

바이너리 대상 자체수정코드 식별 기법 연구

(Self-Modifying Code, SMC)

2020년 10월 29일

최 광 훈, 유 재 일, 김 상 업
김사연, 박재현, 이안나

전남대학교 전자컴퓨터공학부

목차

- 연구개발 목적
- 연구 내용 및 범위
- 연구 결과
- 활용에 대한 건의

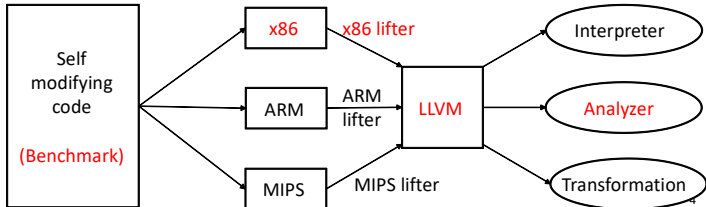
1. 연구 개발 목적

- 목적: HW 비종속 바이너리의 자체수정 탐지
- 중요성:
 - 악성 바이너리가 자신을 숨기는 전형적인 기법
 - 아키텍처와 무관한 식별 기법으로 분석 비용을 줄임
- 배경:
 - LLVM 컴파일러 인프라스트럭처 (오픈소스SW)

2. 연구 내용 및 범위

- 자체수정 벤치마크 프로그램
- 바이너리를 LLVM IR로 리프팅하는 방법 조사
- 자체수정 코드 검출 방법

[연구 범위(붉은색)]



2-1. 자체수정 벤치마크 프로그램

- 자체수정코드(Self-Modifying code, SMC)
 - 실행 중에 자신의 명령어를 읽고 쓰는 코드
 - 용도: 실행시간 정보를 활용한 최적화, 프로그램 저작권 보호, 악성코드가 스스로를 숨김

```
.data # Data declaration section

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

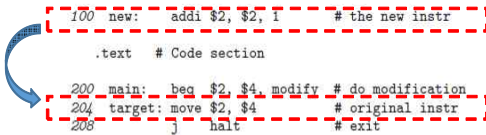
.text # Code section

200 main: beg  $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
```

Self-Modifying Code ←

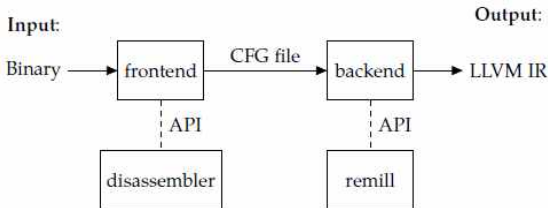


2-1. 자체수정 벤치마크 프로그램(계속)

- 일반적인 이유
 - SMC에 대한 기존 연구는 다수 존재
 - 이러한 연구를 비교할 수 있는 벤치마크가 없음
- 연구관련 구체적인 이유
 - SMC 탐지를 위한 패턴 조사
 - SMC 탐지 방법의 평가
 - (MIPS/x86/ ARM 아키텍처 별 SMC 코드 비교)

2-2. x86-LLVM IR 리프터: McSema

- 리눅스 x86 바이너리를 LLVM IR로 역컴파일
- 아키텍처:
 - 프론트엔드: 디스어셈블러 (E.g., IdaPro)를 활용하여 x86 바이너리 정보 추출
 - 백엔드: LLVM 비트코드 생성
 - CFG: 바이너리의 모든 정보



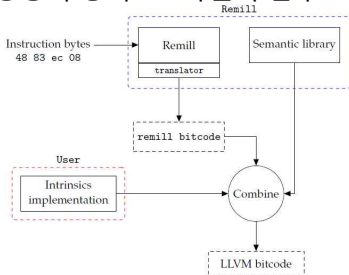
2-2. McSema

- CFG 파일 구성

- 레퍼런스, 세그먼트, 함수, 블록과 명령어, 외부 심볼
=> (다음 슬라이드)

- 명령어 시뮬레이션 방식

- 레지스터 셋 => State 구조체
- 개별 명령어 동작 => 시맨틱 함수로 변환(Remill)



