Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.32, NO.2, Apr. 2022 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2022.32.2.171

자체 수정 코드를 탐지하는 정적 분석 방법의 LLVM 프레임워크 기반 구현 및 실험*

유 재 일,[1†] 최 광 훈[2‡]
[1,2]전남대학교 (대학원생, 교수)

An LLVM-Based Implementation of Static Analysis for Detecting Self-Modifying Code and Its Evaluation*

Jae-IL Yu,[1†] Kwang-hoon Choi[2‡]
[1,2]Chonnam National University(Graduate student, Professor)

요 약

자체 수정 코드(Self-Modifying-Code)란 실행 시간 동안 스스로 실행 코드를 변경하는 코드를 말한다. 이런 기법은 특히 악성코드가 정적 분석을 우회하는 데 악용된다. 따라서 이러한 악성코드를 효과적으로 검출하려면 자체 수정 코드를 파악하는 것이 중요하다. 그동안 동적 분석 방법으로 자체 수정 코드를 분석해왔으나 이는 시간과 비용이 많이 든다. 만약 정적 분석으로 자체 수정 코드를 검출할 수 있다면 악성코드 분석에 큰 도움이 될 것이다.

본 논문에서는 LLVM IR로 변환한 바이너리 실행 프로그램을 대상으로 자체 수정 코드를 탐지하는 정적 분석 방법을 제안하고, 자체 수정 코드 벤치마크를 만들어 이 방법을 적용했다. 본 논문의 실험 결과 벤치마크 프로그램을 컴파일로 변환한 최적화된 형태의 LLVM IR 프로그램에 대해서는 설계한 정적 분석 방법이 효과적이었다. 하지만 바이너리를 리프팅 변환한 비정형화된 LLVM IR 프로그램에 대해서는 자체 수정 코드를 검출하기 어려운 한계가 있었다. 이를 극복하기 위해 바이너리를 리프팅 하는 효과적인 방법이 필요하다.

ABSTRACT

Self-Modifying-Code is a code that changes the code by itself during execution time. This technique is particularly abused by malicious code to bypass static analysis. Therefor, in order to effectively detect such malicious codes, it is important to identify self-modifying-codes. In the meantime, Self-modify-codes have been analyzed using dynamic analysis methods, but this is time-consuming and costly. If static analysis can detect self-modifying-code it will be of great help to malicious code analysis.

In this paper, we propose a static analysis method to detect self-modified code for binary executable programs converted to LLVM IR and apply this method by making a self-modifying-code benchmark. As a result of the experiment in this paper, the designed static analysis method was effective for the standardized LLVM IR program that was compiled and converted to the benchmark program. However, there was a limitation in that it was difficult to detect the self-modifying-code for the unstructured LLVM IR program in which the binary was lifted and transformed. To overcome this, we need an effective way to lift the binary code.

Keywords: Self-Modifying-Code, Static Analysis, Benchmarking

Received(02. 03. 2022), Modified(03. 02. 2022), Accepted(03. 07. 2022)

* 이 논문은 ETRI 부설연구소의 위탁연구과제[2020-017]로 수행한 연구 결과입니다.

* 이 논문은 2019년도 정부 교육부의 재원으로 한국연구재단의 지원을 받아 수행된 연구입니다(No.2019R1I1A3A01058608).

* 이 논문은 2022년도 정부(과학기술정보통신부)의 재원으로 정보통신기획평가원의 지원을 받아 수행된 연구임(IITP-2019-0-01343)

† 주저자, dbwodlf3@naver.com

‡ 교신저자, kwanghoon.choi@jnu.ac.kr(Corresponding author)

I. 서 론

오늘날 많은 사용자는 프로그램 코드로부터 바이너리 파일을 직접 만들어 사용하는 대신 사전에 컴파일된 실행파일을 사용하는 것이 보편화되어 있다. 이는 분명히 편리한 방식의 프로그램 배포이지만 동시에 해당 프로그램을 불투명하게 만든다. 이러한 프로그램은 소스 코드가 드러나지 않기 때문에 해당 프로그램에 악성 행위가 잠재되어 있는지 파악하기 위해서는 바이너리 레벨의 코드 분석이 필요하다.

