# 자체수정코드를탐지하는정적분석방법의LLVM프레임워크기반구현및실험

유재일

2023년2월

공학 석사학위논문

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

전남대학교 대학원
인 공 지 능 융 합 학 과

유 재 일

2023년 2월

공학 석사학위논문

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


전남대학교 대학원

인 공 지 능 융 합 학 과

유 재 일

2023년 2월

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

이 논문을 공학 석사학위 논문으로 제출함


전남대학교 대학원

인공지능융합학과

유 재 일

지도교수 최 광 훈


유재일의 공학 석사 학위논문을 인준함

심사위원장 박 태 준 (인)

심 사 위 원 최 광 훈 (인)

심 사 위 원 최 석 우 (인)

2023년 2월

# 목 차

# List of figures

# List of tables

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

유 재 일

전남대학교대학원 인공지능융합학과

(지도교수 : 최광훈)

(국문초록)

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 연구에서는 LLVM IR으로 변환한 바이너리 실행 프로그램을 대상으로 자체 수정 코드를 탐지하는 정적 분석 방법을 제안하고, 자체 수정 코드 벤치마크를 만들어 이 방법을 적용했다. 본 연구의 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.

# 1. 서 론

## 가. 연구 배경

1) 서론

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 방법은 크게 동적 분석과 정적 분석으로 나뉜다. 동적 분석은 실제로 프로그램의 실행 흐름을 따라 진행되기 때문에 악성코드를 검출하는 데 있어서 효과적일 수 있다. 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 실제 악성 행위를 할 때만 악성코드를 드러내 악성코드 분석을 어렵게 만든다. 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.

Fig 1.1 은 MIPS 어셈블리어로 작성된 자체 수정 코드의 일부다. 0x200 주소에서 시작하여, modify 라벨로 실행 흐름이 넘어간다. 해당 실행 흐름에서 0x204 주소의 명령어를 0x100 주소의 명령어로 변경한 후 target 라벨로 실행 흐름이 되돌아간다. 따라서 move $2, $4의 명령어는 실행되지 않고 addi $2,

$2, 1의 명령어가 실행된다. 자체 수정 코드를 고려하지 않은 바이너리 분석에서는 addi $2, $2, 1 명령어가 실행되었음을 바로 찾아낼 수 없다.

```
    .data   # Data declaration section

100 new:    addi $2, $2, 1       # the new instr

    .text   # Code section

200 main:   beq  $2, $4, modify  # do modification
204 target: move $2, $4          # original instr
208         j    halt            # exit

212 halt:   j    halt

216 modify: lw   $9, new         # load new instr
224         sw   $9, target      # store to target
232         j    target          # return
```

Fig. 1.1 SMC Example[1]

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

2) 연구목적

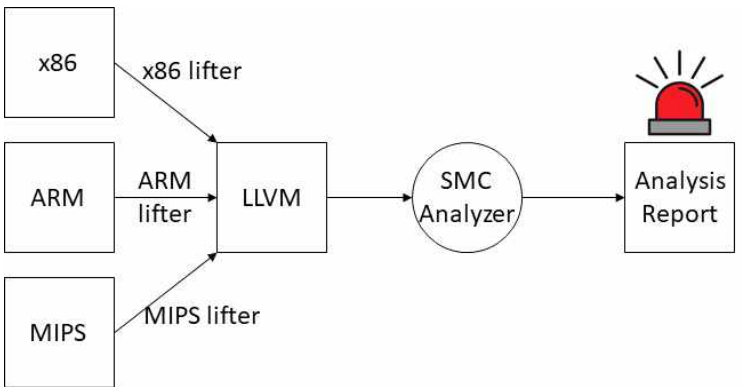



Fig. 1.2 Analyzer Overview

악성코드는 각 바이너리 플랫폼에 맞게끔 변형되어 배포된다. 따라서 특정 바이너리의 악성코드를 하드웨어 종속적인 방법만을 통해서 분석하면 비슷한 악성코드에 대해서 각 바이너리 플랫폼마다 추가적인 악성코드 분석이 필요하다. 따라서 Fig 1.2와 같이 공통의 중간언어로 변환 후에 이를 분석하는 것이 더욱 효

과적이다.

본 연구에서는 그 공통의 중간언어로 LLVM IR을 택했다. LLVM은 C++ 으로 작성된 현대화된 컴파일러 모듈과 툴체인의 집합으로 가장 활성화된 현대적인 컴파일러 프레임워크다. LLVM IR은 특정 언어나 하드웨어에 종속적이지 않은 중간 표현을 이용하여 컴파일러를 구현한다. 해당 프레임워크는 코드 분석과 최적화를 위한 다양한 도구를 갖추고 있다. 이런 오픈소스 생태계를 이용하여 바이너리 분석 도구를 개발한다면 매우 유용할 것이다.

본 연구의 목적은 LLVM IR(Intermediate Representation)에서의 자체 수정 코드 검출이다.

# 2. 관련 연구

### 가. 바이너리 분석

1) 바이너리 계측(Binary Instrumentation)

바이너리 계측은 프로그램에 코드를 삽입하여 분석하는 방법으로, 코드의 삽입 방식에 따라 정적 바이너리 계측(Static Binary Instrumentation)과 동적 바이너리 계측(Dynamic Binary Instrumentation)으로 나뉜다. 구체적인 예를 들어서 프로그램상에서 foo이라는 함수가 몇 번이나 호출되었는지 조사하기 위해서, 함수의 시작점에 호출 횟수를 기록하는 코드를 삽입하는 것이 대표적이다. 자체 수정 코드는 이러한 바이너리 계측에서도 실행 흐름을 어렵게 만들기 때문에 분석을 어렵게 만든다. 따라서 바이너리 계측에서도 휴리스틱을 활용하여 자체 수정 코드를 풀어 쓰는 단계를 거친다[2].

### 나. 자체 수정 코드 검출

많은 악성 코드는 자체 수정 코드를 악용할 때, 공격자에 의해서 수동적으로 작성되기 보다는 바이너리 패커(Binary Packer) 프로그램을 이용하여 자체 수정 코드를 사용하게 된다. 기존의 많은 연구는 이런 패커에 의해서 작성된 자체 수정 코드를 검출하는 것에 초점이 맞추어져 있다.

1) 정적 기반 검출

대표적인 정적 분석 방법으로는 시그니처를 활용한 방법이 있다. 구체적으로 패커의 시그니처를 휴리스틱을 이용해 분석한다. 패커는 패키징된 바이너리를 풀어내야 하기 때문에 일련의 바이너리 시그니처를 가지는 것이 특징적이기 때문에 PEiD[3]]와 같은 프로그램을 통해서 검출될 수 있다. 또 바이너리 프로그램의 헤더 정보와 같은 메타데이터를 활용한 휴리스틱을 통해 분석하는 방법들 또한 시도되고 있다[4].

2) 동적 기반 검출

동적 분석 방법은 IDAPro, GDB, VM, Cuckoo Sandbox 와 같은 제어된 환경에서 이루어진다. 많은 경우에 프로그램을 모니터링해 가며 시스템 API 호출을 중점적으로 분석한다. 이러한 동적 분석은 자체 수정 코드 자체를 검출한다기 보다는 악성 행위에 초점을 맞추어 분석이 이루어진다. 만약 자체 수정 코드를 통해서 실행되는 코드에서 악성 행위가 이루어진다면, 동적 분석을 통해서 이를 파악할 수 있을 것이다.

3) 하이브리드 기반 검출

정적 기반 검출은 동적 기반 검출에 비해 빠르고 그 비용이 적게 든다. 동적 기반 검출은 정적 기반 검출에 비해 그 검출의 효과가 좋지만 느리고 비용이 많이 든다. 따라서 각각의 장점을 살린 검출 정보를 바탕으로 머신 러닝 모델을 만들어 검출 하는 것이 대표적이다[5].

### 다. LLVM IR 리프팅

McSema[6]는 기계어 바이너리를 LLVM IR으로 리프팅하는 오픈 소스 프로그램이다. McSema는 Fig 2.4와 같이 바이너리 코드로부터 CFG(Control Flow Graph)을 생성하는 프론트엔드와 LLVM IR으로 변환하는 백엔드로 나뉜다.

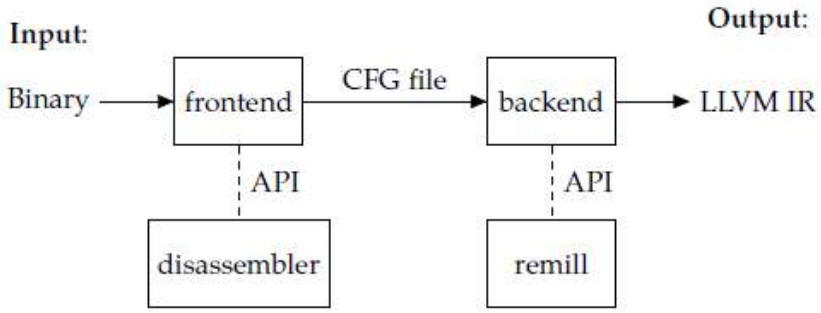



Fig. 2.1 Mcsema Architecture[6]

바이너리 코드로부터 리프팅에 활용되는 CFG을 생성하기 위해서 McSema는 IDAPro을 디스어셈블러로 이용한다. 마찬가지로 CFG 파일 형태로 리프팅된 명령어를 LLVM IR 으로 변환하기 위해서 오픈소스 라이브러리인 remill을 이용한다.

1) Remill

Remill은 McSema에 의해서 생성된 CFG 파일로부터 기계어 명령어를 LLVM IR 으로 맵핑하는데 사용되는 모듈화된 라이브러리다. McSema에서는 Remill 을 사용하여 리프팅된 기계어 명령어를 LLVM IR 으로 변경하고 시뮬레이션된 LLVM IR 으로 통합한다.

Remill은 프로세서나 메모리 같은 하드웨어에 영향을 주는 명령어들은 Instrincis 으로 구분한다. 임의의 기계어 명령어에 대한 Instrincis와 시맨틱 함수(semantic function)가 정의되면 Remill 은 해당 기계어 명령어를 LLVM IR 으로 변환할 수 있다. Remill 에서는 x64를 비롯한 많은 하드웨어 플랫폼 명령어들에 대한 많은 시맨틱 함수를 제공한다.

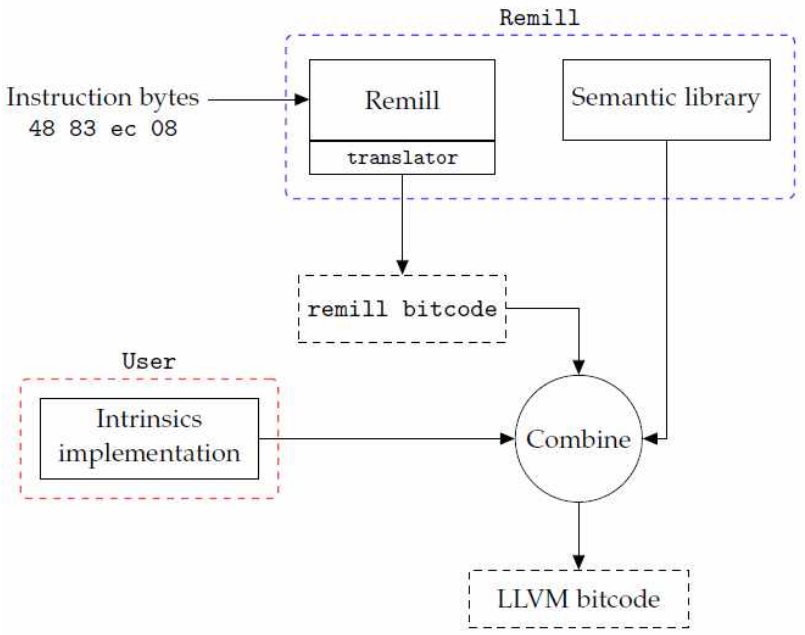



Fig. 2.2 Remill Architecture[6]

예를 들어, 8비트 메모리 영역을 읽고 쓰는 기계어 명령어에 대한 Remill instrincis은 Fig 2.2과 같다.

```
declare i8 @__remill_read_memory_8(%struct.Memory*, i64 %addr)
declare %struct.Memory* @__remill_write_memory_8(
  %struct.Memory*, i64 %addr, i8 %value)
```

Fig. 2.3 Remill Example

Remill은 프로그램의 실행을 가상화하기 위해 실제로 메모리에 영향을 주는 load 명령어는 read instrincis으로, store 명령어는 write instrincise 으로 대체된다.