2-2. McSema: CFG 파일 형식

- ```
<Module> := { <Segment> | <External Variable> | <External Function> |
 <Function> | <Global Variable> };

<Segment> := address, name, data, is_read_only, is_thread_local,
 { <Data Reference> };
<Data Reference> := address, width,
 target_address, target_name, target_is_code;

<External Variable> := name, address, size, is_weak, is_thread_local
<External Function> := <Calling Convention>, name, address,
 no_return, argument_count, is_weak;

<Function> := address, { <Block> }, is_entrypoint, [name],
 [<Stack Variable>];
<Block> := address, { <Instruction> }, { successor_addresses };
<Instruction> := address, bytes, { <Code Reference> }, [local_noreturn];
<Code Reference> := <Location>, <Operand Type>, address;
<Location> := "internal" | "external";
<Operand Type> := "immediate" | "memory" | "displacement" |
 "control flow" | "offset table";

<Stack Variable> := names, size, sp_offset,
 { <Instruction Reference> };
<Instruction Reference> := instruction_address, offset;
<Global Variable> := address, name, size;
```

## 2-2. McSema 리프팅 코드 생성

- **McSema의 리프팅 방법1**: 명령어 시뮬레이션!
  - 각 명령어 → Remill의 semantic function
  - x86 sub 명령어의 semantic function (C++ 템플릿)

```
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 subtraction
 WriteZExt(dst, sum); // Write into dst
 // zero extend if types do not match
 WriteFlagsAddSub<tag_sub>(state, lhs, rhs, sum); // Update ArithFlags
 return memory;
}
```

- 1개 명령어 리프팅: **48 83 ec 20 sub \$0x20, %rsp**

```
%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)
```

- x86 foo 함수를 리프팅하기

```
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
```



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

## 2-2. McSema 리프팅 코드 생성(계속)

- **McSema의 리프팅 방법2**: 각 함수 몸체 동작을 시뮬레이션하는 함수와 문맥 전환 함수로 변환

- 예) main 함수 => @sub\_주소\_main, @main

main 몸체의 명령어들을 시뮬레이션하는 함수 @sub\_main

```
define %struct.Memory* @sub_400520_main(%struct.State*, i64, %struct.Memory*)
```

wrapper 함수 @main  
(native 실행 문맥에서 호출할 때 lifted 실행 문맥으로 변환)

```
; Function Attrs: naked nobuiltin noline nounwind
define dso_local dllexport void @main() #6 !remill.function.type !1310 !remill.function.tie !1312 {
 tail call void @asm_sideeffect "pushq $0;pushq $0x91a;jmpq *$1;", "**m,*m,~{dirflag},~{fpsr},~{flags}"
 (%struct.Memory* (%struct.State*, i64, %struct.Memory*))** nonnull @7, void ()** nonnull @1 #7
 ret void
}
```

## 2-2. McSema 리프팅 코드 생성(계속)

### • McSema의 리프팅 방법3: 레퍼런스와 세그먼트

C프로그램

```
char *global = "I am global string";
char **ptr_to_global = &global;
char *hardcoded_ptr = 0x123456;
```

바이너리

Hex dump of section '.rodata':

```
0x004005b0 01000200 4920616d 20676c6f 62616c20I am global
0x004005c0 73747269 6e6700 string.
```

Hex dump of section '.data':

```
0x00601020 00000000 00000000 00000000 00000000
0x00601030 b4054000 00000000 30106000 00000000 ..@.....0.
```

리프팅된  
LLVM IR

```
@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) }>
```

## 2-3. 자체수정 코드 검출 방법

- 동적 방법

- 명령어 세트 시뮬레이터 기반
- 코드 영역 메모리를 실행 중에 모니터링
  - 페이지 기반 플래그
  - fetch한 명령어를 보관하고 나중에 fetch한 명령어와 비교
  - 하드웨어/운영체제에서 제공하는 방법 활용

=> 정적 방법

- Write 명령어에서 쓰기 대상의 주소가 코드 영역인지 정적 프로그램 포인터 분석

## 2-3. SMC 검출 방법: 아이디어

- SMC-C-42와 SMC-C-shell 프로그램에서 패턴
  - (1) 함수 주소를 가리키는 포인터
  - (2) 이 포인터에 x86 명령어 16진수 숫자를 대입

(1)

```
void foo(void);

int main(void)
{
 void *foo_addr = (void *)foo;

 if (change_page_permissions_of_address(foo_addr) != 0)
 fprintf(stderr, "Error : %s\n", strerror(errno));
 return 1;
}
```