바이너리 코드를 분석하는 방법으로 크게 동적 분석과 정적 분석으로 나뉜다. 동적 분석은 실제로 프로그램의 실행 흐름을 따라 진행되기 때문에 악성코드를 검출하는 데 있어서 매우 효과적이다. 정적 분석은 프로그램을 실행하지 않고 사전에 정의된 규칙에 따라 전체 코드를 검사한다. 정적 분석은 동적 분석과는 달리 코드를 직접 실행하지 않기 때문에 그 비용이 상대적으로 낮고 전체 코드에 대해서 검사하기 때문에 일괄적으로 적용하는데 편리하다. 따라서 일반적으로 악성 프로그램의 검출과 방지는 악성코드 분석관에 의해 특정 악성코드가 분석되고 이를 정적 분석 검출기에 등록하여 일반 사용자에게 배포해 효율적으로 악성코드를 검출한다.

하지만 위와 같은 과정에서 자체 수정 코드로 작성된 악성코드는 실행 시간에만 악성코드를 드러내 악성코드 분석관의 분석을 어렵게 한다. 이뿐만 아니라 이미 분석된 악성코드라 할지라도 다양한 형태로 복호화 되어 자체 수정 코드로 나타날 수 있기에 자체 수정 코드를 고려하지 않은 정적 분석기로는 이를 검출할 수 없다.

그러므로 이러한 우회 방법을 이용하는 악성코드를 정적 분석으로 검출하기 위해서는, 악성코드를 분석하기 이전에 자체 수정 코드에 대한 정적 분석이 선행되어야 한다.

본 논문이 기여한 바는 다음과 같다. 첫째, 자체 수정 코드 SMC 벤치마크를 만들어 분석기의 실험 환경을 구축하였다. 둘째, 전통적인 포인터 분석을 활용한 자체 수정 코드 정적 분석기를 설계하고 LLVM 프레임워크에서 구현하였다. 셋째, 이 정적 분석 방법을 SMC 벤치마크에 적용한 실험 결과를 리포트하고 장단점과 한계를 극복하는 방법을 논의했다.

II. 배경지식

바이너리 코드를 기계어 단위에서 바로 분석하는 데에는 어려움이 따른다. 따라서 바이너리 코드를 분석하기 위해서 리프팅을 하게 되는데 본 논문에서는 LLVM IR로 리프팅을 하기 위해서 오픈소스 프로그램인 McSema[1]을 사용한다.

2.1 LLVM

LLVM은 Low-Level Virtual Machine의 약어로서 모듈화된 LLVM 컴파일러 인프라스트럭처 전체를 가리키는 말이다. Fig 1과 같이 LLVM은 크게 언어를 정의하는 프론트엔드와 바이너리 코드를 생성하는 백엔드로 이루어진다. 프론트엔드를 통해서 LLVM IR을 생성한다. 생성된 LLVM IR을 최적화한 후 백엔드에서 바이너리 코드를 생성한다.

Fig 2와 같이 LLVM IR의 코드 구조는 파일 단위인 Module, 데이터 타입을 가지는 글로벌 변수, 타입을 가지는 지역 변수, 반환 타입을 가지는 함수, 베이지 블록으로 구성되어 있으며 65여 가지의 명령어와 나머지 플랫폼에 맞는 각각의 Intrinsic Function으로 이루어져 있다.

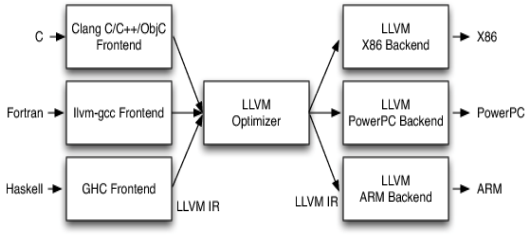

Fig. 1. LLVM Structure[2].

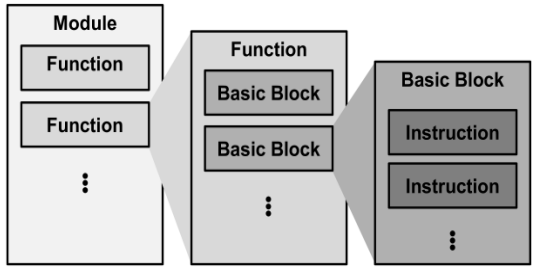

Fig. 2. LLVM IR Structure[3].

2.2 McSema

McSema는 윈도우의 PE(Portable Executable)와 리눅스의 ELF(Executable and Linkable Format) 같은 바이너리 실행파일을 LLVM IR로 리프팅 해주는 오픈소스 프로그램이다.

Fig 3에서와 같이 McSema는 Binary 파일로부터 IDA Pro와 같은 디스 어셈블러 프로그램을 통해서 CFG 파일을 생성한다. 내부의 리프팅 라이브러리인 remill[1]을 사용하여 CFG 파일로부터 LLVM IR 파일을 만들어낸다.

