두 가지 규칙, call 할 시에는 항상 다른 한 function 의 entry 로만 이동해야 한다는 규칙과 indirect jump 를 할 시에는 function 내에서 이동하거나 혹은 다른 function 의 entry 로만 이동할 수 있다는 규칙을 IBBC 에서 담당하여 탐지한다.

하지만 문제는 PTM 에서 제공하는 정보가 indirect branch bounds checking 을 하기에는 충분하지 않다는 점이다. PTM 은 indirect branch 명령어가 있을 때 target address 주소를 레지스터로부터 읽어와 전달해 주거나 그 branch 가 taken 되었는지 not taken 되었는지의 정보는 알려 줄 수는 있다. 하지만 IBBC 를 위해서는 바이너리 코드의 명령어들에 대해서 각 주소가 function entry 에 해당하는지 아닌지, 각 function 의 size 는 몇인지의 정보가 추가적으로 필요하다. 또한 프로그램의 flow 를 따라가는 과정에서 direct branch 가 taken 되었을 경우의 target address 정보와 해당 instruction 으로부터 가장 가까운 branch 의 source address 정보 역시 필요한 부분이다.

따라서 우리는 이 정보들을 runtime detection 을 하기에 앞서서 정적 바이너리 분석을 통해 meta-data 의 형태로 생성하여 메모리에 저장해둔다. 이 과정은 그림 2 에서 offline binary analysis 부분에 해당한다. 그리고 프로그램이 실제로 수행되는 runtime 에는 CRA detector 가 메모리에 저장해둔 meta-data 의 정보를 실시간으로 읽어오면서 indirect branch 의 bound checking 을 수행하게 된다.



(그림 2) CRA Detection System

3. ARM Instruction Set Architecture (ISA)의 Branch 명령어 종류

본격적으로 meta-data 생성 및 압축 기술에 대하여 설명하기 전에 branch 의 종류에 대하여 설명할 필요가 있다. 본 연구에서는 ARM 프로세서를 기반으로 한 모바일 기기를 감지 대상으로 가정하고 있으므로, meta-data 를 생성하기 위하여 ARM 의 바이너리 코드를 분석해야 한다. CRA Detection System 에서 원활한 CRA 탐지 및 분석을 하기 위해 우리는 ARM ISA 의 branch 들을 총 다섯 가지 타입으로 분류하였다. Direct branch, direct call, indirect branch, indirect call 그리고 return 이 이에 해당한다.

일반적으로 현재 address 와 target address 사이의 offset 을 계산하여 jump 할 때 사용하는 branch 명령어의 경우 direct branch 로 분류하고 어떠한 레지스터의 값을 target address 로 사용하여 jump 하는 branch 명령어의 경우를 indirect branch 라 한다.

그 중에서도 ARM processor 에서 return address 를 저장하는 데 사용하는 레지스터인 LR (link register)에 다음 명령어의 주소를 저장하고, direct branch 와 같은 방식으로 jump 하는 **BL <label>**과 **BLX <label>** 명령어를 direct call 로 분류한다. 마찬가지로 다음 명령어의 주소를 LR 에 저장하면서 indirect branch 와 같은 방식으로 jump 하는 **BLX Rm** 이 indirect call 에 해당된다.

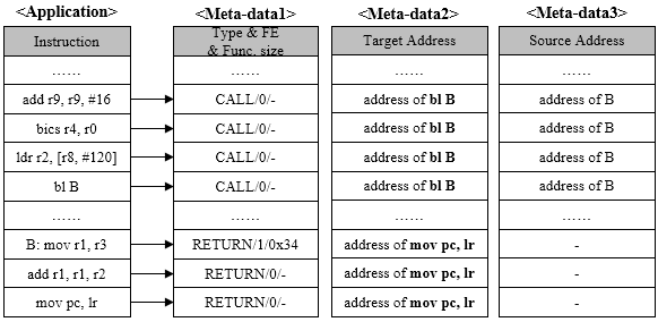
마지막으로 return type 의 branch 는 ARM ISA 에서 여러 가지 형태의 명령어로 표현될 수 있다. 먼저 **BX LR** 의 명령어를 쓰면 call 할 때 저장했던 주소로 되돌아가므로 return 을 나타낸다. **SUBS PC, LR** 이나 **MOVS PC, LR** 과 같이 PC 에 LR 에 저장되어 있는 address 를 직접 넣어주는 명령어로 return 을 표현할 수도 있다. 그리고 stack 에 push 되었던 LR 의 값을 PC 로 pop 하는 것도 return 의 한 방법이다.