(2)

```
unsigned char *instruction = (unsigned char *)foo_addr + 18;
*instruction = 0x2A;

puts("Calling foo...");
foo();

return 0;
}
```

i += 42;

```
void foo(void)
{
 int i = 0;
 i++;
 printf("i: %d\n", i);
}
```

## 2-3. SMC 검출 방법: 아이디어

- SMC 검출 정적 프로그램 분석 방법
  - **Step1: 포인터 분석** – 프로그램의 각 포인터 변수가 가리킬 수 있는 가능한 영역들의 집합을 구하기
    - 앤더슨 포인터 분석
    - 큐빅 제약식 풀이 엔진
  - **Step2: SMC write 명령어 찾기** – 코드 영역을 가리키는 포인터에 기록하는 명령어들을 찾아 리포트



## 2-3. SMC 검출 방법: 포인터 분석

### • 예제

$$Cells = \{p, q, x, y, z, alloc-1\}$$

```

p = alloc-1alloc null;
x = y;
x = z;
*p = x;
p = q;
q = &y;
x = *p;
p = &z;

```

(1) 앤더슨 알고리즘:  
프로그램에서  
부등식을 유도

```

alloc-1 ∈ [p]
[y] ⊆ [x]
[z] ⊆ [x]
c ∈ [p] ⇒ [z] ⊆ [c] for each c ∈ Cells
[q] ⊆ [p]
y ∈ [q]
c ∈ [p] ⇒ [c] ⊆ [x] for each c ∈ Cells
z ∈ [p]

```

(2) 큐빅 Solver:  
부등식을 만족하는 답을 구함

$[[V]]$  : 값/포인터 변수 V에 담길 값들의 집합

$$pt(p) = \{alloc-1, y, z\}$$

$pt(p)$  : 포인터 변수 p가 가리키는 셀들의 집합

$$pt(q) = \{y\}$$

## 2-3. SMC 검출 방법: 포인터 분석

- 앤더슨 알고리즘: 제약식 생성
  - 5가지 유형의 제약식 생성 방법
  - 이 외의 포인터 사용 유형은 5가지로 변형

- $X = \text{alloc } P$ :  $\text{alloc-i} \in [[X]]$
- $X1 = \& X2$ :  $X2 \in [[X1]]$
- $X1 = X2$ :  $[[X2]] \subseteq [[X1]]$
- $X1 = * X2$ : For each  $c \in \text{Cells}$ ,  $c \in [[X2]] \Rightarrow [[c]] \subseteq [[X1]]$
- $*X1 = X2$ : For each  $c \in \text{Cells}$ ,  $c \in [[X1]] \Rightarrow [[X2]] \subseteq [[c]]$

## 2-3. SMC 검출 방법: 포인터 분석

- 큐빅 알고리즘: 제약식 풀이
  - 토큰: 메모리(데이터/코드) 영역
  - 변수: 포인터 변수

제약식  $C := t \in v$   
|  $v1 \subseteq v2$   
|  $t \in v1 \Rightarrow v2 \subseteq v3$

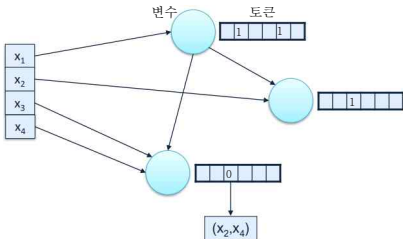
(C1) 초기화 제약식  
포인터 변수  $v$ 가  $t$  메모리 영역을 가리킴

(C2) 무조건 흐름 제약식  
 $v1$ 이 가리키는 것은 모두  $v2$ 도 가리킴

(C3) 조건부 흐름 제약식  
 $v1$ 이  $t$ 를 가리키면  
 $v2$ 가 가리키는 것은 모두  $v3$ 도 가리킴

## 2-3. SMC 검출 방법: 포인터 분석

- 큐빅 알고리즘: 제약식 풀이
  - 토큰: 메모리(데이터/코드) 영역
  - 변수: 포인터 변수
- 변수를 노드, 포인터 흐름을 에지로 표현하는 그래프



[알고리즘에 대한 자세한 설명은 보고서 참고]