McSema가 만들어내는 CFG 파일은 단순히 1:1 디스 어셈블링 하는 것이 아닌, LLVM IR 레벨에서의 가상화 Runtime을 고려하여 만들어진다. 실제 하드웨어의 작동을 모사하기 때문에 이를 LLVM IR 인터프리터를 통해서 실행할 수 있고, 재컴파일하여 바이너리 파일로 만들어 실행할 수도 있다.

예를 들어서 만약 레지스터 A에 값 3을 저장하는 명령어가 있다면, McSema는 레지스터 A라는 타입을 가진 구조체가 있고 LLVM IR 상에서 해당 타입의 글로벌 변수 A를 만들어 해당 변수에 3을 저장한다.

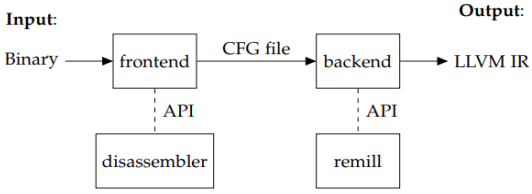

Fig. 3. Lifting Flow Chart of McSema[1].

2.3 자체 수정 코드(Self Modifying Code)

자체 수정 코드(SMC, Self-Modifying Code)란 프로그램이 실행되면서 스스로 실행 코드를 변경하는 코드를 말한다. Fig 4은 자체 수정 코드의 예시로, foo_addr = foo 대입문을 통해 코드 영역(Code Segment)에 있는 foo 함수의 주소를 가져온다. 이후에 memcpy 함수를 호출하여 해당 주소에 x86으로 작성된 셸코드(shellcode)를 기록 후 foo 함수를 호출한다. 이 때 foo의 메모리 주소는 코드 영역이므로 writable하게 변경하기 위해 시스템 호출을 사용했다.

```
void foo(void);

int main(void)
{
    // ....
    void *foo_addr = (void *)foo;
    mprotect(foo_addr, page_size, PROT_READ | PROT_WRITE | PROT_EXEC
    );
    // ....
    char shellcode[] =  "\x48\x31\xd2"
                        "\x48\x31\xc0"
                        //...
    memcpy(foo_addr,shellcode, sizeof(shellcode) -1);
    foo();
}
```

Fig. 4. Self-Modifyting Code Example

III. 자체 수정 코드 정적 분석

자체 수정 코드는 실행 가능한 메모리 영역의 데이터를 수정하는 것에서부터 시작한다. 따라서 실행 가능한 메모리 영역의 값이 변경되었다면 해당 코드는 자체 수정 코드로 판단할 수 있다.

이러한 관찰에 따라 설계된 자체 수정 코드 정적 분석 방법의 기본 원리는 다음과 같다.

1. *메모리 쓰기의 명령어는 포인터 변수를 피연산자로 사용한다.*
2. *포인터 변수가 실행 중에 가리킬 수 있는 모든 메모리 영역의 집합을 구한다.*
3. *피연산자로 사용된 포인터 변수의 집합에 함수가 포함되어 있다면 분석 대상 프로그램은 자체 수정 코드이다.*

본 연구에서는 이 아이디어를 바탕으로 포인터 분석을 수행하여 정적 분석기에 활용한다.

Fig.4의 예시에 포인터 정적 분석을 적용하면 foo_addr = foo 대입문으로부터 포인터 변수 foo_addr가 실행 중에 가리키는 메모리 영역 중 하나로 함수 foo가 포함됨을 파악할 수 있다. 이 분석 결과를 통해 이 예시는 자체 수정 코드로 판단한다.

3.1 포인터 정적 분석

앤더슨 알고리즘(Andersen's Algorithm)[4]

은 포인터 분석 알고리즘 중 하나이다. 실행 흐름과 문맥에 민감하지 않은 방식의 정적 분석으로 포인터의 연산에 대해 제약식을 세우고 제약식의 관계에 기반해 그 해를 구한다.

앤더슨 알고리즘에서 나타나는 관계식은 집합의 포함 관계, 원소의 포함 관계, 조건 관계식의 크게 3가지이고 해당 관계식을 통해 표현되는 C 언어의 제약식은 Table 1과 같다. 〔{ }〕 기호는 집합을 나타내는 기호다.

Table 2는 LLVM IR 내부에 포인터와 관련된 명령어의 Syntax으로 앤더슨 알고리즘에 맞게끔 생성한 LLVM IR의 제약식은 다음의 Table 3과 같다.

Table 1. Constraints of Andersen's Algorithm〔4〕.