시맨틱 함수는 C++ 함수로 구현하고 나중에 LLVM 코드로 컴파일된다. 시맨틱 함수의 2개의 인자는 State 구조체 포인터와 Memory 구조체 포인터로 명령어와 무관하게 도입해야 한다. 이 인자들은 시뮬레이션 환경을 제공한다. remill은 C++ 템플릿과 매크로를 사용하여 시맨틱 함수를 구현하였다.

```
template <typename D, typename S1, typename S2>
DEF_SEM(SUB, D dst, S1 src1, S2 src2) {
  auto lhs = Read(src1);  // Read value form source
  auto rhs = Read(src2);
  auto sum = USub(lhs, rhs); // Unsigned substraction
  WriteZExt(dst, sum); // Write into dst
                       // zero extend if types do not match
  WriteFlagsAddSub<tag_sub>(state, lhs, rhs, sum); // Update ArithFlags
  return memory;
}
```

Fig. 2.4 Remill Code Example

DEF_SEM은 새로운 명령어의 시맨틱 함수를 정의하는데 사용하는 매크로이다. 첫 번째 인자는 명령어 이름이고 나머지는 시맨틱 함수에서 사용하는 인자들이다. 각 인자는 쓰임새에 따라 이름을 지었다. 매크로 몸체는 프로세서의 원래 명령어의 동작대로 State 구조체를 읽거나 쓰도록 정의한다. Remill은 매크로 몸체를 정의할 때 유용한 연산자를 제공한다. 예를 들어 Read는 인자가 상수거나 레지스터 어느 경우에도 동작한다. 위 예에서 사용한 함수 호출 구문은 모두 이러한 연산자를 사용한 것이다.

위 매크로를 사용하여 sub 0x20, %rsp 명령어를 리프팅하는 과정은 다음과 같다. Remill은 명령어의 인자와 타입을 분석한다. 적절한 시맨틱 함수를 선택하여 이 (매크로) 함수를 호출하는 코드를 생성한다.

```
%rsp_val = load i64, i64* %RSP
%new_mem = call %struct.Memory* @SUB<i64*, i64, i64>(
    %struct.Memory* %mem, %struct.State* %0,
    i64* %RSP, i64 %rsp_val, i64 32)
```

Fig. 2.5 Remill Lifting Code Example

2) 리프팅

2-1) 세그먼트

CFG 파일의 각 세그먼트는 원래 바이너리의 스코프에 해당하는 전역 변수로 리프팅된다. 세그먼트 크기는 고정되어 있으므로 배열로 표현할 수 있다. 이 전역 변수는 이 세그먼트의 데이터로 초기화하고, 이 전역 변수에 대한 모든 레퍼

런스를 대체한다. 즉, 각 레퍼런스의 주소는 해당하는 코드 요소에 대한 포인터로 바뀐다.

```
Hex dump of section '.rodata':
0x004005b0 01000200 4920616d 20676c6f 62616c20 ....I am global
0x004005c0 73747269 6e6700 string.

Hex dump of section '.data':
0x00601020 00000000 00000000 00000000 00000000 ................
0x00601030 d4054000 00000000 30106000 00000000 ..@.....0.`.....
== 세그먼트 리프팅 ==>
@seg_rodata = internal constant %rodata_type
  <{ [23 x i8] c"\01\00\02\00I am global string\00" }>

@seg_data = internal global %data_type <{ [16 x i8] zeroinitializer,
 i64 add (i64 ptrtoint (%rodata_type* @seg_rodata to i64), i64 4),
 i64 add (i64 ptrtoint (%data_type* @seg_data to i64), i64 16) }>
```

Fig. 2.6 Remill Lifting Segment Example

리프팅 예제에서 .data 섹션에 두 개의 참조, 주소 0x601030 (0x4005b4를 가리킴)의 참조와 주소 0x601038 (앞의 참조를 가리킴)의 참조가 있다. 리프팅 후에 두 개의 섹션은 각각 전역 변수로 표현된다.

이 예제에서 McSema는 .rodata 섹션을@seg_rodata로 .data 섹션을 @seg_data로 리프팅 시켜 LLVM IR 코드를 생성한다. 전역 변수 @seg_rodata는 .rodata 섹션과 동일한 바이트를 해당 크기(23 바이트)의 배열에 담고 있다. constant 키워드는 원래 바이너리에서 이 섹션이 읽기 전용임을 표현한다.

전역 변수 @seg_data는 3개의 요소로 나누어져 있다. 처음 16바이트는 0으로 초기화된 배열이다. 원래 바이너리의 .data 섹션의 처음 16바이트에 해당한다. 그 다음은 @seg_rodata + 4 주소를 갖는 다른 세그먼트를 가리키는 참조와 @seg_data + 16 주소를 갖는 내부 참조다.

2-2) 함수

CFG 파일의 각 함수는 적어도 둘 이상의 함수로 리프팅 된다. 원래 함수를 시뮬레이션 하는 함수는 State 구조체 포인터, 프로그램 카운터, Memory 구조체 포인터를 인자로 받아 Memory 구조체 포인터를 리턴하는 타입으로 정의한다.

```
foo:
  movq $0x42, (%rsi) # 48 c7 06 42 00 00 00
  add $0x1, %rdi     # 48 83 c7 01
  callq boo          # e8 e8 ff ff ff
  retq               # c3
```

== 함수 리프팅 ==>

```
Memory *sub_400570_foo(State *state, int64_t pc, Memory *memory) {
  auto *rip = state->gpr.rip;
  auto *rsi = state->grp.rsi;
  auto *rdi = state->gpr.rdi;

  // movq 0x42, (%rsi)
  *rip += 7;   // add size of the instruction
  memory = MOV<I64, R64W>(memory, state, 0x42, *rsi);

  // add 0x1, %rdi
  *rip += 4;
  memory = ADD<I64, R64, R64W>(memory, state, 0x1, *rdi, rdi);
  // callq boo
  *rip += 5;
  // simulation of the effect of a call instruction on the State
  memory = CALL<I64>(memory, state, address_of_boo_in_binary, *rip);
  // actual call
  memory = sub_400568_boo(memory, rip, state);

  // retq
  *rip += 1;
  memory = RET(memory, state);

  return memory;
}
```

Fig. 2.7 Remill Lifting Function Example

원래 함수 이름 main에 @sub_와 원래 주소 400520을 붙여 이름을 짓는다. 함수 이름 정보가 바이너리에 포함되어 있지 않으면 @sub와 주소만 사용한다.

Wrapper 함수 (indirection wrapper)는 원래 함수와 동일한 이름을 부여한다. 리프팅된 코드가 아닌 곳에서 호출되거나 indirect 함수 호출을 처리하기 위한 게이트웨이 역할을 한다.

CFG 파일의 각 <Basic Block>에 대해 해당하는 기본 블록을 함수 안에 새로 만들고 Remill의 시맨틱 함수에 의해 리프팅시킨 명령어들로 채운다.

# 3. 연구의 내용 및 범위

자체 수정 코드 검출 방법에 관한 연구 수행으로 1) 자체 수정 코드 벤치마크 프로그램 SMC Bench와 2) 정적 프로그램 분석 기반 자체 수정 코드 검출 방법 SMC Analyzyer을 개발하였고, 3) 검출 방법을 벤치마크 프로그램에 적용한 결과를 리포트 한다.

## 가. SMC Bench

SMC Bench는 9가지 자체 수정 동작을 보여주는 코드를 모아놓은 벤치마크 프로그램이다. 해당 프로그램들은 MIPS 기반 자체 수정 코드 9건을 “Certified self-modifying code” PLDI 2007 논문[7]을 참고하여 작성되었다.

기존의 다른 벤치마크는 주로 PE나 ELF 형태의 바이너리로 패커를 사용하여 만들어졌거나 실제 있는 악성 프로그램을 그대로 가져와 이용하지만, 본 연구의 벤치마크는 소스 코드 레벨에서부터 직접 작성된 다양한 형태의 자체 수정 코드 벤치마크를 제공한다. 따라서 연구의 목적에 따라서 수정하여 더 효과적으로 재활용될 수 있다.

SMC Bench는 MIPS 어셈블리 세트 (smc1.mip.s – smc9.mips.s), C 언어 세트(smc1.c – smc9.c), x86 어셈블리 세트 (smc1.x86.s – smc9.x86.s)으로 27개의 소스 파일이 있다. 해당 프로그램은 다음의 깃허브 저장소에서 확인할 수 있다. https://github.com/dbwodlf3/SMC/tree/master/src/smcbench/smc

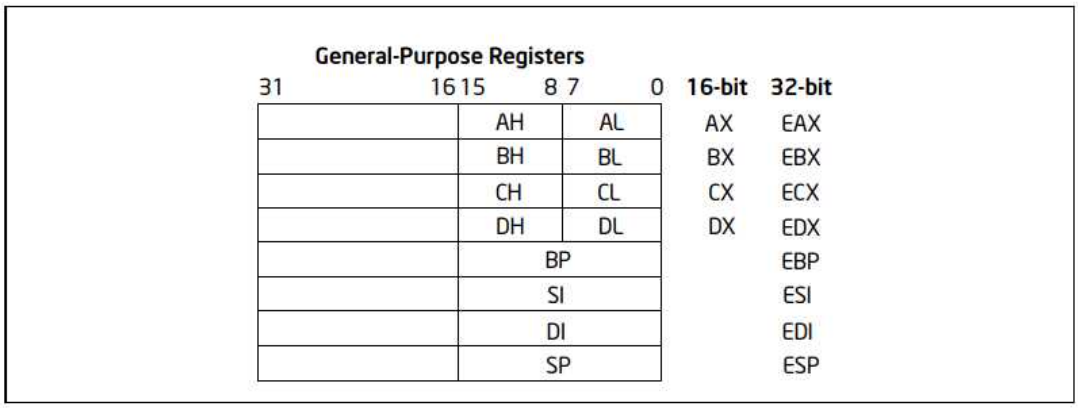



Fig. 3.1 X86 General Purpose Register

x86 Architure는 CISC 구조로 MIPS에 비하여 그 구조고 복잡하다. 레지스터 또한 단순 32비트 레지스터가 아니라, Fig 3.10과 같이 8비트 레지스터, 16비트 레지스터, 32비트 레지스터로 구성되어 있고 명령어의 길이와 형식 또한 재각각이다. 자체 수정 코드를 작성하고 실행하기 위해서는 레지스터와 메모리 주소의 계산에 유의해야 하는 어려움이 있다.

C 언어로 작성된 자체 수정 코드는 컴파일러의 영향을 받는다. 여러 가지 옵션에 의해서 생성되는 바이너리의 양상이 달라진다. Position Independent Code와 같은 옵션을 통해서 바이너리를 생성하게 되는 경우 프로그램의 메모리 주소가 변하기 때문에 이러한 바이너리의 메모리 주소와 관련된 모든 특성을 고려 해야 하는 어려움이 있다.

Table. 3.1 SMC Bench List

| 이름 | 설명 |
|---|---|
| smc1 | 피보나치 수열(Unbounded code rewriting) |
| smc2 | 런타임 코드 검사(Runtime code checking) |
| smc3 | 런타임 코드 생성(Runtime code generation) |
| smc4 | 다중 런타임 코드 생성(Multilevel runtime code generation) |
| smc5 | 스스로 바꾸고 다시 돌아오는 코드(Self-mutating code block) |
| smc6 | 서로 상대를 변경하는 코드(Mutual modifying modules) |
| smc7 | 스스로를 복제하는 코드(Self-growing code) |
| smc8 | 동일한 알고리즘을 여러 형태로 바꾸는 코드(Polymorphic code) |
| smc9 | 암호/압축을 푸는 코드(Encrypting and decrypting code) |

1) MIPS Benchmark

MIPS Architecture는 Instruction의 크기가 4byte 로 항상 일정한 RISC 구조이기 때문에 상대적으로 SMC 코드를 작성하는데 주소 계산이 간편하다는 장점이 있다. MIPS 코드는 MARS[8]을 이용하여 작성되었다.

1-1) SMC1.MIPS

```
1       .data
2 num:   8 .byte 8
3
4     .text
5 init:
6     lw $4, num  # Set Argument
7     lw $9, key  # $9 = Ec(add $2 $2 0)
8     li $8, 1    # counter
9     li $2, 1    # accumulator
10
11 loop:
12     beq $8, $4, halt # check if done
13     addi $8, $8, 1   # inc counter
14     add $10, $9, $2  # new instr to put
15 key:
16     addi $2, $2, 0
17     sw $10, key      # store new instr
18     j loop            # next round
19
20 halt:
21      j halt
```

Fig. 3.2 SMC1.MIPS.S

해당 프로그램은 피보나치 수열의 값을 구하는 프로그램이다. 구하고자 하는 n 번째 항은 num을 통해서 초기화 된다. 16번 줄이 피보나치의 이전항과 이후의 항을 더 하는 명령어로, 바로 다음 17번 줄의 명령어를 통해서 16번 줄의 명령어에서 3번째 오퍼랜드 값이 이전 값으로 변경된다.

1-2) SMC2.MIPS

```
1 .text
2 main:
3       jal f
4       move $2, $8
5       j halt
6
7 f:
8       li $8, 42
9       lw $9, -4($31)
10      lw $10, addr
11      bne $9, $10, halt
12      jr $31
13
14 halt :
15       j halt
16
17 addr:
18       jal f
```

Fig. 3.3 SMC2.MIPS.S

해당 프로그램은 자신의 코드를 읽는 코드(self-reading code)를 보여주기 위한 목적의 코드다. jal f 명령어를 통해서 f 으로 실행흐름이 넘어간다. ret 명령을 통해서 돌아올 때, 돌아가는 실행흐름의 앞에 있는 명령어가 정말로 jal f 인지 확인한다. 악성 프로그램의 경우 리턴 주소 값을 수정하여 악성 바이너리 코드로 실행흐름이 이동할 수 있다. 위 MIPS 예제 코드는 실행 시간에 코드를 검사하여 이런 시도를 막는 법을 보여준다.