### 3. 연구 결과

- SMC Bench 자체수정 벤치마크 프로그램
- LLVM IR 대상 SMC Analyzer
- SMC 검출 실험 결과 및 개선점

### 3. 연구 결과: SMC Bench

- SMC Bench 자체수정 벤치마크 프로그램
  - smc1 ~ smc9: 자체수정 코드의 9가지 유형
  - MIPS 1세트(smcN.mips.s), x86 1세트(smcN.x86.s), C 1세트(smcN.c): 총 27개 프로그램

<https://github.com/JNU-SoftwareLAB/SMC-Bench/tree/master/src/asm>

| 이름   | 설명                                                                     |
|------|------------------------------------------------------------------------|
| smc1 | 무한 코드 변경(Unbounded code rewriting) : 피보나치 수열                           |
| smc2 | 런타임 코드 검사(Runtime code checking) : 호출한 프로그램의 유효성 검사                    |
| smc3 | 런타임 코드 생성(Runtime code generation) : 벡터 내적                             |
| smc4 | 다중 런타임 코드 생성(Multilevel runtime code generation) : 생성한 코드가 또 다른 코드를 생성 |
| smc5 | 스스로 바꾸고 다시 돌아오는 코드(Self-mutating code block)                           |
| smc6 | 서로 상대를 변경하는 코드(Mutual modifying modules)                               |
| smc7 | 스스로 복제해서 증가하는 코드(Self-growing code)                                    |
| smc8 | 동일한 알고리즘을 여러 형태로 바꾸는 코드(Polymorphic code)                              |
| smc9 | 암호/압축을 푸는 코드(Encrypting and decrypting code)                           |

### 3. 연구 결과: SMC Analyzer

- SMC Analyzer: LLVM IR 대상 SMC 검출 정적 프로그램 분석
  - Step1: 포인터 분석
    - LLVM IR 명령어에 대한 제약식 생성
    - C++기반 큐빅 엔진 구현
  - Step2: 코드 영역을 write하는 명령어 찾기
    - 포인터 분석 결과를 이용하여
    - LLVM IR 프로그램 탐색하여 store나 memcpy 명령어의 LHS 포인터가 코드 영역을 가리키는지 확인하여 리포트



<https://github.com/JNU-SoftwareLAB/SMC-Bench/tree/master/src/tools/Analyzer>

### 3. 연구 결과: SMC Analyzer 결과 예시

- smc1.c.ll.result

=====

Function : main  
BasicBlock :main!0  
Order :9

Variable: [[ main!2 ]]

Tokens: { Constant-Value, main, main!2, }

Instruction: store i8\* getelementptr inbounds (i8, i8\* bitcast (i32 ()\* @main to i8\*), i64 107), i8\*\* %"main!2", align 8

=====

=====

Function : main  
BasicBlock :main!0  
Order :14

Variable: [[ main!8 ]]

Tokens: { Constant-Value, main, main!2, main!4, main!8, main!9, }

Variable: [[ main!9 ]]

Tokens: { Constant-Value, main, main!2, main!9, }

Instruction: call void @llvm.memcpy.p0i8.p0i8.i64(i8\* align 1 %"main!8", i8\* align 1 %"main!9", i64 4, i1 false)

=====

=====

Function : main  
BasicBlock :main!29  
Order :8

Variable: [[ main!34 ]]

Tokens: { Constant-Value, main, main!2, main!27, main!34, main!35, main!5, }

Instruction: call void @llvm.memcpy.p0i8.p0i8.i64(i8\* align 1 %"main!34", i8\* align 1 %"main!35", i64 4, i1 false)

=====

Execution Time: 107.947ms

- 함수 main의
- 0번째 블록
- 9번째 명령어에서
- 변수 %2가 가리키는 코드 영역 main에
- store를 시도하는 SMC 동작 검출!!