Statement	Constraint	Type
p = &x	〔{p}〕⊇x	1
p = x	〔{p}〕⊇〔{x}〕	2
p = *x	∀v∈〔{x}〕 → 〔{p}〕⊇〔{v}〕	3
*p = x	∀v∈〔{p}〕 → 〔{v}〕⊇〔{x}〕	4

Table 2. LLVM IR Instruction Syntax.

Instruction	Syntax
alloca	〈result〉 = alloca 〈type〉
getelementptr	〈result〉 = getelementptr 〈ty〉, 〈ty〉* 〈ptrval〉 {, 〈ty〉 idx}*
inttoptr	〈result〉 = inttoptr 〈ty〉 〈value〉 to 〈ty2〉
bitcast	〈result〉 = bitcast 〈ty〉 〈value〉 to 〈ty2〉
phi	〈result〉 = phi 〈ty〉 〔 〈val0〉, 〈label0〉〕, ...
select	〈result〉 = select selty 〈cond〉, 〈ty〉 〈val1〉, 〈ty〉 〈val2〉
extractvalue	〈result〉 = extractvalue 〈aggregate type〉 〈val〉, 〈idx〉{, 〈idx〉}*
store	store 〔volatile〕 〈ty〉 〈value〉, 〈ty〉* 〈pointer〉
load	〈result〉 = load 〔volatile〕 〈ty〉, 〈ty〉* 〈pointer〉
call	〈result〉 = call 〈ty〉 〈fnty〉 〈fnptrval〉(〈function args〉)

Table 3. LLVM IR Constraints for Pointer Analysis.

Instruction	Constraint	Type
alloca	result ∈ 〔{result}〕	1
getelementptr	ptrval ∈ 〔{result}〕	1
inttoptr	〔{value}〕 ⊆ 〔{result}〕	2
bitcast	〔{value}〕 ⊆ 〔{result}	2
phi	〔{val1}〕 ⊆ 〔{result}〕, 〔{val2}〕 ⊆ 〔{result}〕, ...	2
select	〔{val1}〕 ⊆ 〔{result}〕, 〔{val2}〕 ⊆ 〔{result}〕	2
extractvalue	〔{val}〕 ⊆ 〔{result}〕	2
store	c ∈ 〔{pointer}〕 → 〔{value}〕 ⊆ 〔{c}〕	3
load	c ∈ 〔{pointer}〕 → 〔{c}〕 ⊆ 〔{result}〕	4
call	p1, p2, ..., v1, v2 ..., ret ret_var ∧ 〔{v1}〕⊆〔{p1}〕,〔{v2}〕⊆〔{p2}〕,... .∧ 〔{ret_var}〕⊆〔{result}〕	2

제약식 생성기는 Table 3의 제약 규칙을 기반으로, 리프팅 된 LLVM IR 명령어를 순회하여 제약식을 생성한다. 만들어진 제약식은 큐빅 알고리즘(Cubic Algorithm)〔5〕을 통해 그 해를 구한다.

포인터 변수 집합 P와 실제 값인 토큰 집합 T을 DAG 형태로 변환하여 연산이 이루어지는데 각 노드는 집합 V의 원소가 각 노드의 포함 관계는 튜플의 리스트를 통해서 표현된다.

노드마다 집합 T를 나타내는 비트 벡터가 있으며, 비트의 조건에 따른 노드의 전파 관계를 나타내기 위해서 각 비트에는 포함 관계를 나타내는 튜플의 리스트가 있다.

제약식 t∈〔{v}〕 은 노드 v의 비트 벡터의 t의 값을 1로 바꾸는 것과 같다. 제약식 〔{v1}〕⊆〔{v2}〕 은 Edge(v1, v2)으로 v1의 비트 벡터를 v2의 비트 벡터에 OR 연산을 해서 v2 노드에 저장 하는 것과 같다. ∀v∈〔{p}〕 의 조건문을 통한 제약식은 해당 비트 벡터의 튜플 리스트를 통해서 해당 비트 벡터 t의 값이 1인 경우에 Edge 을 구성하는 것과 같다.

Edge와 비트 벡터 그리고 전파 관계 리스트를 통해서 더 이상 DAG 그래프의 전파가 일어나지 않으면 포인터 분석은 종료된다.

이렇게 구해진 포인터 변수의 결과값은 다음 모듈에서 입력으로 사용된다.

3.2 포인터 분석 결과 기반 자체 수정 코드 검출

포인터 분석의 결과값이 주어지면 LLVM IR 명령어 중에 쓰기 명령어를 찾아 자체 수정 코드의 가능성이 있는지 확인한다. 쓰기 명령어의 피연산자로 있는 포인터 변수의 집합에 코드 영역이 포함되는 경우 해당 LLVM IR 프로그램을 자체 수정 코드로 판별한다.