1-3) SMC3.MIPS.S

```
1     .data
2 vec1:   .word 22, 0, 25
3 vec2:   .word 7, 429, 6
4 result: .word 0
5
6         .text              # Code section
7 main:   li $4, 3
8         li $8, 0
9         la $9, gen
10        la $11, tpl
11        lw $12, 0($11)
12        sw $12, 0($9)
13        addi $9, $9, 4
14
15 loop:   beq $8, $4, post
16         # ...
17
18 next:   addi $8, $8, 1
19         j loop
20
21 post:   lw $12, 20($11)
22         # ...
23
24 tpl:    li $2, 0           # template code
25         # ...
26
27 #Generated code of vector dot product
28 gen:    li $2, 0      # int gen(int *v)
29    lw $13, 0($4)      # {
30    li $12, 22         # int res = 0;
31    mul $12, $12, $13  # res += 22 * v[0];
32    add $2, $2, $12    # res += 25 * v[2];
33    lw $13, 8($4)      # return res;
34    li $12, 25         # }
35    mul $12, $12, $13
36    add $2, $2, $12
37    jr $31
38
39    nop
40    nop
```

Fig. 3.4 SMC3.MIPS.S

SMC3.MIPS.S 는 실행시간에 코드 생성을 하는 것을 보여주기 위한 목적의 코드다. 두 백터의 내적을 구하는 프로그램으로, 구할 내적의 데이터가 주어지면, 프로그램을 실행시간동안에 직접 생성한다.

tpl 라벨의 코드는 생성할 명령어의 템플릿 코드이고, gen은 만들어지는 새로운 명령어의 라벨이다.

1-4) SMC4.MIPS

```
1     .text
2 main: la $9, gen        # get the target addr
3       li $8, 0xac880000 # load Ec(sw $8,0($4))
4       sw $8, 0($9)      # store to gen
5       li $8, 0x00800008 # load Ec(jr $4)
6       sw $8, 4($9)      # store to gen+4
7       la $4, ggen       # $4 = ggen
8       la $9, main       # $9 = main
9       li $8, 0x01200008 # load Ec(jr $9) to $8
10      j gen             # jump to target
11
12gen:  nop               # to be generated
1       nop
2
3 ggen: nop
```

Fig. 3.5 SMC4.MIPS.S

SMC4.MIPS.S는 런타임 코드 생성의 중첩성을 보여주기 위한 목적의 코드이다. 프로그램을 실행하면서 생성한 코드를 실행하면 또 다른 코드를 생성하는 방식이다. main 에 의해서 gen의 코드가 생성되고, 생성된 gen의 코드에 의해서 ggen의 코드가 생성된다.

1-5) SMC5.MIPS

```
1      .text
2  main:
3      la $8, g
4      lw $9, 0($8)
5      addi $10, $9, 4
6      sw $10, g
7      lw $11, h
8  g:   sw $9, 0($8)
9  h:   j dead
10     sw $11, h
11     j main
12
13
14 dead:    # ...
```

Fig. 3.6 SMC5.MIPS.S

SMC5.MIPS.S는 스스로 바꾸고 다시 원래 코드로 돌아오는 코드를 보여주기 위한 목적의 코드이다. 단순한 실행흐름상 j dead 으로 진행이 될 것 같지만, j dead 명령어가 수정되어 실행되지 않는다. 해당 명령어를 넘어가고 다음 명령어에서 수정되어서 실행되었던 명령어는 사라지고 본래의 명령어로 돌아간다. 결과적으로 j dead 는 실행되지 않고, j main 이 실행된다.

1-6) SMC6.MIPS

```
1      .text
2  main:
3      j alter
4      sw $8, alter
5
6  alter:
7      lw $8, main
8      li $9, 0
9      sw $9, main
10     j main
```

Fig. 3.7 SMC6.MIPS.S

SMC6.MIPS는 서로를 변경하는 코드을 보여주기 위한 목적의 코드이다. main에 의해서 alter의 코드가 변경되고, alter에 의해서 main의 코드가 변경된다.

1-7) SMC7.MIPS

```
1     .text
2 main:
3     la $8, loop
4     la $9, new
5     move $10, $9
6
7 loop:
8     lw $11, 0($8)
9     sw $11, 0($9)
10    addi $8, $8, 4
11    addi $9, $9, 4
12    bne $8, $10, loop
13    move $10, $9
14 new:
15    lw $11, 0($8)
16    sw $11, 0($9)
17    addi $8, $8, 4
```

Fig. 3.8 SMC7.MIPS.S

SMC7.MIPS.S는 자신의 코드를 읽는 기법과 런타임 코드 생성을 응용하여 스스로를 복제하는 코드를 보여주기 위한 목적의 코드이다. 처음 초기화 이후 라벨 loop에서부터 6개의 명령어들을 라벨 new에 복사하고, 라벨 new의 복제된 명령어를 실행하여 그 뒤에 다시 반복해서 복사한다. 마치 바이러스가 자가복제 하는 것처럼 자기 자신의 코드를 계속 복제한다.

1-8) SMC8.MIPS

```
1
2 main: la $10, body
3
4 body: lw $8, 12($10)
5       lw $9, 16($10)
6       sw $9, 12($10)
7       addi $2,$2, 21
8       addi $2,$2, 21
9       sw $8, 16($10)
10      lw $9, 8($10)
11      lw $8, 20($10)
12      sw $9, 20($10)
13      sw $8, 8($10)
14      j body
```

Fig. 3.9 SMC8.MIPS.S

SMC8.MIPS는 코드의 원래 동작은 유지하면서 형태를 바꾸는 코드를 보여주기 위한 목적의 코드다. 예제 프로그램은 실행할 때 명령어의 순서가 바뀐다. 이 프로그램의 순서가 바뀌지만, 레지스터 $2에 42를 더하는 동작은 항상 같다.

1-9) SMC9.MIPS

```
1     .text
2 main:
3     la $8, pg
4     la $9, pgend
5     li $10, 0xffffffff
6
7  xor1:
8     lw $11, 0($8)
9     xor $11, $11, $10
10    sw $11, 0($8)
11    addi $8, $8, 4
12    blt $8, $9, xor1
13 decr:
14    la $8, pg
15    la $9, pgend
16    la $10, 0xffffffff
17 xor2:
18    lw $11, 0($8)
19    xor $11, $11, $10
20    sw $11, 0($8)
21    addi $8, $8, 4
22    blt $8, $9, xor2
23    j pg
24 halt:
25    j halt
26 pg:
27    li $2, 1
28    li $3, 2
29    add $2, $2, $3
30    j halt
31 pgend:
```

Fig. 3.10 SMC9.MIPS.S

SMC9.MIPS는 실행 중에 암호화된 코드를 풀어 실행하는 코드를 보여주기 위한 목적으로 작성되었다. 특히 암호화된 코드가 메모리 영역에 그대로 코드를 생성한다. 암호화 루틴과 암호를 푸는 루틴은 서로 독립적으로 구성되어 있다. 암호화한 코드와 암호를 푸는 루틴을 함께 바이너리를 만들어 저장하고 로딩하면서 풀어 실행할 수 있다.

2) x86 Benchmark

x86의 코드 또한, MIPS 코드를 원본으로 하여 x86 아키텍처에 맞게끔 재작성 했다. NASM 어셈블러를 기반으로 작성하였고 x86 기반의 리눅스 운영체제에서 작동한다.

2-1) SMC1.x86

```
1   section .data
2     num db 8
3
4   section .text
5   global _init
6   _init:
7     mov ebp, num
8     mov ecx, 1
9     mov edx, 1
10
11  loop:
12    cmp ecx, [ebp]
13    jz halt
14    add ecx, 1
15    mov bl, dl
16
17  key:
18    add edx, 0
19    mov al, bl
20    mov [key+2], al
21
22    jmp loop
23
24  halt:
25    mov eax, 1
26    xor ebx, ebx
27    int 0x80
```

Fig. 3.11 SMC1.X86.S

MIPS 코드와 마찬가지의 일을 한다. X86 코드상에서 add 명령어의 피연산자

의 값을 변경해나가며 피보나치 수열의 값을 구한다.

MIPS 에서는 명령어의 크기가 4byte 이어서 명령어를 데이터로 보아서 명령어 전체에 피연산자 값만을 증가시켜줘야 했지만, X86에서는 8bits 레지스터를 사용하여 오퍼랜드의 값만을 수정할 수 있다.

2-2) SMC2.x86

```
1 section .text
2 global _start
3   _start:
4       nop
5       mov eax, 0x804806c
6       call eax                ; jal f
7       mov ebx, edi    ; move $2, $8
8       jmp halt                ; j halt
9
10  f:
11      mov edi, 42
12      mov ebp, [esp]
13      mov cx, [ebp-2]
14      mov esi, addr           ; esi <- addr of addr
15      mov dx, [esi+5]
16      cmp cx, dx
17      jne halt
18      ret
19
20  halt:
21      mov eax, 1
22      int 0x80                ; j halt
23
24  addr:
25      mov eax, 0x804806c
26      call eax
```

Fig. 3.12 SMC2.X86.S

X86 에서는 CALL 명령어와 RET 명령어를 통해서 실행흐름을 변경할 수 있다. 함수 f을 호출할 때, f의 주소를 명시적으로 입력하여 호출하였다.

MIPS 코드와는 다르게 X86에서는 리턴주소의 값을 스택에 저장하게 된다. call f 명령어 즉, call eax 명령어의 길이는 16bits 으로 16bits 레지스터인 cx

와 dx을 통해서 그 값을 비교한다.

2-3) SMC3.x86

```
1 section .text
2 _vec1 dd 22, 0, 25
3 _vec2 dd 7, 429, 6
4 result dd 0
5
6 global _start
7
8   _start:
9      mov eax, 3
10     mov ecx, 0
11     mov edi, gen
12     mov ebx, tpl
13
14     mov edx, [ebx]       ; edx <- tpl 1st instruction
15     mov [edi], edx       ; gen 1st instruction <- tpl 1st instruction
16                          ; overwriting
17     mov [edi+1], word 0  ; 00 00
18     mov [edi+3], word 0
19
20     add edi, 5           ; edi <- 2nd instruction
21
22  loop:
23     cmp eax, ecx         ; loop cycle count 3
24     je post
25     mov ebp, 4
26     imul ebp, ecx        ; vec index
27
28     mov esi, dword [_vec1+ebp]
29     cmp esi, 0           ; if vec's value 0 -> jmp next //vec1 2nd value
30     je next
31
32     mov dx, [ebx+5]      ; gen 2nd instruction <- tpl 2nd instruction
33                          ; overwriting
34     mov [edi], dx
35     mov edx, ebp
36     mov [edi+2], dl
37
38     mov dx, [ebx+8]      ; gen 3rd instruction <- tpl 3rd instruction
39                          ; overwriting
```

```
40     mov [edi+3], dx
41
42     mov dl, [ebx+10]     ; gen 4th instruction <- tpl 4th instruction
43                          ; overwriting
44     mov [edi+5], dl
45     mov edx, esi
46     mov [edi+6], dl
47     mov [edi+7], byte 0
48     mov [edi+8], word 0
49
50     mov edx, [ebx+15]    ; gen 5th instruction <- tpl 5th instruction
51                          ; overwriting
52   mov [edi+10], edx
53
54     mov dx, [ebx+19]     ; gen 6th instruction <- tpl 6th instruction
55                          ; overwriting
56     mov [edi+14], dx
57
58     add edi, 16
59
60  next:
61     add ecx, 1
62     jmp loop
63
64  post:
65    mov dx, [ebx+21]       ; gen 7th instruction <- tpl 7th instruction
  overwriting
66    mov [edi], dx
67
68    mov esi, _vec2
69    mov esp, post
70    add esp, 21          ; esp <- post 7th instruction (mov [result], eax)
71    jmp gen
72
73    mov [result], eax
74
75    mov eax, 1           ; exit
76    xor ebx, ebx
77    int 0x80
78
79  tpl:
80    mov eax, 0
81    add esi, 0
82    mov ebp, [esi]
```

```
83    mov edx, 0
84
85    imul edx, ebp
86    nop
87    add eax, edx
88    jmp esp              ; jmp post
89    nop
90
91  gen:
92    ; ...
```

Fig. 3.13 SMC3.X86.S

MIPS 코드와 마찬가지로 벡터의 내적을 구하는 자체 수정 코드이다. MIPS 코드와는 다르게 각 명령어의 길이가 달라서 세밀하게 메모리의 주소를 계산하여 명령어를 써야한다. 결과적으로 gen 주소에 명령어를 작성하고, 마지막으로 post의 주소로 실행흐름이 변경되어 result 에 결과값을 저장한다.