### 3. 연구 결과: SMC Analyzer

#### • LLVM IR 명령어 제약식 생성(1/2)

| INSTRUCTION   | CONSTRAINT                                                            | SYNTAX                                                                                                             |
|---------------|-----------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------|
| ALLOCA        | $result \in [[result]]$                                               | <code>&lt;result&gt; = alloca &lt;type&gt;</code>                                                                  |
| INTTOPTR      | $[[value]] \subseteq [[result]]$                                      | <code>&lt;result&gt; = inttoptr<br/>&lt;ty&gt; &lt;value&gt; to &lt;ty2&gt;</code>                                 |
| BITCAST       | $[[value]] \subseteq [[result]]$                                      | <code>&lt;result&gt; = bitcast<br/>&lt;ty&gt; &lt;value&gt; to &lt;ty2&gt;</code>                                  |
| PHI           | $[[val1]] \subseteq [[result]], [[val2]] \subseteq [[result]], \dots$ | <code>&lt;result&gt; = phi &lt;ty&gt;<br/>[ &lt;val10&gt;, &lt;label10&gt;], ...</code>                            |
| SELECT        | $[[val1]] \subseteq [[result]], [[val2]] \subseteq [[result]]$        | <code>&lt;result&gt; = select selty<br/>&lt;cond&gt;, &lt;ty&gt; &lt;val1&gt;,<br/>&lt;ty&gt; &lt;val2&gt;</code>  |
| EXTRACTVALUE  | $[[val]] \subseteq [[result]]$                                        | <code>&lt;result&gt; = extractvalue<br/>&lt;aggregate type&gt; &lt;val&gt;,<br/>&lt;idx&gt;{, &lt;idx&gt;}*</code> |
| STORE         | $c \in [[pointer]] \Rightarrow [[value]] \subseteq [[c]]$             | <code>store [volatile]<br/>&lt;ty&gt; &lt;value&gt;,<br/>&lt;ty&gt;* &lt;pointer&gt;</code>                        |
| LOAD          | $c \in [[pointer]] \Rightarrow [[c]] \subseteq [[result]]$            | <code>&lt;result&gt; = load [volatile]<br/>&lt;ty&gt;, &lt;ty&gt;* &lt;pointer&gt;</code>                          |
| GETELEMENTPTR | $c \in [[ptrval]] \Rightarrow [[c]] \subseteq [[result]]$             | <code>&lt;result&gt; = getelementptr<br/>&lt;ty&gt;, &lt;ty&gt;* &lt;ptrval&gt;<br/>{, &lt;ty&gt; idx}*</code>     |

### 3. 연구 결과: SMC Analyzer

- LLVM IR 명령어 제약식 생성(2/2)

- call 명령어 `%result = call i32@(val1, val2, ... valN)`
  - 함수: `fn(param1, param2, ...){... ret return_variable}`
    - $[[val1]] \subseteq [[param1]]$
    - $[[val2]] \subseteq [[param2]]$
    - ...
    - $[[valN]] \subseteq [[paramN]]$
    - $[[return\_variable]] \subseteq [[result]]$

### 3. 연구 결과: SMC Analyzer

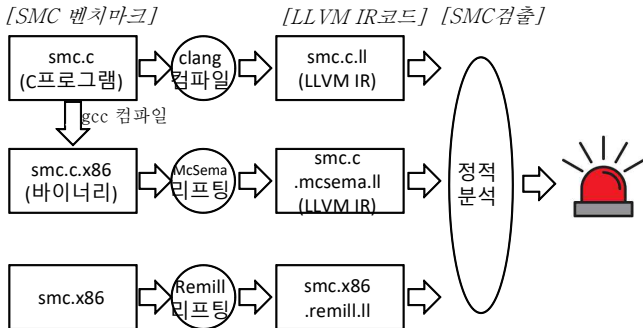
- Cubic 엔진 인터페이스: 3개 유형 제약식 추가
  - 토큰  $t$ : 데이터/코드 영역
  - 변수  $v$ : 포인터 변수

제약식  $C := t \in v$  (C1) 초기화 제약식  
|  $v1 \subseteq v2$  (C2) 무조건 흐름 제약식  
|  $t \in v1 \Rightarrow v2 \subseteq v3$  (C3) 조건부 흐름 제약식

- $\text{token} \in [[\text{variable}]]$  제약식 추가:  
CubicSolver.addConstantConstraint(Operand \*token, Operand \*variable)
- $[[\text{variable1}]] \subseteq [[\text{variable2}]]$  제약식 추가:  
CubicSolver.addSubsetConstraint(Operand \*variable1, Operand \*variable2)
- $c \in [[\text{variable1}]] \Rightarrow [[\text{variable2}]] \subseteq [[\text{variable3}]]$  제약식 추가:  
CubicSolver.addConditionalConstraint(T t, V x, V y, V z)