4. Meta-data 의 생성 및 최적화

Meta-data 는 2 장에서 언급한 대로 branch 의 type, source address 와 target address 외에도 function entry 인지의 여부, function entry 일 경우 그 function 의 크기에 대한 정보도 포함한다. 하지만 만약 이와 같은 정보를 meta-data 에 담기 위하여 text section 의 각 명령어 주소에 일대일로 대응되도록 entry 들을 만들어 그림 3 과 같이 meta-data 들을 생성한다면, meta-data 파일이 최소 세 개가 필요하여 대상 바이너리 코드에 비하여 meta-data 의 크기가 과도하게 커지게 된다.

또한 target address 들은 정적 바이너리 분석으로 이를 구할 수 있는 direct branch 및 direct call 에서만 그 값을 meta-data 에 쓸 수 있다. 따라서 target address 를 정적으로 구할 수 없는 indirect branch, indirect call 및 return 의 경우에는 meta-data 의 address 를 맞추기 위하여 해당 entry 들을 0 으로 채워 넣어야 하므로 필요 없는 공간 낭비가 심한 편이다.

우리는 한 branch 명령어가 등장한 후, 또 다른 branch 명령어가 등장하기 전까지 그 사이에 존재하는 instruction 의 meta-data 들은 function entry 에 대한



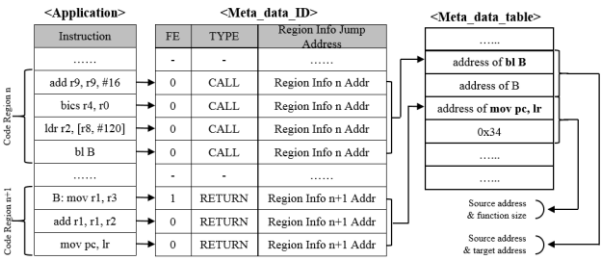
(그림 3) Meta-data 생성 예시

정보를 제외하고는 모두 같다는 점에서 착안하여 meta-data 의 크기를 압축하는 방법을 제안한다

새로운 방식으로 생성된 meta-data 는 그림 4 와 같이 meta_data_id 와 meta_data_table 의 두 개의 파일로 나뉘어 만들어진다. Meta_data_id 의 한 entry (32-bit)에는 대응되는 명령어가 function entry 인지의 여부를 기록하는 FE (1-bit) 와 가장 가까운 branch 의 type (3-bit)이 직접 기록되며 해당 branch 의 추가 정보가 담긴 meta_data_table 의 올바른 위치로 jump 할 수 있도록 그 주소가 들어가게 된다 (28-bit). Meta_data_table 의 해당 위치에는 기본적으로 source address 가 들어가며, branch type 이 direct branch 혹은 direct call 의 경우 target address 가, 그 branch 명령어가 function entry 이래로 처음 등장하는 branch 일 경우 해당 function 의 size 정보가 추가적으로 담긴다.

이와 같은 meta-data 를 생성하기 위하여 제일 먼저 ARM 바이너리 파일을 분석하여 text section 의 offset 을 찾아낸다. 그 뒤 심볼 테이블에 담겨있는 정보를 바탕으로 text section 의 각 부분이 ARM 혹은 THUMB 중 어떤 모드의 명령어로 되어 있는지를 분석하여 기록한다. 그리고 text section 의 명령어들을 모드에 맞게 순서대로 disassemble 하여 target address, source address 와 같은 정보들을 한 class 에 담아 queue 에 push 한다. 그 뒤 한번 더 text section 을 읽어 내리면서 각 명령어의 주소를 queue 의 제일 앞에 있는 branch 의 주소와 비교하면서 같은 주소가 나올 때까지 그 branch 의 type 과 jump address 등을 meta_data_ID 에 쓴다. 그리고 다시 순서대로 queue 의 branch 정보들을 pop 하면서 meta_data_table 에 순차적으로 쓴다.