2-4) SMC4.x86

```
1 section .text
2 global _start
3
4   _start:
5     mov eax, [copy1]
6     mov ebx, [copy1+5]
7     mov [gen], eax
8     mov [gen+5], ebx
9     mov eax, [copy2]
10    mov ebx, [copy2+5]
11    mov [ggen], eax
12    mov [ggen+5], ebx
13    jmp gen
14
15  copy1:
16    mov eax, ggen
17    jmp eax
18  copy2:
19    mov eax, _start
20    jmp eax
```

```
21
22  gen:
23     nop
24     nop
25     nop
26     nop
27     nop
28     nop
29     nop
30  ggen:
31     nop
32     nop
33     nop
34     nop
35     nop
36     nop
37     nop
```

Fig. 3.14 SMC4.X86.S

SMC4.MIPS와 마찬가지로 gen에 코드를 생성하고, gen에 생성된 코드가 ggen에 코드를 생성한다. _start 레이블에서 copy1 레이블의 명령어 코드를 gen 레이블에 덮어쓴다. 다음으로 gen 레이블이 실행되며, copy2 레이블의 내용을 ggen 레이블에 덮어쓴다.MIPS 에서는 nop 명령어가 4byte 이지만, x86의 경우에 단순 nop 명령어의 길이는 1byte다. 생성하는 명령어의 길이가 7bytes 이므로 7개의 nop 명령어를 사용했다.

2-5) SMC5.x86

```
1 section .text
1 global _start
2
3   _start:
4       nop
5       mov esi, 0              ; count = 0
6       mov eax, g              ; eax <- load address of g
7       mov ebx, [eax]          ; ebx <- load instruction of g
8                               ; (mov [eax], ebx)
```

```
9       add eax, 7      ; eax <- load address of h
10      mov edx, [h]    ; edx <- load instruction of h (jmp dead)
11  g:
12      mov [eax], ebx ;
13      nop
14      nop
15      sub eax, 7    ; eax <- load address of g
16  h:
17      jmp dead
18      nop
19      nop
20      mov [h], edx
21      add esi, 1              ; count += 1
22      cmp esi, 2              ; if(count == 2) jmp dead:
23      je dead
24      mov edi, _start         ; edi <- load address of _start
25      add edi, 6              ; edi <- load address of _start + 6
26                              ; (mov eax, g)
27      jmp edi       ; jmp _start
28
29  dead:
30      mov eax, 1
31      int 0x80
```

Fig. 3.15 SMC5.X86.S

Fig 3.15에서, 17번째 줄의 jmp dead는 실행되지 않고 그 앞의 명령어들에 의해서 다른 명령어로 변환된다. jmp dead 명령어가 실행되지 않으므로, 그 다음의 명령어들이 실행되고 바뀌었던 jmp dead 명령어가 다시 jmp dead 명령어로 복원된다. 총 2번 실행되어서, 처음에는 jmp dead 명령어가 실행되지 않지만 두 번째에는 dead 명령어로 실행 흐름이 넘어가게 된다.

2-6) SMC6.x86

```
1 section .text
2 global _start
3
4   _start:
5     jmp alter
6     mov [alter], ax          ; [alter](mov ax, [_start]) <-
```

```
7                               ; load instruction [ax] (jmp alter)
8
9
10  alter:
11    mov ax, [_start]          ; ax <- load instruction [_start]
12                              ; (jmp alter)
13    mov bl, 0x90              ; bl <- load instruction [0x90] (nop)
14    mov [_start], bl
15    mov [_start+1], bl        ; instruction [_start]
16                              ; (jmp alter) is changed (nop)
17    mov [re_start], bl
18    mov [re_start+1], bl      ; instuction [re_start]
19                              ; (jmp halt) is changed (nop)
20
21  re_start:
22    jmp halt                  ; this doesn't execute.
23    jmp _start
24
25  halt:
26    nop
```

Fig. 3.16 SMC6.X86.S

첫 실행에 의해서 _start의 jmp alter 명령어와 re_start 의 jmp halt 명령어가 nop 명령어로 변경된다. 따라서 alter에서 re_start 명령어를 수정하고, 수정된 re_start의 명령어에 의해서 _start의 mov [alter], ax 명령어가 또 다시 alter 의 첫 번째 명령어를 수정한다.

2-7) SMC7.x86

```
1   _start:
2     nop             ; gdb point
3     mov eax, loop   ; eax <- addr of loop
4     mov ebx, new
5     mov ecx, new
6     mov esi, 2
7     mov edi, 4
8     mov ebp, 0
9
```

```
10  loop:
11    mov edx, [eax]  ; edx <- instruction of loop
12    mov [ebx], edx  ; [ebx] <- store instruction of loop
13                    ; in pointer of ebx
14    add eax, esi    ; eax += 2, instruction length is 2byte
15    add ebx, esi    ; ebx += 2
16    cmp eax, ecx    ; comparison about eax and ecx
17    jne loop        ; if not equal, than jump to the loop.
18    mov ecx, ebx    ; if equal, all code is copied in new label
19    add ebp, esi
20    cmp ebp, edi
21    je halt         ;halt is relative address, so if we repeat
22                    ; 2 times code, we need to subtract value
23  new:
24    nop
25    nop
26    ; nop ...
27
28  halt:
29    mov eax, 1
30    int 0x80
```

Fig. 3.17 SMC7.X86.S

loop의 명령어를 2byte 씩 읽어서 new에 작성한다. halt 명령어 위에 덮어 쓰면 안되기 때문에 nop 명령어로 충분한 크기의 주소공간을 확보한다.

2-8) SMC8.x86

```
1 section .text
2 global _start
3
4   _start:
5     mov eax, body           ; eax <- load address of body
6
7   body:
8     mov ebx, [eax + 12]      ; ebx <- load instruction [eax + 12]
9                              ; (add esi, 10)
10    nop
11    mov ecx, [eax + 16]     ; ecx <- load instruction [eax + 16]
```

```
12                              ; (add esi, 10)
13      nop
14      mov [eax + 12], ecx     ; allocate instruction of ecx
15                              ;in pointer of eax + 12
16      nop
17      add esi, 10             ; add esi = esi + 10
18      nop
19      add esi, 10             ; add esi = esi + 10
20      nop
21      mov [eax + 16], ebx     ; allocate instruction of ebx
22                              ; in pointer of eax + 16
23      nop
24      mov ecx, [eax + 8]      ; ecx <- load instruction [eax + 8]
25                              ; (mov [eax + 12], ecx)
26      nop
27      mov ebx, [eax + 20]     ; ebx <- load instruction [eax + 20]
28                              ; (mov [eax + 16], ebx)
29      nop
30      mov [eax + 20], ecx     ; allocate instruction of ecx
31                              ; in pointer of eax + 20
32      nop
33      mov [eax + 8], ebx      ; allocate instruction of ebx
34                              ; in pointer of ebx + 8
35      nop
36      add edx, 1
37      cmp edx, 2
38      jnz body                ; jmp body
39      mov eax, 1
40      mov ebx, 0
41      int 0x80
```

Fig. 3.18 SMC8.X86.S

SMC8.MIPS와 마찬가지로 바이너리 코드가 변형되지만, 프로그램의 의도는 변경되지 않는다. MIPS와 달리 명령어의 길이가 다르기 때문에 nop 명령어를 추가하여 명령어의 길이를 맞추어 코드를 수정한다.

2-9) SMC9.x86

```
1 section .text
```

```
2 global _start
3
4   _start:
5     mov eax, pg             ; eax <- load address of pg
6     mov ebx, pgend          ; ebx <- load address of pgend
7     mov ecx, 0xffffffff     ; ecx <- assign value of 0xffffffff
8
9   xor1:
10    mov edx, [eax]          ; store instruction that pointed
11                            ; by eax in edx
12    xor edx, ecx            ; xor edx, ecx, and store result value
13                            ; in edx
14    mov [eax], edx          ; store instruction in edx
15                            ; where eax points
16    add eax, 4              ; add 4 to eax
17    cmp eax, ebx            ; compare eax and ebx so if eax is
18                            ; less than ebx
19    jl xor1                 ; loop xor1
20                            ; so code of xor1 converts pg code
21                            ; to xor form
22  decr:
23    mov eax, pg             ; eax <- reload address of pg
24    mov ebx, pgend          ; ebx <- reload address of pg
25    mov ecx, 0xffffffff     ; ecx <- reassign value of 0xffffffff
26
27  xor2:
28    mov edx, [eax]          ; store instruction that pointed
29                            ; by eax in edx
30    xor edx, ecx            ; xor edx, ecx, and store result value
31                            ; in edx
32    mov [eax], edx          ; store instruction in edx where
33                            ; eax points
34    add eax, 4              ; add 4 to eax
35    cmp eax, ebx            ; compare eax and ebx so if eax is
36                            ; less than ebx
37    jl xor2                 ; jmp xor2
38    jmp pg                  ; else jmp pg
39
40  halt:
41    jmp halt                ; when program get here,
42                            ; the program end
```

```
43  pg:
44    add esi, 10             ; meaningless code
45    nop
46    add edi, 10             ; meaningless code
47    nop
48    jmp halt
49    nop
50    nop
```

Fig. 3.19 SMC9.X86.S

xor1에서, pg의 4byte을 xor 연산자를 통하여 암호화 한 다음, 암호화된 코드를 pg에 재작성한다. decr 블록에서 복호화를 하기 위한 초기 값을 레지스터에 할당한 다음, xor2에서 다시 한번 xor 연산자를 사용해 복호화된 코드를 pg 블록에 재작성 한다.

3) C Benchmark

C언어 벤치마크는 리눅스 x86-x64 아키텍처에서 gcc 컴파일러 상에서 동작하도록 작성되었다. C언어로 작성되었지만 자체 수정 코드의 실행을 위해서는 해당 자체 수정 코드가 실행될 플랫폼에서의 메모리 주소 계산에 더 유의를 구해야 한다. 또 앞의 어셈블리와는 다르게 C언어로 작성된 프로그램의 경우에는 바이너리 코드가 전적으로 컴파일러에 의해서 생성되므로 컴파일러의 최적화 또한 생각해야 한다.

3-1) SMC1.C

```
memcpy(instr9, ptr_key, 4);

loop:
    if (index == cnt) goto halt;
    index = index + 1;
    memcpy( instr10, instr9, 4);
    instr10[3] = instr10[3] + fib_index  - 1;
key:
    fib_index = fib_index + 1;
    memcpy(ptr_key, instr10, 4);
    goto loop;
halt:
    printf("fib(%d)=%d₩n", cnt, fib_index);
    return 0;
```

Fig. 3.20 SMC1.C

SMC1.MIPS 와 같이 피보나치 수열의 값을 구할 때. add 명령어의 피연산자 값을 계속해서 변경하여 그 값을 구한다. 앞의 경우와는 다르게 memcpy 함수를 호출하여 메모리의 코드 값을 변경하게 된다. 이 때 memcpy 함수의 쓰는 값이 4 bytes 이므로 컴파일러는 이것을 최적화해 memcpy을 호출하는 것이 아니라 mov 명령어로 이것이 대체될 수 있다.

3-2) SMC2.C

```
void f(){
  //.... local variables
  *ptr_reg8 = 42;
  reg31 = (unsigned char*)&ptr_reg8 + 8 + 71; // ret address
  // ....
}
```

Fig. 3.21 SMC2.C

SMC2.MIPS 에서와 같이 직접적인 자체 수정 코드는 없고, 함수의 실행흐름이 변경되었는지 감지하는 코드이다. 앞의 두 어셈블리와는 다르게 C 언어에서는 함수의 지역 변수는 스택에 저장되는데, 함수 호출시에 가장 앞에 리턴 주소

가 저장이 되고, 그 뒤에 지역 변수의 저장공간으로 사용된다. Fig 3.21 에서와 같이 리턴 주소를 알기 위해서는 지역 변수의 사용 또한 고려해야 한다.

3-3) SMC3.C

```
int main(void){
  // ... local variables
    unsigned char*  ptr_gen_reg9;
    unsigned char*  ptr_tpl_reg11;
    unsigned char*  ptr_tpl_end;
    unsigned char*  ptr_tpl_body;
  // ...
}
```

Fig. 3.22 SMC3.C

SMC3.C 에서 코드를 생성하는데 사용할 템플릿 코드의 위치를 구하기 위해서 해당 명령어의 위치를 가리키는 포인터 변수를 사용했다. loop을 돌며 포인터 변수를 사용해 명령어를 생성하고 그 결과 값을 출력 한다.

3-4) SMC4.C

```
unsigned char   store_instruction[SIZE_OF_SW] = {
  /* binary instructions
   * ....
   */
};
/* ... */
int main(void){
  // ...
  memcpy(reg8, store_instruction, SIZE_OF_SW);
  // ...
  memcpy(reg8, jr_instruction, SIZE_OF_JR);
  /* ... */
}
```

Fig. 3.23 SMC4.C

실행하고자 하는 명령어의 바이너리 코드를 데이터로 변수로 저장하여, 해당 변수의 값을 reg8 이라는 포인터 변수를 통해서 자체 수정 코드를 작성한다.