### 3. 연구 결과: SMC 검출 결과

- SMC 검출 실험 환경



### 3. 연구 결과: SMC 검출 결과1

- SMC.c.ll 분석: 9개 모두 분석 성공



| smc.c.ll | 정답 | 분석 결과 | 명령어 수 | 시간(ms) |
|----------|----|-------|-------|--------|
| smc1     | O  | O     | 93    | 202    |
| smc2     | X  | X     | 178   | 748    |
| smc3     | O  | O     | 202   | 997    |
| smc4     | O  | O     | 136   | 456    |
| smc5     | O  | O     | 214   | 1,079  |
| smc6     | O  | O     | 179   | 716    |
| smc7     | O  | O     | 80    | 284    |
| smc8     | O  | O     | 232   | 1976   |
| smc9     | O  | O     | 112   | 306    |

### 3. 연구 결과: SMC 검출 결과1

- SMC.c.ll에서 코드 주소 패턴이 명확함

[ smc1.c.ll ]

```
; Function Attrs: noinline nounwind optnone
define dso_local i32 @main() #0 {
 %1 = alloca i32, align 4
 %2 = alloca i8*, align 8
 ; unsigned char* ptr_key
 %3 = alloca i32, align 4
 ; int cnt
 %4 = alloca [4 x i8], align 1
 ; unsigned char instr9[4]
 %5 = alloca [4 x i8], align 1
 ; unsigned char instr10[4]
 %6 = alloca i32, align 4
 ;int index
 %7 = alloca i8, align 1
 ;unsigned char fib_index
 store i32 0, i32* %1, align 4
 store i8* getelementptr inbounds (i8, i8* bitcast (i32 ()* @main to i8*), i64 107), i8** %2, align 8
 ; unsigned char* ptr_key = (unsigned char*)main + 107
 call void @get_permission(i8* bitcast (i32 ()* @main to i8*))
```

[ smc1.c ]

```
int main(void){
 // code ptr to key
 unsigned char* ptr_key = (unsigned char*)main + 107; // objdump로 확인!!!

 int counter;
 unsigned char instr9[4];
 unsigned char instr10[4];
 int index;
 unsigned char fib_index;

 // initialize
 get_permission(main);
```

(초기화 제약식을 정확히 만들 수 있음!)

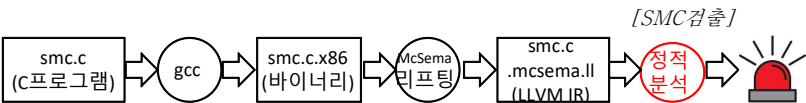
@main ∈ [[%2]]



%2 → ptr\_key

### 3. 연구 결과: SMC 검출 결과2

- SMC.c.mcsema.ll 분석



| smc.c.mcsema.ll | 정답 | 1차 결과 | 2차 결과 | 명령어 수 | 시간(ms) |
|-----------------|----|-------|-------|-------|--------|
| smc1            | O  | X     | O     | 2,009 | 35,135 |
| smc2            | X  | X     | O     | 2,568 | 51,385 |
| smc3            | O  | X     | O     | 2,071 | 35,877 |
| smc4            | O  | X     | O     | 2,167 | 40,985 |
| smc5            | O  | X     | O     | 2,122 | 37,447 |
| smc6            | O  | X     | O     | 1,965 | 31,519 |
| smc7            | O  | X     | O     | 1,830 | 28,506 |
| smc8            | O  | X     | O     | 2,018 | 33,071 |
| smc9            | O  | X     | O     | 2,309 | 45,271 |

### 3. 연구 결과: SMC 검출 결과2

- SMC.c.mcsema.ll 분석

- 1차 분석에서 모두 실패한 이유:  $t \in v$ 
  - SMC 검출에 필요한 (C1) 초기화 제약식이 생성되지 않음
  - smc.c를 x86을 PIC (position independent code)로 컴파일하면 전역 함수 이름 @main이 코드에서 사라짐
  - 그 대신 @main의 주소를 참조할 때 특별한 코드 패턴 사용
- 2차 분석: 이 코드 패턴 발견=>초기화 제약식 추가!
  - 바이너리 코드에 대한 도메인 지식 => 제약식 추가
  - smc1, smc3~9 모두 SMC 검출!!
  - smc2에 대해 오탐!!
  - smc2의 전역변수(함수가 아닌)에 대해서도 해당 코드 패턴이 사용되어 오탐이 발생함
    - 전역변수와 함수를 구분하는 더 정교한 패턴 탐지 필요