이와 같은 방식으로 meta-data 를 생성할 경우 두 개의 파일에 IBBC 를 위해 필요한 모든 정보를 표현할 수 있다. 생성된 meta-data 는 runtime detection



(그림 4) 최적화된 Meta-data 생성 예시

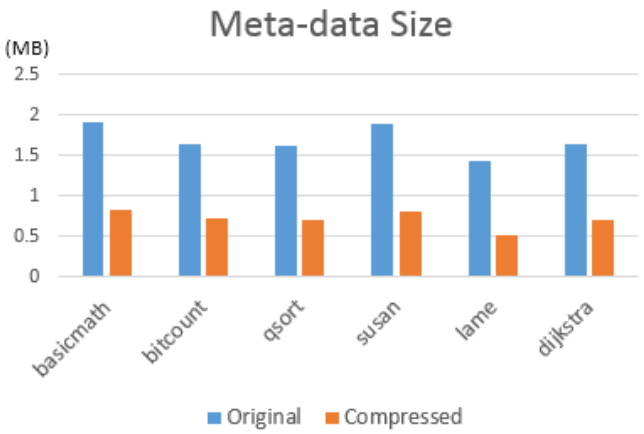
process 중 현재의 명령어 주소에 해당하는 entry 를 메모리에 저장된 meta_data_id 에서 확인하여 jump address 를 읽어온 뒤 다시 메모리의 meta_data_table 의 올바른 위치에 접근해서 원하는 일련의 정보를 받아오는 방식으로 사용된다.

5. 실험

본 논문에서 제안한 CRA 탐지를 위한 바이너리 분석과 meta-data 생성 및 압축 기술에 대한 검증은 MiBench Version 1.0 을 벤치마크로 사용하여 진행되었다. MiBench 의 소스 코드를 arm-linux-gnueabihf-gcc 4.7.3 버전으로 컴파일 하였으며 컴파일 시에 –static option 을 추가하여 library 코드 영역을 바이너리 파일에 포함시킴으로써 정적 분석이 용이하도록 하였다.

그림 5 는 Mibench 의 6 가지 종류의 벤치마크들에 대하여 meta-data 를 생성해 보고, 앞서 제시한 압축 기술을 적용 했을 때와 적용하지 않았을 때의 meta-data 크기를 비교하여 나타낸 것이다. 압축 기술을 적용하였을 경우 그렇지 않았을 때에 비하여 meta-data 의 크기가 작게는 1/2 에서 크게는 1/3 가량 압축되었다는 것을 알 수 있다.

또한 생성된 meta-data 를 시스템의 메모리에 넣어서 CRA Detection System 에서 사용할 수 있도록 한 뒤, CRA 가 발생한 상황을 가정하여 직접 시뮬레이션 해본 결과 meta-data 가 올바르게 생성되었으며 CRA 의 탐지가 정상적으로 이루어지는 것을 확인할 수 있었다.



(그림 5) 생성된 Meta-data 의 크기 비교

6. 결론 및 향후 과제

본 연구에서 제안한 정적 meta-data 생성 기술을 사용하면 CRA Detection System 으로 모바일 기기에 대한 ROP 및 JOP 공격에 대한 효과적인 CRA 탐지가 가능해지기 때문에 의의가 있다. 또한 meta-data 의 크기를 압축함으로써 시스템에서의 메모리 사용률을 줄이고 이로 인한 오버헤드를 줄일 수 있었다.

본 연구에 이어서 향후에 이루어져야 할 과제로 본 연구의 의의를 좀 더 명확히 하기 위해 meta-data 의 크기를 좀 더 압축하는 기술 연구가 요구된다. 또한 CRA Detection System 에서 메모리에 반복적으로 접근하기 때문에 발생하는 성능 오버헤드가 존재하기에, 이를 방지하고자 캐시를 중간에 달아서 메모리 접근에 걸리는 시간을 감소시키는 방법을 고안 중에 있다.

Acknowledgement

본 연구는 미래창조과학부 및 한국산업기술평가관리원의 산업융합원천기술개발사업(정보통신) [No.10047212, 1kB 이하 암호문 간의 연산을 지원하는 동형 암호 원천기술 개발 및 응용 연구]과 2015 년도 두뇌한국 21 플러스사업, 중소기업청에서 지원하는 2014 년도 산학연협력기술개발사업(No. C0218072), IDEC 의 지원 및 2014 년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원(No. 2014R1A2A1A10051792)을 받아 수행된 연구임을 밝힙니다.

참고문헌

[1] Kaspersky Lab. and INTERPOL “Mobile Cyber Threats.” (2014)

[2] Pincus, Baker, et al. “Beyond stack smashing: Recent advances in exploiting buffer overruns.” IEEE Security and Privacy. 2004.

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

[4] Chiueh, Hsu, et al. “RAD: A compile-time solution to buffer overflow attacks.” International Conference on Distributed Computing Systems (ICDCS). 2001.

[5] Kayaalp, Ozsoy, et al. “Branch Regulation: Low-Overhead Protection from Code Reuse Attacks.” International Symposium on Computer Architecture (ISCA). 2012.