자체 수정 코드 정적 분석기의 전체적인 구성은 Fig. 5와 같다.

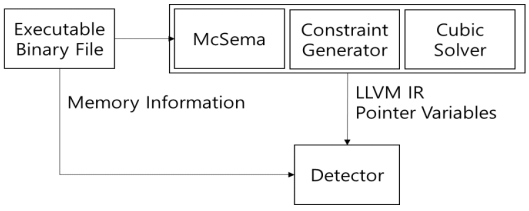

Fig. 5. Static Analyzer Structure.

IV. 자체 수정 코드 벤치마크 및 실험

검출기와 벤치마크의 소스코드는 https://github.com/dbwodlf3/SMC 깃허브 저장소에서 확인할 수 있고, 각 벤치마크에서 어떤 패턴으로 스스로의 코드를 변경하는지에 대한 설명은 이 연구결과의 보고서[8]에서 자세하게 확인할 수 있다.

Table 4는 자체 수정 코드 정적 분석기의 테스트를 위해서 "Certified self-modifying code" 논문[6]에 소개된 MIPS 코드를 9가지로 분류한 것으로 X86 코드와 C언어로 재작성하였다.

MIPS 코드는 RISC 구조로 모든 명령어의 길이가 4byte로 일정하지만, X86 코드는 CISC 구조로 명령어의 길이가 일정하지 않아 MIPS의 경우보다 더 복잡하다. C 언어로 작성한 벤치마크의 경우 컴파일러의 Position Independent Code 및 최적화 옵션 등 코드를 어떻게 생성할지에 관해서 의존성이 있어 벤치마크 코드는 x86 기반의 리눅스에서 동작하도록 고려하여 재작성 되었다.

Table 4. SMC Benchmark Code[6].

SMC1	unbounded code rewriting, Fibonachi
SMC2	runtime code checking
SMC3	runtime code generation
SMC4	multilevel runtime code generation
SMC5	self-mutating code block
SMC6	mutual modifying blocks
SMC7	self-growing code
SMC8	polymorphic code
SMC9	encryption/compression

모든 실험은 Ubuntu 18.04, NASM 2.13, gcc 7.5, Clang 9.0.0-2, McSema 3.0 버전과 IDA Pro 7.1 버전에서 이루어졌다.

4.1 X86과 C 기반 SMC 벤치마크

자체 수정 코드에 관한 기존 연구는 다수 존재하지만 여러 연구를 비교할 수 있는 자체 수정 벤치마크 프로그램은 제안된 바가 없다. 본 연구에서는 바이너리를 리프팅 하는 도구, 정적 분석기의 완성도와 최대한 독립적으로 SMC 벤치마크를 구성하였다.

각 SMC 코드는 단순한 형태로 한 가지 이상의 자체 수정 코드의 특징을 가지고 있다. SMC1은 가장 간단한 형태의 자체 수정 코드로 더하기 명령어의 오퍼랜드를 수정한다. SMC2는 SMC1과 같이 직접적인 쓰기 작업을 하지는 않지만, 어떻게 명령어 코드를 읽어 들이는지를 보여준다. SMC3는 코드가 실행되는 중에 벡터의 내적을 구하는 코드를 생성한다. SMC4는 생성된 코드가 또 다른 코드를 생성하는 복잡한 예를, SMC5는 실행 흐름에 관여하는 명령어를 수정하여 실행 흐름 자체를 변화시키고 다시 본래의 명령어로 되돌려 마치 정상적인 실행 흐름을 가지는 것처럼 속인다. SMC6은 코드와 코드가 서로를 변조하는 복잡한 양상을, SMC7은 루프를 돌면서 특정 영역의 명령어를 복사한다. SMC8은 명령어를 수정하나 그 프로그램의 동작은 같음을 보인다. SMC9는 XOR 연산을 이용해서 복호화된 코드를 메모리에 생성 후에 실행한다. 복잡한 방식의 SMC 또한 위의 각 특징들을 내포하고 있는 SMC의 조합으로 표현될 수 있다.

복합적인 자체 수정 코드는 SMC5와 마찬가지로 실행 흐름을 변경시키고 SMC9와 같이 암호화된 코드를 복원하고 SMC7과 같이 루프문을 이용해서 코드를 생성한다. 이렇게 우회된 코드는 자체 수정 코드 분석 없이는 악성코드를 검출할 수 없으며 정확히 어떻게 악성코드가 실행되는지 숨겨 바이너리 분석가의 분석을 어렵게 만든다.