3-5) SMC5.C

```
int main(void){
  // ..local variables

  start:
        ptr_h = (unsigned char*)main + OFFSET_H;

        // la $8, g
        ptr_g = (unsigned char*)main + OFFSET_G;

        // lw $9, 0($8)a
        for (i=0; i< SIZE_G; i++) instr9[i] = ptrg[i];

        // addi $10, $9, 4
        for (i=0; i < SIZE_G; i++) instr10[i] = instr9[i];
        // change reference at ptr_g to ptr_h
        instr10[21] = instr10[21] + NEXT_INST;

        // sw $10, g
        for(i=0; i < SIZE_G; i++) ptr_g[i] = instr10[i];

        // lw $11, h
        for(i=0; i < SIZE_H; i++) instr11[i] = ptr_h[i];
  g:
        // sw $9, 0($8)
        for (i=0; i < SIZE_H; i++) ptr_g[i] = instr9[i];

  h:
        num *= 2; // 6byte Instruction
        num *= 2;
        num *= 2;
        num *= 2;
        num *= 2;
        num *= 2;
        num *= 2;
        num *= 2;
        num *= 2; // so size of all instructions is 54byte

        // sw $11, h
        for (i=0; i< SIZE_H; i++) ptr_h[i] = instr11[i];

        printf("Num Value : %d₩n", num);
        // change reference at ptr_h to ptr_g
        ptr_g[21] = ptr_g[21] + BEFORE_INST;

        //j main
        goto start;
  dead:
         return 0;
}
```

Fig. 3.24 SMC5.C

instr9 변수에 g 라벨의 명령어 값들이 저장된다. instr11 변수에 h 라벨의 명령어 값들이 저장된다. 이후에 for 문을 통해서 각 변수들로부터 명령어 값을 이용하여 h 라벨의 num *= 2 명령어를 g 라벨의 명령어로 수정 후 실행한 다음, for 문을 사용하여 본래의 num *= 2 명령어를 복원한다.

3-6) SMC6.C

```
int main(void){
// ..local variables

num = 1;
ptr_goto_alter = (unsigned char*)main + 192;
ptr_main = (unsigned char*)main + 194;
ptr_alter = (unsigned char*)main + 288;
for(i=0; i<SIZE_GOTO_ALTER; i++) instr_j_alter[i] = ptr_goto_alter[i];
for(; i<SIZE_MAIN; i++) instr_j_alter[i] = '₩x90';

goto_alter:
        goto alter;

main_start:
        // j alter
        // dummy code to be overrwritten
        num *= 2;   // 6 bytes
        num *= 2;   // 6 bytes
        num *= 2;   // 6 bytes
        num *= 2;   // 6 bytes
        num++;      // 7 bytes

        // sw $8, alter
        for(i=0; i<SIZE_ALTER; i++) ptr_alter[i] = instr8[i];

alter:
        // lw $8, main
        for(i=0; i<SIZE_MAIN; i++) instr8[i] = ptr_goto_alter[i];

        instr8[1] = '₩xfe';

        // li $9, 0
        for(i=0; i<SIZE_MAIN; i++) instr9[i] = '₩x90';    // 0x90: x86
NOP instruction

        // sw $9, main
        for(i=0; i<SIZE_MAIN; i++) ptr_main[i] = instr9[i];

        goto main_start;

}
```

Fig. 3.25 SMC6.C

첫 실행시 alter 레이블에서 main_start 레이블의 명령어를 수정한다. 수정된 main_start 명령어는 jmp alter 명령어를 alter 의 첫 명령어로 수정한다.

3-7) SMC7.C

```
start:
  offset = 0;
  ptr_loop = (unsigned char*) main + LOOP;
  ptr_new  = (unsigned char*) main + NEW;

loop:
  for(i=0; i<SIZE_LOOP; i++)
    (ptr_new + offset)[i] = ptr_loop[i];
  offset += SIZE_LOOP;

new:
  return 0;
```

Fig. 3.26 SMC7.C

loop 레이블의 명령어를 new 라벨에 새롭게 쓰고, new 라벨의 loop 명령어는 증가된 offset 값을 가지고 다음 주소에 똑같은 loop 명령어를 생성한다. 프로그램은 종료되지 않고, loop 레이블의 명령어가 무한히 증식하게 된다.

3-8) SMC8.C

```
start:
  ptr_body_reg10 = (unsigned char*)main + BODY;

body:
  for (i = 0; i < SIZE_OF_ADD; i++)
    reg8[i] = (ptr_body_reg10 + OFFSET_12)[i];

  for (i = 0; i < SIZE_OF_ADD; i++)
    reg9[i] = (ptr_body_reg10 + OFFSET_16)[i];

  for (i = 0; i < SIZE_OF_ADD; i++)
    (ptr_body_reg10 + OFFSET_12)[i] = reg9[i];

  reg2 += 21;
  reg2 += 21;

  for(i = 0; i < SIZE_OF_ADD; i++)
    (ptr_body_reg10 + OFFSET_16)[i] = reg8[i];

  dummy++;

  for(i = 0; i < SIZE_OF_SW; i++)
    reg9[i] = (ptr_body_reg10 + OFFSET_8)[i];

  for(i = 0; i < SIZE_OF_SW; i++)
    reg8[i] = (ptr_body_reg10 + OFFSET_20)[i];

  for(i = 0; i < SIZE_OF_SW; i++)
    (ptr_body_reg10 + OFFSET_20)[i] = reg9[i];

  for(i = 0; i < SIZE_OF_SW; i++)
    (ptr_body_reg10 + OFFSET_8)[i] = reg8[i];

  goto body;
```

Fig. 3.27 SMC8.C

여기에서 OFFSET8, OFFSET12 등등은 MIPS코드 상에서 명령어의 순서에 해당한다. MIPS에서는 한 개의 명령어가 4바이트로 이뤄지므로 OFFSET8 같은 경우에는 3번째 명령어, OFFSET12 같은 경우에는 4번째 명령어를 의미한다.

이후 프로그램이 진행되면서 실제 프로그램의 실행과 명령어의 순서가 바뀌는 명령어가 같이 진행된다.

3-9) SMC9.C

```
int main(void)
{
  /** .... */
  unsigned char *foo_code = (unsigned char *)malloc(sizeof(unsigned char) * 55);
  memcpy(foo_code, foo, 55);

  for (int i = 0; i < 55; i++)
  {
    foo_code[i] = foo_code[i] ^ -1;
  }

  memcpy(foo, foo_code, 55);

  for (int i = 0; i < 55; i++)
  {
    foo_code[i] = foo_code[i] ^ -1;
  }

  memcpy(foo, foo_code, 55);

  foo();
}

void foo()
{
        int num = 0;
        printf("This is Foo Function₩n");
        num += 10;
        printf("num = %d₩n", num);
}
```

Fig. 3.28 SMC9.C

foo_code 변수에 foo 함수의 코드를 저장한다. 해당 코드를 xor 연산으로 암호화해서 foo 함수의 주소에 쓴다. 이후에 xor 연산으로 복호화해서 foo 함수에

다시 쓴다. 예시를 보이기 위해서 암호화와 복호화를 동시에 진행했다. 이를 응용하면 미리 암호화된 코드 데이터를 복호화하여 쓰는 방식을 활용할 수 있다.

### 나. 포인터 분석 기반 정적 자체 수정 코드 탐지

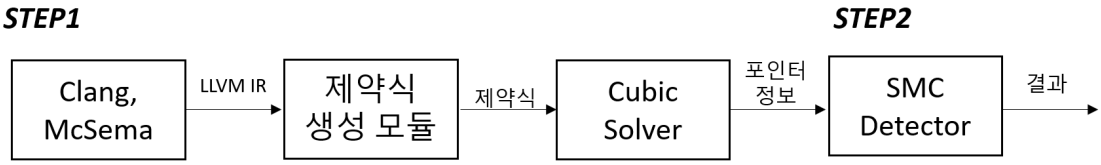



Fig. 3.29 SMC Analyzer Achitecture

자체 수정 코드는 Fig 1.1과 같이 메모리에 명령어의 값을 쓰게 된다. 컴퓨터 아키텍처에서 메모리의 값을 변경하는 명령어는 피연산자로 메모리의 주소 값을 가진다. 이러한 피연산자는 포인터로 자체 수정 코드에 사용된다. 따라서 포인터 변수에 대한 분석을 통해서 메모리 쓰기 명령어의 목적지가 데이터 영역이 아니라 코드 영역이라면 해당 코드는 자체 수정 코드라고 할 수 있다.

자체 수정 코드 탐지는 Fig 3.1과 같은 단계로 이루어 진다. 처음으로 Clang 컴파일러 또는 McSema 등을 통해서 분석할 LLVM IR을 준비한다. 준비한 LLVM IR은 이후 포인터 분석을 위해서 제약식을 생성한다. 준비된 제약식을 풀고 얻은 포인터 정보를 바탕으로 LLVM IR의 명령어를 분석하여 자체 수정 코드의 결과를 리포트한다.

1) 포인터 분석

1-1) 앤더슨 알고리즘

```
- X = alloc P:   alloc-i ∈ [[X]]

- X1 = & X2:   X2 ∈ [[X1]]

- X1 = X2:      [[X2]] ⊆ [[X1]]

- X1 = * X2:    For each c ∈ Cells,
                    c ∈ [[X2]]  =>  [[c]] ⊆ [[X1]]

- *X1 = X2:     For each c ∈ Cells,
                    c ∈ [[X1]]  =>  [[X2]] ⊆ [[c]]
```

Fig. 3.30 Andersen Pointer Analysis Constraints

각 포인터 변수는 [[X]] 와 같은 집합으로, 각 포인터 변수가 가질 수 있는 실제 메모리값은 X2와 같이 집합의 원소로 표현된다. 예를 들어 X=0x447과 같은 포인터 연산은 0x447∈[[X]] 과 같은 제약식을 생성한다. 생성된 제약식은 큐빅 알고리즘[6]을 통해서 최소해를 구하게 된다.

1-2) LLVM IR 제약식

Table. 3.2 LLVM IR Constraints

| INSTRUCTION | CONSTRAINT |
|---|---|
| ALLOCA | result ∈ [[result] |
| INTTOPTR | [[value]] ⊆ [[result]] |
| BITCAST | [[value]] ⊆ [[result]] |
| PHI | [[val1]] ⊆ [[result]],<br>[[val2]] ⊆ [[result]], ... |
| SELECT | [[val1]] ⊆ [[result]], [[val2]] ⊆ [[result]] |
| EXTRACTVALUE | [[val]] ⊆ [[result]] |
| STORE | c ∈ [[pointer]] => [[value]] ⊆ [[c]] |
| LOAD | c ∈ [[pointer]] => [[c]] ⊆ [[result]] |
| GETELEMENTPTR | c ∈ [[ptrval]] => [[c]] ⊆ [[result]] |
| CALL | p1, p2, ..., v1, v2 ..., ret ret_var ∧<br>[[v1]]⊆[[p1]],[[v2]]⊆[[p2]], ... ∧<br>[[ret_var]]⊆[[result]] |

Table. 3.3 LLVM IR Instructions

| INSTRUCTION | SYNTAX |
|---|---|
| ALLOCA | <result> = alloca <type> |
| INTTOPTR | <result> = inttoptr<br><ty> <value> to <ty2> |
| BITCAST | <result> = bitcast<br><ty> <value> to <ty2> |
| PHI | <result> = phi <ty><br>[ <val0>, <label0>], ... |
| SELECT | <result> = select selty<br><cond>, <ty> <val1>,<br><ty> <val2> |
| EXTRACTVALUE | <result> = extractvalue<br><aggregate type> <val>,<br><idx>{, <idx>}* |
| STORE | store [volatile] <ty> <value>, <ty>* <pointer> |
| LOAD | <result> = load [volatile] <ty>, <ty>* <pointer> |
| GETELEMENTPTR | <result> = getelementptr <ty>, <ty>* <ptrval> {, <ty> idx}* |
| CALL | <result> = call <ty> <fnty> <fnptrval><br>(<function args>) |

LLVM IR 에서는 Fig 3.2의 예제만이 아니라 더 많은 포인터와 관련된 명령어가 있다. 해당 명령어들에 대해서 앤더슨 알고리즘으로 해당 명령어들에 대한 적절한 제약식을 세워야 한다. 본 연구에서 제안하는 제약식은 Table 3.2와 같다. 각 명령어들의 문법은 Table 3.3과 같다. 각 명령어의 더 자세한 정보는 LLVM의 공식문서[something]를 통해서 확인할 수 있다.

본 연구에서는 제약식의 각 이름의 중복을 피하기 위해서 LLVM IR 의 계층구조를 이용하여 표현하였다. 예를 들어 main 함수의 지역변수 1 이라면 main!1 과 같이 표현된다. 전역변수 1 이라면 global!1 으로 표현된다.

1-2-1) Alloca Instruction

Alloca 명령어는 변수를 할당하는 명령어다. %ptr = alloca i32 의 명령어로 %ptr 변수에 i32 타입의 포인터 변수를 할당한다. 실제 메모리 주소 값을 변수 %ptr 에 할당하므로 그 제약식은 function!ptr ∈ [[function!ptr]] 와 같다.