### 3. 연구 결과: SMC 검출 결과2

- SMC.c.mcsema.ll의 PIC 코드 패턴(1/3)
  - PIC: 임의의 메모리 주소에 로딩 할 수 있는 바이너리 구조 (e.g., shared library)
  - 전역 함수(또는 전역 변수) 이름을 찾는 방법
    - Step1: 현재 명령어의 주소를 가져오기
      - 함수 호출: `call x86.get_pc_thunk_bx`
      - 함수 정의: `x86.get_pc_thunk_bx:`  
`call L`  
`L: pop %ebx`
    - Step2: 현재 PC의 상대 오프셋으로 저장된 전역 이름의 주소를 읽기 (cf. GOT, Global offset table)

### 3. 연구 결과: SMC 검출 결과2

#### • SMC.c.mcsema.11의 PIC 코드 패턴 (2/3)

```
=====
smc1.c
=====
```

```
int main(void){
// code ptr to key
unsigned char* ptr_key = (unsigned char*)main + 107;
// objdump로 확인!!!
```

```
int counter;
unsigned char instr9[4];
unsigned char instr10[4];
int index;
unsigned char fib_index;
```

```
// initialize
get_permission(main);
```

PIC 방식 코드

%eax는  
main의 주소

```
=====
[smc1.c.x86.objdump]
```

```
=====
0000063d <main>:
63d: 8d 4c 24 04 lea 0x4(%esp),%ecx
641: 83 e4 f0 and $0xfffff0,%esp
644: ff 71 fc pushl -0x4(%ecx)
647: 55 push %ebp
648: 89 e5 mov %esp,%ebp
64a: 53 push %ebx
64b: 51 push %ecx
64c: 83 ec 20 sub $0x20,%esp
64f: e8 ec fe ff call 540 <__x86.get_pc_thunk.bx>
654: 81 c3 70 19 00 00 add $0x1970,%ebx
65a: 65 a1 14 00 00 00 mov %gs:0x14,%eax
660: 89 45 f4 mov %eax,-0xc(%ebp)
663: 31 c0 xor %eax,%eax
665: 8d 83 e4 e6 ff ff lea -0x191c(%ebx),%eax
66b: 89 45 e4 mov %eax,-0x1c(%ebp)
66c: 83 ec 0c sub $0xc,%esp
671: 8d 83 e6 ff ff lea -0x1987(%ebx),%eax
677: 50 push %eax
678: e8 88 00 00 00 call 705 <get_permission>
```

### 3. 연구 결과: SMC 검출 결과2

#### • SMC.c.mcsema.ll의 PIC 코드 패턴 (3/3)

```
=====
[smc1.c.mcsema.ll - @sub_60d_main]
=====
```

```
%63 = tail call %struct.Memory*
@sub_540___x86_get_pc_thunk_bx(%struct.State*
nonnull %0, i32 %58, %struct.Memory* %2)
%64 = load i32, i32* %4, align 4
%65 = load i32, i32* %9, align 4
%66 = add i32 %64, 6512
store i32 %66, i32* %4, align 4, !tbaa !1275
%67 = load i32, i32* %4, align 4
%68 = add i32 %67, 1
%69 = inttoptr i32 %68 to i32*
%70 = load i32, i32* %69
%71 = load i32, i32* %8
%72 = add i32 %71, -12
%73 = inttoptr i32 %72 to i32*
store i32 %70, i32* %73
%74 = add i32 %64, 84
%75 = add i32 %71, -28
%76 = inttoptr i32 %75 to i32*
store i32 %74, i32* %76
i32 %83, %struct.Memory* %63)
%89 = load i32, i32* %7, align 4
```

PIC 도메인 지식으로 추가한  
초기화 제약식

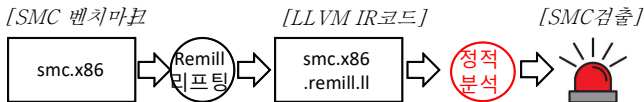
@main ∈ [[%9]]