4.2 C 벤치마크 대상 정적 분석기 성능 실험

Table 5에서 Clang LLVM IR 은 Clang을 통해서 만들어진 정형화된 LLVM IR 벤치마크다. 해당 벤치마크에 본 논문에서 설계한 분석기를 적용한 결과 직접적인 자체 수정 코드가 없는 SMC2를 제외한 다른 SMC 모두에서 자체 수정 코드를 검출할 수 있었다.

Table 6에서 소요된 시간은 LLVM IR 코드로부터 Constraint Generator, Cubic Solver와 Detector을 모두 거쳐 결과를 얻기까지의 시간이다.

검출 과정을 SMC1의 사례로 살펴보면 다음과 같다. SMC1은 피보나치수열의 값을 구하는 자체 수정 코드인데 수열의 값을 구할 때 더하기 명령 코드의 피연산자 값을 직접 수정하는 방법을 사용한다. 그 코드의 흐름은 다음과 같다. 첫째로 더하기 명령어를 코드 영역에 있는 메모리로부터 읽는다. 둘째로 해당 명령어의 오퍼랜드 값을 수정한다. 셋째로 더하기 명령어를 수행한다. 이 과정을 구하고자 하는 피보나치 순열의 번호만큼 반복한다.

Table 5. Static Analyzer Result From Benchmark.

	ANSWER	Clang LLVM IR
SMC1	O	O
SMC2	X	X
SMC3	O	O
SMC4	O	O
SMC5	O	O
SMC6	O	O
SMC7	O	O
SMC8	O	O
SMC9	O	O

Table 6. Static Analyzer Performance.

	Clang LLVM IR (line)	Clang LLVM IR (ms)
SMC1	93	202
SMC2	178	748
SMC3	202	997
SMC4	136	456
SMC5	214	1,079
SMC6	179	716
SMC7	80	284
SMC8	232	1,976
SMC9	112	306

검출기에 의해 최종적으로 검출되는 LLVM IR 명령어는 @llvm.memcpy(%34, %35, 4)으로 더하기 피연산자가 수정된 명령어인 %35을, 더하기 명령을 수행하는 주소를 가리키고 있는 %34에 쓴다.

%34 = load %2 명령어에 의해서 $[[\%2]] \subseteq [[\%34]]$ 의 제약식이 적용된다. %2는 @main의 명령어 주소를 가져오는 store(getelementptr (main, 107), %2)에 의해서 $@main \in [[\%2]]$ 의 제약식이 적용된다.

따라서 결과적으로 %34 변수에 @main 토큰이 포함된다. 그리고 @main 은 실행 권한을 지닌 코드 영역의 메모리이므로 이는 곧 자체 수정 코드이다.

다른 SMC에서도 SMC1과 마찬가지의 과정을 통해 자체 수정 코드가 검출되었다. 특히 SMC5에서 분기문의 명령어 코드를 수정하는 자체 수정 코드, SMC7에서 명령어를 실행 영역의 메모리에 작성하는 자체 수정 코드, SMC9에서 복호화된 명령어를 실행 영역에 쓰는 자체 수정 코드가 검출되었다.

이번에는 최종 바이너리를 리프팅 변환하여 얻은 LLVM IR 프로그램에 대해 실험하였다. 즉, C로 작성된 SMC 벤치마크 프로그램을 Clang 컴파일러를 통해서 실행 바이너리 파일로 만들고 이 파일을 McSema로 리프팅 하여 LLVM IR 프로그램을 만든 다음 정적 분석기를 적용하였다. 적용 결과 정적 분석기의 거짓 양성(false positive) 결과를 유도할

수 있는 리프팅 변환 코드들이 있었다.

예를 들어, SMC2는 코드가 수정되었는지 확인만 하고 직접적인 메모리의 코드를 수정하지 않으므로 검출기에 의해서 자체 수정 코드가 검출되어서는 안 되는 벤치마크 프로그램인데 McSema에 의해서 추가로 생성되는 코드로 인해 자체 수정 코드 명령어가 있다고 거짓 양성(false positive) 결과를 냈다.

McSema의 구조적 특성에 의해서 만들어지는 코드 때문에 다른 모든 C 기반 SMC 코드에서도 실제 자체 수정 코드 검출도 있었지만 많은 거짓 양성 결과를 냈다. 그리고 x86 기반 SMC 코드에 McSema를 적용해 얻은 LLVM IR 프로그램에 대해 분석할 때도 같은 문제가 발생하였다.