1-2-2) Inttoptr Instruction

```
%alloca_ins1 = alloca i32
store i32 1, i32* %alloca_ins1
%load_ins1 = load i32, i32* %alloca_ins1
%ITP_ins1 = inttoptr i8 1 to i16*
%ITP_ins2 = inttoptr i32 %load_ins1 to i16*

alloca_ins1 ∈ [[alloca_ins1]]
[[ Constant-Value ]] ⊆ [[alloca-1]]
[[ Constant-Value ]] ⊆ [[load_ins1]], [[alloca-1]] ⊆ [[load_ins1]]
[[ Constant-Value ]] ⊆ [ITP_ins1]
[[load_ins1]] ⊆ [[ITP_ins2]]
```

Fig. 3.31 INTTOPTR Instruction Constraint

Inttoptr 명령어는 Integer 타입의 value를 포인터 타입으로 result에 할당하는 명령어다. 포인터 타입의 변수 [[result]]에 변수 [[value]]을 할당하는 것과 같다. 그 예는 Fig 3.12와 같다.

1-2-3) Bitcast Instruction

Bitcast 명령어는 값의 변경 없이 변수의 타입만을 변경한다. 따라서 [[value]] ⊆ [[result]] 의 제약식을 생성한다.

1-2-4) Phi Instruction

```
example:
  br label loop
loop:
  %indvar = phi i32 [0, %example ], [ %nextindvar, %loop]
  %nextindvar = add i32 %indvar, 1
  ...

[[ Constant-Value ]] ⊆ [[ indvar ]]
[[ nextindvar ]] ⊆ [[ indvar ]]
```

Fig. 3.32 Phi Instruction Constraint

Phi Instruction은 이전 BasicBlock에 따라 선택적으로 result에 value를 할당하는 명령어다. 견고한 제약식 생성을 위해서 할당될 수 있는 모든 value에 대하여 제약식을 생성한다. 그 예는 Fig 3.13과 같다.

1-2-5) SELECT INSTRUCTION

```
%select_ins1 = select i1 condition1, i64* %alloca_ins4, i64* %alloca_ins5
%select_ins2 = select i1 condition2, i64 1234, i64 123

[[ alloca_ins4 ]] ⊆ [[ select_ins1 ]]
[[ alloca_ins5 ]] ⊆ [[ select_ins1 ]]
[[ Constant-Value ]] ⊆ [[ select_ins2 ]]
[[ Constant-Value ]] ⊆ [[ select_ins2 ]]
```

Fig. 3.33 SELECT Instruction Constraint

select 명령어는 조건 값에 따라 result에 value를 할당한다. 해당 명령어 또한 phi 명령어와 같이 견고한 제약식 생성을 위해서 할당될 수 있는 모든 value에 대하여 제약식을 생성한다. 그 예는 Fig 3.14와 같다.

1-2-6) Extractvalue Instruction

```
%extractvalue_ins1 = extractvalue %struct1 %load_ins1, 0
%extractvalue_ins2 = extractvalue %struct2 %load_ins2, 0, 1

[[ load_ins1 ]] ⊆ [[ extractvalue_ins1 ]]
[[ load_ins2 ]] ⊆ [[ extractvalue_ins2 ]]
```

Fig. 3.34 Extractvalue Instruction Constraint

Extractvalue 명령어는 주어진 val로부터 특정 element를 result에 value를 할당하는 명령어다. 집합에서 특정 원소를 가져오는 것과 같은 명령어다. 견고한 제약식 생성을 위해서 각 원소에 대한 제약식 대신에, 집합 전체에 대하여 제약식을 생성한다. 그 구체적인 예는 Fig 3.15와 같다.

1-2-7) Store Instruction

```
%alloca_ins1 = alloca i32
store i32 16, i32* %alloca_ins1
%load_ins1 = load i32, i32* %alloca_ins1
store i32 %load_ins1, i32* %alloca_ins1

alloca_ins1 ∈ [[alloca_ins1]]
[[ Constant value ]] ⊆ [[alloca_ins1]]
for each c in [[ alloca_ins1 ]], [[ c ]] ⊆ [[ load_ins1 ]]
for each c in [[ alloca_ins1 ]], [[ load_ins1 ]] ⊆ [[ c ]]
```

Fig. 3.35 STORE Instruction Constraint

Store 명령어는 value의 값을 pointer에 저장하는 명령어로 *X=X 와 같다. 구체적인 예는 Fig 3.16과 같다.

1-2-8 Load Instruction

```
%alloca_ins1 = alloca i32
%load_ins1 = load i32, i32* %alloca_ins1

alloca_ins1 ∈ [[ alloca_ins1 ]]
for each ∈ in [[ alloca_ins1 ]], [[ c ]] ⊆ [[ load_ins1 ]]
```

Fig. 3.36 Load Instruction Constraint

Load 명령어는 pointer에 value를 불러오는 명령어로 X=*X 와 같다. 그 구체적인 예는 Fig 3.17과 같다.

1-2-9 Getelementptr Instruction

```
%struct1 = type { i1, i8, i16, i32, i64 }
%alloca_struct1 = alloca %struct1
%GEP_ins1 = %getelementptr %struct1, %struct1* %alloca_struct1, i32 0, i32
0

alloca_struct1 ∈ [[ alloca_struct1 ]]
alloca_struct1 ∈ [[GEP_ins1]]
for each c ∈ [[ alloca_struct1 ]] => [[ c ]] ⊆ [[ GEP_ins1 ]]
```

Fig. 3.37 Getelementptr Instruction Constraint

Getelementptr 명령어는 ptrval로부터 특정 원소를를 가리키는 pointer를 result에 할당한다. ptrval을 기준으로하는 포인터 변수의 값들이 result에 할당될 수 있으므로 <result> = getelementptr <ty>, <ty>* <ptrval> {, <ty> idx}* 의 표현식에서 c ∈ [[ptrval]] => [[c]] ⊆ [[result]] 제약식을 만든다. 그 구체적인 예는 Fig 3.18과 같다.

1-2-10 Call Instruction

```
define dso_local i32 @foobar(i32 %foo, i32 %bar) {
  entry:
    %alloca_ins1 = alloca i32
    store i32 %foo, i32* %alloca_ins1
    %add_ins1 = add i32 %foo, %bar
    %load_ins1 = load i32, i32* %alloca_ins1
    ret i32 %load_ins1
}

define dso_local i32 @main(){
  entry:
  %alloca_ins1 = alloca i32
  %alloca_ins2 = alloca i32
  %load_ins1 = load i32, i32* %alloca_ins1
  %load_ins2 = load i32, i32* %alloca_ins2
  %call_ins1 = call i32 @foobar(i32 %load_ins1, i32 %load_ins2)
  ret i32 0
}

[[ main!load_ins1 ]] ⊆ [[ foobar!foo ]]
[[ main!load_ins2 ]] ⊆ [[ foobar!bar ]]
[[ foobar!load_ins1 ]] ⊆ [[ main!call_ins1 ]]
```

Fig. 3.38 Call Instruction Constraint

Call 명령어는 함수 호출 명령어다. 함수 호출은 앤더슨 알고리즘에 따라서 형식 매개변수와 실질 매개변수의 제약식은 [[ val1 ]] ⊆ [[ param1 ]], [[ val2 ]] ⊆ [[ param2 ]] ... [[ valN ]] ⊆ [[ paramN ]] 와 같다. 함수가 반환하는 데이터에 대해서는 [[ return_variable ]] ⊆ [[ result ]] 와 같다. 그 구체적인 예는 3.19와 같다.

2) 자체 수정 코드 검출

검출기는 포인터 분석을 통해서 포인터 변수에 대한 정보를 바탕으로 자체 수정 코드를 검출한다. 메모리 쓰기 명령어의 오퍼랜드로 사용된 포인터 변수가 코드 영역을 가리킬 수 있다면, 해당 메모리 쓰기 명령어는 자체 수정 코드로 분류하여 그 결과를 리포트 한다.

### 다. 프로그램 구현

검출기 프로그램은 Python3와 C++으로 작성되었다. LLVM을 비롯한 코어 부분은 C++으로 작성되었고, 외부함수(Foreign Function Interface) 형태로 구현하였다. LLVM IR 파일과 바이너리 프로그램을 입력으로 받아 제약식, 포인터 변수, 검출 결과의 세가지 파일을 생성한다.

1) 제약식

1-1) 변수 이름 지정

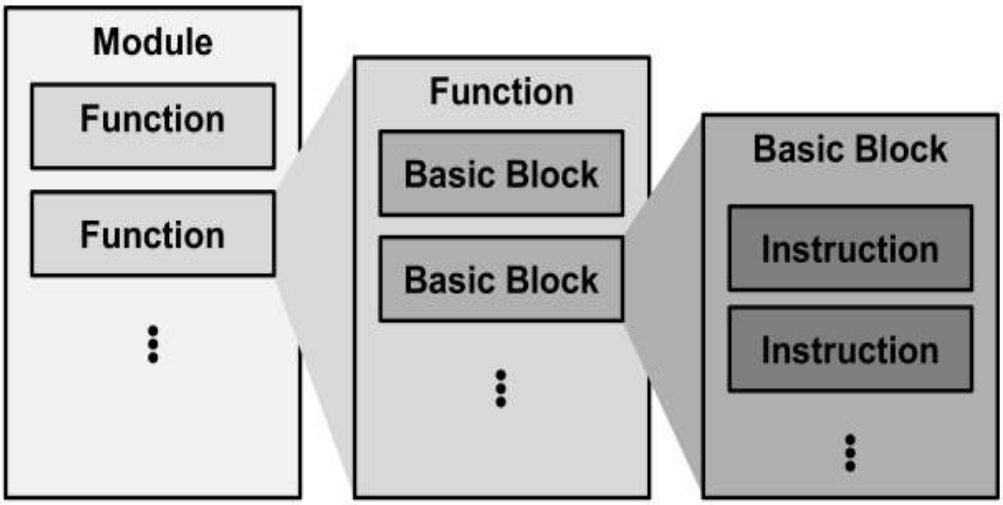



Fig. 3.39 LLVM IR Strcture[10]

LLVM IR은 그림과 같은 구조를 가진다. 한 개의 LLVM IR 파일은 여러개의 Function 으로 이루어져 있고, 그 아래에 명령어들이 구성되어 있다. 정적 분석은 파일 전체에 시행되기 때문에 함수 아래의 변수들의 경우 이름이 중복될 수 있다. 이를 피하기 위해서 변수에는 네임스페이스를 도입했다. main 함수 아래의 %1 변수는 main!1, 전역변수 @foo는 global!foo와 같다.

1-2) 제약식 생성

Table. 3.4 Constraint Format in Implementation

| Type | Operand A | Operand B | Expression |
|---|---|---|---|
| 0 | A | - | A ∈ [[A]] |

| | | | |
|---|---|---|---|
| 1 | A | B | A ∈ [[B]] |
| 2 | A | B | [[A]] ⊆ [[B]] |
| 3 | A | B | ∀a ∈ [[A] →[[a]] ⊆ [[B]] |
| 4 | A | B | ∀a ∈ [[A]] →[[B]] ⊆ [[a]] |
| 5 | A | - | '!code!' ∈ [[A]] |
| 6 | A | - | '!data!' ∈ [[A]] |

LLVM IR 파일을 읽어 들여서 테이블 3.4 Table 3.4와 같은 규칙에 따라서 제약식을 생성한다. A ∈ [[A]] 제약식은 [0, “A”] 이라는 값으로 표현된다. 이 값을 가지고 최소 해를 구하게 된다. 결과적으로 A=[A]와 같이 포인터 변수와 그 포인터 변수가 가질 수 있는 메모리 주소인 토큰의 값이 결과로 나타난다.

2) 자체 수정 코드 검출

```
{
    "binaryFile": "/home/swlab/smc/test/binary/gcc_m64_PIE_smc1.out",
    ....
    "detect": [
        {
            "Function": "sub_7ea_main",
            "Block": "inst_7ea",
            "Order": 23,
            "LLVM Instruction": "store",
            "Pattern": 1.2,
            "CriticalOperand": "sub_7ea_main!stack!-24",
            "Str":"store i64 2133, i64* %₩"sub_7ea_main!11₩", align 8",
            "Tokens": "!code!"
        },
        ....
    ]
    ....
}
```

Fig. 3.40 SMC Report Example

자체 수정 검출 프로그램은 각 LLVM IR 명령어를 순회하면서 해당 명령어가

메모리의 값을 변경할 때, 피연산자로 사용되는 포인터 변수에 코드 실행 영역이 포함되어 있으면 이를 자체 수정 코드로 검출한다. 그 검출 결과는 Fig 3.21과 같다.