%9에 저장된 주소가  
main 코드 영역을 가리킴

```
%77 = load i32, i32* %7, align 4
%78 = add i32 %64, -23
store i32 %78, i32* %3, align 4, !tbaa !1275
%79 = add i32 %77, -16
%80 = load i32, i32* %22, align 8, !tbaa !1279
%81 = add i32 %80, %79
%82 = inttoptr i32 %81 to i32*
store i32 %78, i32* %82
%83 = add i32 %65, 177
%84 = add i32 %65, 41
%85 = add i32 %77, -20
%86 = add i32 %80, %85
%87 = inttoptr i32 %86 to i32*
store i32 %84, i32* %87
store i32 %85, i32* %7, align 4, !tbaa !1275
%88 = tail call %struct.Memory*
@sub_705_get_permission(%struct.State* nonnull %0,
i32 %83, %struct.Memory* %63)
%89 = load i32, i32* %7, align 4
```

### 3. 연구 결과: SMC 검출 결과3

- SMC.x86.remill.ll 분석



- 결과: 모두 SMC 검출 하지 못함
- smc.x86에 @main과 같은 함수 이름이 없음
- smc.x86은 Non-PIC 형식: 프로그래머가 직접 코딩한 어셈블리 명령어를 그대로 사용
- Non-PIC 형식 바이너리에서 함수 주소를 참조하는 코드 패턴을 찾아 (C1) 초기화 제약식을 생성해야 함!
  - 바이너리 코드에 대한 도메인 지식 => 제약식 추가

### 3. 연구 결과: SMC 검출 결과3

#### • SMC.x86.remill.11 분석(계속)

- 명령어에 나타난 상수가 코드 주소임을 파악하고, 적절한 초기화 제약식을 만들어야 함 (McSema 지원 필요???)

[ smc1.x86.s ]

```
section .data
 num db 8

section .text
global _init
_init:
 mov ebp, num ; num
 mov ecx, 1 ; counter
 mov edx, 1 ; accumulator

loop:
 cmp ecx, [ebp] ; compare num
 jz halt ; if num and
 add ecx, 1 ; else, coun
 mov bl, dl ; save const

key:
 add edx, 0 ; Store next
 mov al, bl ; store Fn f
 mov [key+2], al ; update con
 jmp loop
```

[ smc1.x86.objdump ]

```
smc1.x86.out: file format elf32-i386

Disassembly of section .text:

00408060 <_init>:
00408060: bd 90 00 04 00 mov $0x00408090,%ebp
00408065: b9 01 00 00 00 mov $0x1,%ecx
0040806a: ba 01 00 00 00 mov $0x1,%edx

0040806f <loop>:
0040806f: 3b 4d 00 cmp 0x0(%ebp),%ecx
00408072: 74 11 je 00408085 <halt>
00408074: 83 c1 01 add $0x1,%ecx
00408077: 88 d3 mov %dl,%bl

00408079 <key>:
00408079: 83 c2 00 add $0x0,%edx
0040807c: 88 d8 mov %bl,%al
0040807e: a2 7b 80 04 00 mov $al,0x804807b
00408083: eb ea jmp 0040806f <loop>
```

0x804807b

## 4. 결론 및 활용 방안

- 바이너리 대상 SMC 검출 방법

- ✓ SMC Bench 프로그램
- ✓ LLVM 프레임워크를 대상 SMC Analyzer
- ✓ 벤치마크 대상 SMC 탐지 실험 결과 및 논의

- 활용 방안

- SMC Analyzer 고도화 – 바이너리 전문가의 도메인 지식을 추가할 수 있는 유연한 분석기 아키텍처
  - 바이너리 분석가가 LLVM IR 명령어에 주석으로 제약식 (C1,C2,C3)를 직접 작성!
- SMC 동작을 보이는 악성 바이너리 분석에 SMC Analyzer를 활용하여 장단점을 분석

# 연구 논문에 대한 안(~2020.12)

- SMC Bench와 SMC Analyzer 논문을 작성하기 위해 앞으로 연구할 주제

- 분석 대상

- SMC Bench
- 악성 바이너리 사례 1~2 case study (x86 1건, ARM 1건)
- x86 바이너리 + ARM 바이너리

국보연에서  
(SMC 동작이 이미 파악된)  
샘플 2건을 제공해주실 수 있으실까요?

- SMC Analyzer 고도화

- 명시적 함수 주소 패턴 (e.g., smc.c.ll)
- PIC 방식에서 함수 주소 패턴 (e.g., smc.x86.mcsema.ll)
- Non-PIC 방식에서 함수 주소 패턴 (e.g., smc.x86.objdump - 상수 주소 처리)

McSema에서 ARM 바이너리를  
LLVM IR로 충분히(?) 리프팅  
할 수 있는지 확인 필요