실험 결과를 요약하면, SMC 정적 분석 방법의 이론적 건전성(soundness)을 C 기반 SMC 벤치마크를 컴파일한 LLVM IR 프로그램에 적용한 결과로 확인할 수 있었지만, 실행 바이너리를 리프팅해서 얻은 LLVM IR 프로그램에 대해서는 리프팅 방법의 한계로 인해 분석 방법의 적용에 한계가 있었다.

정적 분석기의 성능 실험에 대한 더 자세한 내용은 이 참고문헌[8]에 정리되어 있다.

4.3 SMC 벤치마크 대상 정적 분석기 한계 및 대응

Clang을 통해 정형화된 LLVM IR은 정적 분석이 성공적이었다. 하지만 바이너리 코드로부터 McSema를 사용하여 리프팅 된 LLVM IR은 많은 거짓 양성을 검출했다.

Clang으로부터 정형화되는 LLVM IR 과는 다르게 McSema로 리프팅 하게 되는 경우 정적 분석기는 McSema의 리프팅 구현 방식에 영향을 받는다.

McSema는 하드웨어를 LLVM IR 상에서 가상화하여 바이너리 코드가 실제로 하드웨어의 상태를 변경시키는 것을 모사하는 방식으로 리프팅을 한다.

따라서 자체 수정 코드의 명령어로 인해서 수정되는 메모리 영역을 가리키는 포인터 변수가 레지스터를 통해 이용되면, 해당 레지스터를 사용하는 모든 변수와 명령어가 제약식을 통해 토큰이 퍼져 포인터 변수가 오염되어 많은 거짓 양성을 만들어낸다.

이에 대해서 readelf, objdump, remill 과 같은 도구를 사용하여 SMC1, SMC2, SMC3의 x86으로 작성된 바이너리 코드를 수동적으로 직접 리프팅을 한 후 정적 분석기를 적용한 결과, 거짓 양성 없이 실제 정답과 같은 검출 결과를 얻었다.

V. 논의 및 한계

악성코드는 자체 수정 코드를 통해서 정적 분석을 우회하고 프로그램 자체를 분석하기 어렵게 만든다. 따라서 이러한 악성코드를 효과적으로 검출하고 분석하기 위해서는 사전에 자체 수정 코드를 검출하는 것이 중요하다.

이를 위해서 LLVM IR 상에서 자체 수정 코드를 검출하는 정적 분석기를 설계하였고 실제로 정적 분석기를 통해서 자체 수정 코드를 검출하였다.

하지만 바이너리로부터 리프팅 된 코드는 실제 소스 코드로부터 생성되는 LLVM IR과는 다르게 많은 거짓 양성을 내포하고 있었다.

이러한 거짓 양성은 포인터 변수가 오염되어 발생하는 경우로 해당 경우를 사전에 방지할 수 있다면 기능을 개선할 수 있을 것으로 보인다. 구체적인 예로 0x445의 데이터를 레지스터 A에 저장하고, 레지스터 A를 피연산자로 사용하는 경우 해당 포인터 변수를 레지스터 A로 보지 않고, {{0x445}}을 포인터 변수로 보는 것이다. 스택을 사용하는 레지스터의 경우가 가장 대표적인데 이 경우 또한 마찬가지로 StackRegister+8의 값을 레지스터 A에 저장하고 레지스터 A가 피연산자로 사용될 때, 새로운 포인터 변수 {{SP8}}로 표현한다면 포인터 변수의 오염을 방지할 수 있을 것으로 보인다.

x86으로 작성된 SMC1, SMC2, SMC3에 대해서 바이너리 도구를 사용하여 직접 LLVM IR 상으로 정밀하게 리프팅 한 경우 거짓 양성 없이 자체 수정 코드가 검출되었다. 따라서 더 정확한 자체 수정 코드를 검출하기 위해서는 자체 수정 코드의 특징을 반영한 정적 분석에 유리한 LLVM IR 리프터에 대한 연구가 필요하다.

VI. 결론 및 향후 연구

이 논문 연구에서 MIPS로 작성된 자체 수정 코드[6]를 보편적으로 사용되는 x86 코드와 C언어로 재작성하여 벤치마크를 구성했다. 해당 벤치마크를 이용하면 해당 정적 분석기의 개발과 테스트뿐 아니라, 관련된 자체 수정 코드의 연구에 효과적으로 이용할 수 있을 것으로 고려된다.