# 4. 연구 결과

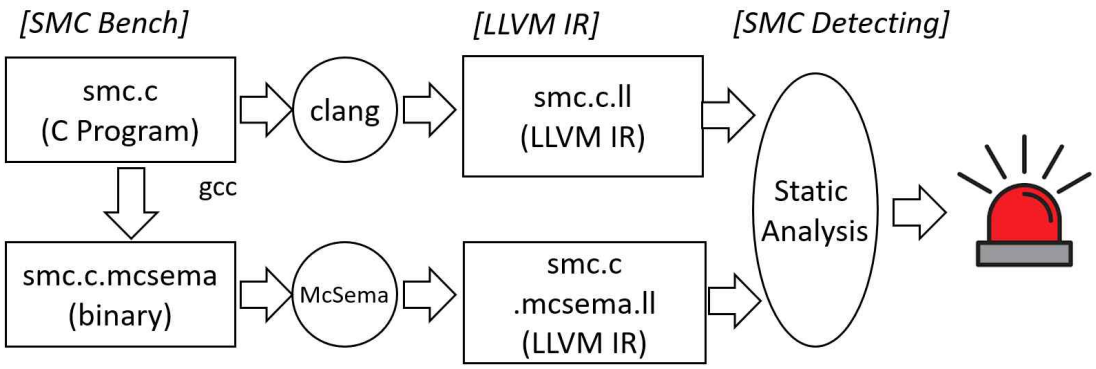



Fig. 4.1 SMC Static Analysis

SMC Bench의 9가지 자체 수정 코드에 대해서 검출기를 적용하고 그 결과를 리포트 한다. X86 벤치마크의 경우 McSema을 통해서 리프팅을 할 수 없었다. IDAPro을 이용하여 CFG 파일을 생성하게 되는데, McSema 의 경우 저수준의 어셈블러를 사용해서 임의로 작성했을 때 바이너리에 대한 정보가 적어서 자동으로 CFG 파일을 생성할 수 없었다.

### 가. C 언어 기반 벤치마크

1) Clang

Table. 4.1 Clang LLVM IR Analysis Result

| smc.c.ll | Answer | Result | Instructions |
|---|---|---|---|
| smc1 | O | O | 93 |
| smc2 | X | X | 178 |
| smc3 | O | O | 202 |
| smc4 | O | O | 136 |
| smc5 | O | O | 214 |
| smc6 | O | O | 179 |
| smc7 | O | O | 80 |
| smc8 | O | O | 232 |
| smc9 | O | O | 112 |

Clang을 통해서 생성된 LLVM IR의 경우 생성된 명령어의 개수도 적고 코드 주소 패턴이 명확하여 정적 분석을 하는데 용이하였다. Fig 4.1과 같은 LLVM IR이 생성되는데, store 명령어의 오퍼랜드로 @main 이라는 함수를 가리키는 변수가 바로 사용됨을 확인할 수 있다.

```
define dso_local i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca i8*, align 8

  ; ...
  store i8* getelementptr inbounds (i8, i8* bitcast (i32 ()* @main to
i8*), i64 107), i8** %2, align 8
  ; ...

  ret i32 0
}
```

Fig. 4.2 SMC1.c.ll

2) McSema

Table. 4.2 McSema LLVM IR Analysis Result

| smc.c.mcsema.ll | Answer | 1st Result | 2nd Result | Instructions |
|---|---|---|---|---|
| smc1 | O | X | O | 2009 |
| smc2 | X | X | O | 2568 |
| smc3 | O | X | O | 2071 |
| smc4 | O | X | O | 2167 |
| smc5 | O | X | O | 2122 |
| smc6 | O | X | O | 1965 |
| smc7 | O | X | O | 1830 |
| smc8 | O | X | O | 2018 |
| smc9 | O | X | O | 2309 |

C언어로 작성된 바이너리 프로그램을 McSema으로 리프팅한 LLVM IR의 경우 첫 시도에서 자체 수정 코드를 검출하는데 어려움이 있었고, McSema의 특징을 분석기에 적용한 2차 시도의 경우에는 자체 수정 코드를 검출할 수 있었다.

Clang을 통해서 LLVM IR을 생성할 때와 다르게, McSema을 사용하여 LLVM IR으로 리프팅하게 되는 경우 @main 과 같은 변수가 직접적으로 드러나지 않는다.

Fig 4.2에서는 오퍼랜드로 @main이 나타나지만, Fig 4.3 에서는 직접적으로 나타나지 않는다.

```
%0 = load i64, i64* @RBP_2328_55db47a2f1c8, align 8
%1 = load i64, i64* @RSP_2312_5613298471c8, align 8, !tbaa !1220
%10 = add i64 %1, -32
%11 = inttoptr i64 %10 to i64*
store i64 2133, i64* %11, align 8
....
%15 = load i64, i64* @RBP_2328_5613298471c8, align 8
%18 = add i64 %15, -24
%19 = inttoptr i64 %18 to i32**
%98 = load i32*, i32** %19, align 8
store i32 %97, i32* %98, align 4
```

Fig. 4.3 SMC1.c.mcsema.ll

```
801:    48 8d 05 4d 00 00 00   lea 0x4d(%rip),%rax #855<main+0x6b>
808:    48 89 45 e8            mov    %rax,-0x18(%rbp)
....
85c:    48 8b 45 e8            mov    -0x18(%rbp),%rax
860:    89 10                  mov    %edx,(%rax)
```

Fig. 4.4 SMC1.C Binary Code

Fig 4.4 에서와 같이 @main + 107의 바이너리상에서의 값은 0x855(2133)이다. Fig 4.3에서 2133의 값은 %11에 저장되고, %11은 %1은 @RSP-32(@RBP-24) 의 주소값이다. 즉. @main 과 같은 변수형태가 아니라 레지스터를 사용하는 바이너리상에서 명령어 그대로 리프팅되어 나타난다.

리프팅된 LLVM IR을 분석하는데 문제가 되는 것은 Clang을 통해 생성된 LLVM IR과는 다르게 리프팅 과정에서 문맥 정보의 손실이 있을 뿐 아니라, @RBP와 같이 하드웨어 변수를 사용하기 때문이다.

2차 분석에서는 이런 특징을 반영했다. %15 = load i64, i64* @RBP 명령어와 %18 = add i64 %15, -24 와 같은 명령어가 있을 때 %18을 [[function!stack!-24]] 으로 표현하여 좀 더 정밀하게 분석하였다. Table 4.2에서와 같이 긍정 오류(False Positive)가 있기는 했지만 자체 수정 코드를 검출할 수 있었다.

# 5. 결론 및 향후 연구

### 가. 결론

자체 수정 코드는 메모리의 값을 수정하여 악성코드를 숨기는 데 악용된다. 본 연구는 이러한 자체 수정 코드에 대해서 전통적인 포인터 정적 분석 방법인 앤더슨 알고리즘을 LLVM IR에 적용하여 자체 수정 코드 검출에 대한 가능성을 실험했다.

또 MIPS로 작성된 자체 수정 코드를 보편적으로 사용되는 x86 코드와 C언어로 재작성하여 SMC Bench 벤치마크 프로그램을 구성했다. 해당 벤치마크를 이

용하면 본 연구의 정적 분석기의 개발과 테스트뿐 아니라, 다른 자체 수정 코드의 연구에도 효과적으로 활용 할 수 있을 것으로 기대된다.

SMC Bench에서 Clang과 McSema을 통해서 LLVM IR을 생성했을 때, Clang을 통해서 C 언어 벤치마크에 대해서 효과적으로 앤더슨 알고리즘이 작동하여 자체 수정 코드를 검출할 수 있었다. 그에 반하여 McSema을 통해서 만들어진 C 언어 밴치마크의 LLVM IR은 McSema에 의해서 리프팅되는 바이너리의 특징에 의해서 앤더슨 알고리즘의 정확도가 떨어지게 되었다. 이를 고려하여 바이너리의 Stack에 저장되는 일련의 특징을 반영한 결과 기존보다 더 개선된 결과를 얻을 수 있었다.

LLVM 프레임워크 위에서의 정적 분석을 통해 보안 분야에 있어서 LLVM 프레임워크가 충분히 활용될 수 있는 가능성을 확인했다.

### 나. 향후 연구

1) 바이너리 분석 플랫폼

대부분의 많은 바이너리 분석과 보안에 대한 연구는 특정 분석 도구와 하드웨어에 종속적이기 때문에 파편화 되어 있어 그 성과가 공유되는데 어려움이 따른다.

LLVM 프레임워크가 LLVM IR을 생성하는 프론트엔드와 이를 이용하는 백엔드로 나뉘어져 있는 것처럼 바이너리의 분석 또한 바이너리로부터 LLVM IR을 생성하는 리프터와 그를 활용한 바이너리 분석 프로그램을 작성한다면 바이너리 보안에 대한 많은 성과가 LLVM 프레임워크 상에서 재사용되고 공유될 수 있을 것이다.

2) 리프터 개선

McSema는 바이너리 코드를 LLVM IR 으로 리프팅하기 위해서 두 단계를 거친다. 첫 단계로 디스 어셈블러를 이용해 바이너리 코드로부터 CFG 파일을 만든다. 다음 단계로 어셈블리 명령어에 대한 시멘틱 함수를 제공하는 Remill을 사용하여 LLVM IR을 생성한다.

X86의 NASM 어셈블러로 작성된 바이너리 프로그램의 경우 McSema는 리프

팅할 수 없었다. 프로그램의 시작과 끝과 같은 정보들을 알 수 없었기 때문이다. 프로그램의 소스코드와 다르게 바이너리 프로그램의 경우에는 변수의 정보를 포함하여 많은 정보들이 사라지기 때문에 리프팅을 하는 것은 쉬운 일이 아니다.

따라서 McSema의 CFG와 Remill을 이용하여 인터렉티브한 리프터를 설계하여 바이너리에 대하여 상호작용해가며 프로그램에 대한 정보를 추가하여 부분적으로 리프팅을 할 수 있다면 바이너리 분석가 또한 적극적으로 LLVM IR 기반의 분석을 시도해볼 수 있을 것이다.

3) 휴리스틱 개선

본 연구에서 자체 수정 코드로 추정하는 휴리스틱은 실제로 해당 코드가 실행되는 것의 여부와 관계없이 데이터가 쓰여지는 메모리에 실행권한이 있으면 자체 수정 코드로 추정한다.

만약 데이터를 저장하는 메모리에 실행권한이 있으면 이를 자체 수정 코드로 추정하게 된다. 따라서 좀 더 정확한 분석을 위해서는 원론적으로 정말로 해당 메모리가 실행될 수 있는지 확인해야 한다. 따라서 기존의 실행권한이 있는 메모리에 더해서 프로그램의 진입점을 포함한 호출되는 모든 포인터 변수의 영역만을 더 세밀하게 특정하는 방법을 통해서 더 나은 결과를 얻을 수 있을 것으로 기대된다.

# 참고 문헌


[1] H. Cai, Z. Shao, and A. Vaynberg. 2007. Certified self-modifying code. In Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07). Association for Computing Machinery, New York, NY, USA, 66-77.

[2] Wu, Y., Chiueh, Tc., Zhao, C. (2009). Efficient and Automatic Instrumentation for Packed Binaries. In: Park, J.H., Chen, HH., Atiquzzaman, M., Lee, C., Kim, Th., Yeo, SS. (eds) Advances in Information Security and Assurance. ISA 2009. Lecture Notes in Computer Science, vol 5576. Springer, Berlin, Heidelberg.

[3] https://github.com/packing-box/peid

[4] Arora, R., Singh, A., Pareek, H., & Edara, U. R. (2013). A heuristics-based static analysis approach for detecting packed pe binaries. International Journal of Security and Its Applications, 7(5), 257-268.

[5] P. V. Shijo and A. Salim, "Integrated static and dynamic analysis for malware detection," in Procedia Computer Science, 2015, vol. 46, pp. 804-811.

[6] RNDr. Petr Rockai, "Decompiling Binaries Into LLVM IR Using Mcsema And Dynist", Masaryk University, Faculty of Informatics, 2019

[7] H. Cai, Z. Shao, and A. Vaynberg, "Certified selfmodifying code", ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: ACM, pp. 66-77. 2007.

[8] K. Vollmar and P. Sanderson, "MARS: An Education-Oriented MIPS Assembly Language Simulator," SIGCSE 2006 paper, ACM SIGCSE Bulletin, 38:1 (March 2006), 239-243.

[9] B. Hardekopf and C. Lin, "The Ant and the Grasshopper: Fast and

accurate pointer analysis for millions of lines of code", Programming Language Design and Implementation (PLDI), pp. 290-299, 2007.

[10] R. Tschuter, "An LLVM Instrumentation Plug-in for Score-P", LLVM-HPC'17, November 12-17, 2017.

# LLVM framework-based implementation and experimentation of static analysis method to detect self-modifyinh code

Jae IL Yu

Department of Applied Artificial Intelligence,
Graduate School, Chonnam National University
(Supervised by Professor Kwang Hoon Choi)

(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.

# 부록 A. 자체 수정 코드 검출기

## 가. SMC Bench

https://github.com/dbwodlf3/SMC/tree/master/src/smcbench/smc 저장소에서 소스 코드를 찾아볼 수 있다. 각 폴더에 mips, x86, c 언어로 작성된 자체 수정 코드의 파일과 설명이 들어있다.

run_x86.py 스크립트 파일을 통해서 빠르게 바이너리 프로그램을 생성할 수 있다. python3 run_x86.py smc1/smc1.x86.s을 실행하면 x86 자체 수정 코드에 대해서 자체 수정 코드에 알맞은 바이너리 포맷을 만들어 준다.

## 나. 자체 수정 코드 검출기

https://github.com/dbwodlf3/SMC/tree/master/src/llvmsmc 저장소에서 소스 코드를 찾아볼 수 있다.

자체 수정 코드 검출기를 실행하기 위해서는 LLVM 11 버전과 Python3의 llvmlite 라이브러리가 설치되어 있어야 한다. C++으로 작성된 LLVM 프로그램을 컴파일 하기 위해서 Cmake가 필요하다. cmake —build . 명령어를 통해서 관련 코드를 dll 으로 컴파일 할 수 있다.

test.py 스크립트 파일을 통해서 빠르게 제약식, 포인터 변수, 자체 수정 코드 검출의 결과를 진행할 수 있다. 입력이 되는 바이너리 파일과 LLVM IR 파일은 저장소의 test 디렉토리 아래에 있다. python3 test.py을 실행하면 각각의 결과 파일은 장소의 dest 폴더 아래에 저장된다.

# 부록 B. 큐빅 알고리즘

큐빅 알고리즘은 집합 제약식을 입력으로 받아 해를 구해 출력한다. 많은 정적 프로그램 분석 방법의 해를 구하는 방법으로 사용하고 있어 큐빅 프레임워크라 불리기도 한다. 큐빅 알고리즘은 두 가지 유형의 집합에 대해 정의된 세 가지 형태의 제약식을 다룬다.

```
- 유한개의 토큰 집합 T = {t1, ..., tk}
- 유한개의 변수 집합 V = {v1, ..., vn}
```

토큰으로 집합에 포함 가능한 원소를 표현하고 변수는 토큰들의 집합을 표현한다. 각 변수들로 표현하는 집합은 전체 토큰 집합 T의 부분집합이다. 앤더슨 알고리즘에서 토큰은 변수 x, y, z (포인터 변수 포함), 메모리 할당 alloc-i, 함수 f, g가 된다.

```
- 제약식 C := t ∈ v      (토큰 t는 변수 v의 집합의 원소)
          | v1 ⊆ v2     (변수 v1의 모든 토큰 원소는 변수 v2의 원소)
          | t ∈ v1 => v2 ⊆ v3
                    (토큰 t가 x의 원소이면 y의 모든 원소는 z의 원소)
```

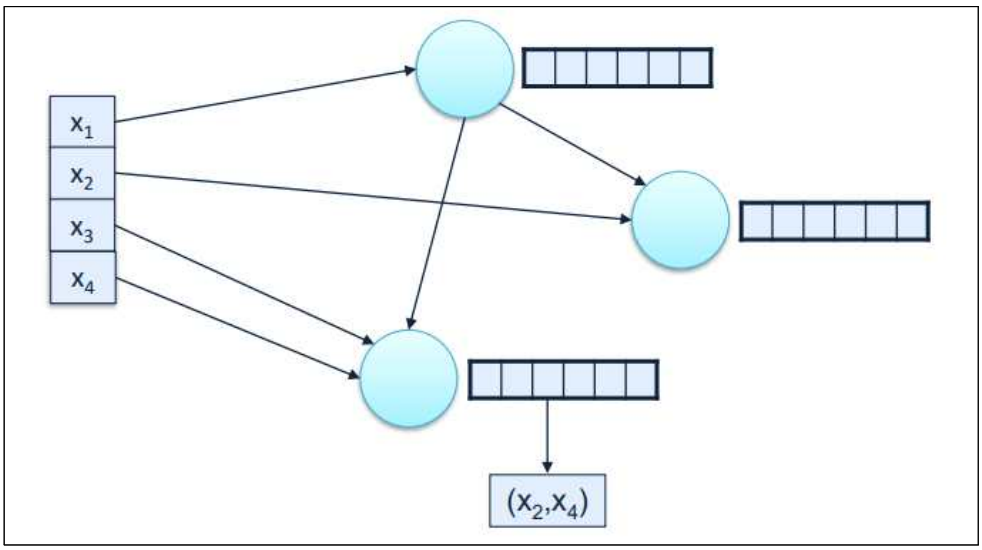
첫 번째 유형의 제약식은 변수 집합에 토큰을 추가하고, 두 번째와 세 번째 유형의 제약식에서 여러 변수 집합으로 이 정보를 전파한다. 두 번째 제약식은 조건 없이 정보를 전파하고 세 번째 제약식은 조건부로 정보를 전파한다. 큐빅 알고리즘은 이러한 형태의 제약식을 받아 변수들의 최소 해 (minimal solution)를 구한다.

큐빅 알고리즘은 제약식 집합을 입력으로 받아 각 변수 집합의 해를 구해 출력한다. 이 알고리즘은 DAG (directed acyclic graph)를 사용하여 제약식 간의 의존 관계를 표현한다.

DAG의 노드와 에지는 다음과 같이 구성한다.

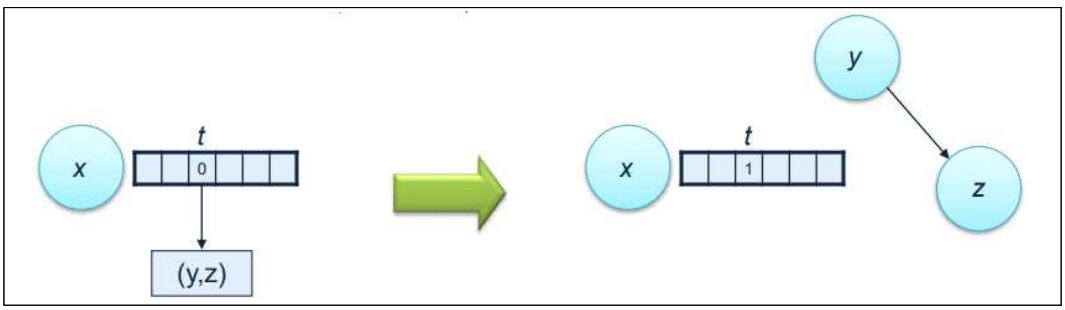
- 각 변수마다 노드를 둔다. n개의 변수를 사용하면 n개의 노드를 생성한다.
- 각 노드마다 비트 벡터 (bitvector) 속성을 두어 k개 토큰을 0과 1로 표시한다.
- 두 노드 v1과 v2의 방향성 에지 (directed edge)로 두 변수 집합들의 포함 관계를 나타낸다.

- 각 비트마다 변수 쌍 (v1, v2) 리스트를 두어 해당 비트가 1이면 v1의 모든 토큰들을 v2로 모두 전달한다.



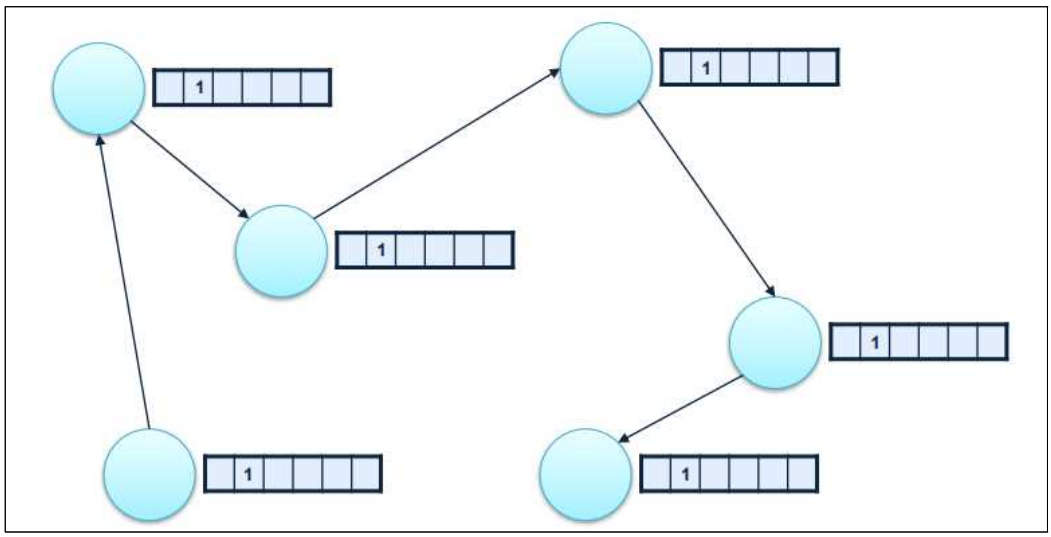

큐빅 알고리즘의 절차는 제약식을 DAG에 하나씩 추가하는 방식으로 구성되어 있다. 첫 번째 형태의 제약식 ti ∈ xj 조건을 확인하려면 변수 xj에 해당하는 j번째 노드를 찾아서 그 노드에 딸린 비트 벡터에서 토큰 ti에 해당하는 i번째 비트를 본다. 이 비트가 1이면 해당 조건이 참이다.

아래 그림과 같이 x에 해당하는 노드를 찾고, 이 노드의 비트 벡터에 1을 설정한다. 만일 이 비트 벡터에 매달린 리스트에 y ⊆ z가 포함되어 있으면 y에서 z로 연결하는 에지를 새로 연결하여 y ⊆ z 제약식을 추가한다. 재귀적으로 두 번째 제약식을 추가할 때의 큐빅 알고리즘을 실행한다.

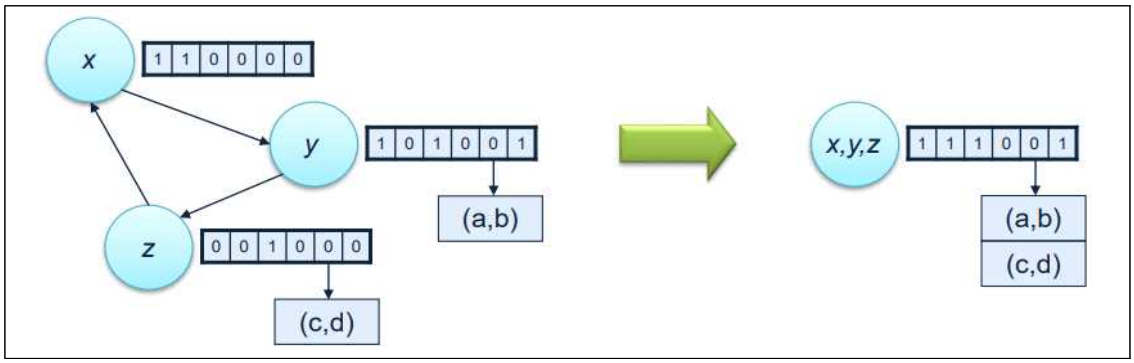



두 번째 형태의 제약식 y ⊆ z을 추가하면, y의 비트 벡터에 1로 설정된 모든 토큰을 z의 비트 벡터에 전파한다. 각 노드에 새로운 토큰 비트를 전파할 때마

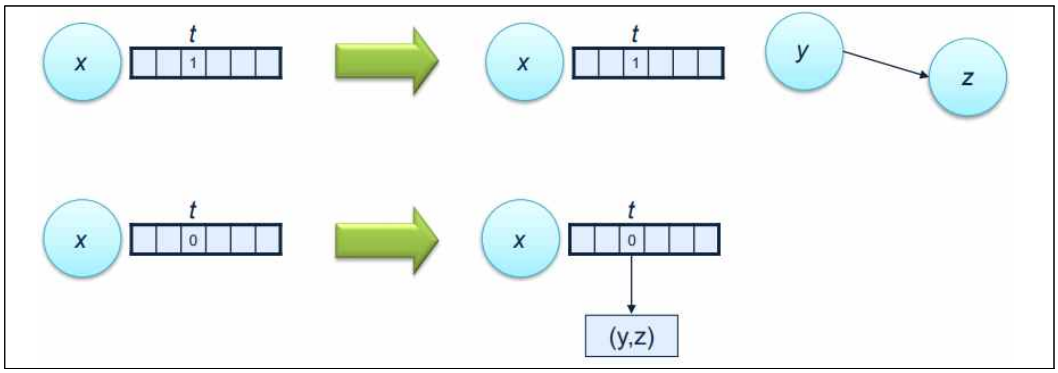
다 재귀적으로 tk ∈ z 제약식을 추가할 때의 큐빅 알고리즘을 실행한다.




제약식 y ⊆ z을 추가하면, y의 비트 벡터에 1로 설정된 모든 토제약식 y ⊆ z 에지를 DAG에 추가할 때 새로 사이클이 형성될 수 있다. 이 경우 토큰 비트 벡터를 전파하는 과정이 무한히 반복될 수 있다. 따라서 이 때는 이 사이클을 구성하는 모든 노드들을 하나의 노드로 합한다. 노드의 비트벡터 속성과 비트별 속성들도 합한다. 이 노드들이 가지고 있던 비트 벡터들을 모두 OR시키고 동일한 비트에 연결된 변수 쌍 리스트들도 하나의 리스트로 합한다.




세 번째 형태의 제약식 ti ∈ x => y ⊆ z은 다음과 같이 추가한다. 노드 x의 비트 벡터에 ti 비트는 0 또는 1이다. 만일 ti 비트가 1이면 y ⊆ z 제약식을 추가한다. 재귀적으로 두 번째 형태의 제약식을 추가할 때의 큐빅 알고리즘을 실행한다.




큐빅 알고리즘은 입력 받은 프로그램에 대한 변수 집합과 토큰 집합을 구해 초기 DAG을 만들고, 각 제약식을 설명한 방식대로 DAG에 추가한 결과의 DAG에서 제약식의 해를 구해 출력한다.