그리고, 포인터 정적 분석인 앤더슨 알고리즘[4]을 활용하는 자체 수정 코드 정적 분석기를 LLVM 프레임워크 상에서 구현하였다. 이를 통해서 일반화된 정적 분석을 여러 언어로부터 만들어진 LLVM IR과 여러 머신 코드에서 리프팅 된 LLVM IR 코드에 대해서도 효과적으로 일괄적인 정적 분석이 가능할 것이다. 또한 Marcus Botacin의 논문[7]에서 다루어진 것과 같은 동적 분석을 바이너리 코드에 사전 적용 후에 자체 수정 코드가 삽입된 것으로 의심되는 경우 정적 분석을 사용하거나 반대로 정적 분석으로 의심되는 코드 영역을 위주로 동적 분석하는 방안으로 두 가지 방법을 함께 활용한다면 자체 수정 코드 분석이 더욱 효율적일 것이다.

SMC 벤치마크에 정적 분석기를 적용한 결과 Clang으로 만들어낸 정규화된 LLVM IR 코드에 대해서는 자체 수정 코드를 검출할 수 있었지만, McSema에 의해서 리프팅 된 LLVM IR 코드에서는 거짓 양성이 포함되었다. 이는 자체 수정 코드를 검출하기 위한 정적 분석 방법의 오류가 아닌, 리프팅 된 LLVM IR 코드에 의존적인 환경이 만들어낸 덜 민감한 정적 분석(insensitive static analysis)에 대한 거짓 양성이다.

McSema는 IDAPro와 같은 상용프로그램을 디스어셈블러로 사용하여 자체 포맷의 분석 파일인 CFG 파일을 생성한 후, Remill을 사용하여 LLVM IR으로 리프팅 한다. 이때 핵심이 되는 것은 CFG 파일로 McSema가 효과적인 동적 분석을 위해서 리프팅 레벨에서의 Feature을 적용한다.

따라서 좀 더 효과적으로 자체 수정 코드를 분석하기 위해서는 바이너리를 정적 분석에 알맞은 형태로 리프팅하고 추가적인 바이너리 레벨에서의 기능을 반영한 CFG 파일을 생성하고 이를 Remill을 통해서 LLVM IR으로 리프팅하는 기술을 함께 연구해야 한다.

References

[1] RNDr. Petr Rockai, "Decompiling binaries into llvm ir using mcsema and dynist,", Ph.D. Thesis, Masaryk University, 2019.

[2] A. Brown and G. Wilson, The Architecture of Open Source Applications, Lulu Press, May. 2011.

[3] R. Tschüter, J. Ziegenbalg, B. Wesarg, M. Weber, C. Herold, S. Döbel, and R. Brendel, "An LLVM Instrumentation Plug-in for Score-P", Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure, pp. 1-8, Nov. 2017.

[4] B. Hardekopf and C. Lin, "The Ant and the Grasshopper: Fast and accurate pointer analysis for millions of lines of code", Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 290-299, Jun. 2007.

[5] M. Sridharan and S.J. Fink. "The complexity of andersen′sanalysis in practice", International Static Analysis Symposium, pp. 205-221, Aug. 2009.

[6] H. Cai, Z. Shao and A. Vaynberg, "Certified selfmodifying code", Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 66-77, Jun. 2007.

[7] M. Botacin, M. Zanata and A. Grégio, "The self modifyingcode (smc)-aware processor (sap)", Journal of Computer Virology and Hacking Techniques, vol. 16, no. 3, pp. 185-196, Mar. 2020.

[8] K. Choi, Binary target self-correction code identification technique(2020-017), National Security Research Institute, Oct. 2020. https://github.com/dbwodlf3/SMC/blob/master/docs/report.pdf

〈저 자 소 개〉

유 재 일 (Jae-IL Yu) 정회원
2020년 2월: 순천대학교 컴퓨터공학과 졸업
2020년 3월~현재: 전남대학교 인공지능융합과 석사과정
〈관심분야〉 정보보호, 소프트웨어공학, 웹

최 광 훈 (Kwang-hoon Choi) 정회원
1994년 2월: KAIST 전산학과 (공학사)
1996년 2월: KAIST 전산학과 (공학 석사)
2003년 2월: KAIST 전산학과 (공학 박사)
2006년~2010년: LG전자 책임연구원
2011년~2016년 8월: 연세대학교(원주) 컴퓨터정보통신공학부 조교수
2016년9월~2021년 8월: 전남대학교 전자컴퓨터공학부 부교수
2021년9월~현재: 전남대학교 소프트웨어공학과 교수
〈관심분야〉 프로그래밍언어, 컴파일러, 소프트웨어공학