|  |
| --- |
| **LLVM 기반의 BareMetal IoT 디바이스 펌웨어 난독화**  박우진(컴퓨터공학과, 2015104175)  이한솔(컴퓨터공학과, 2017110275)  **요 약**  최근 다양한 IoT 디바이스의 사용이 증가되고 있고, 그로 인해 보안 위협도 증가되고 있다. 베어메탈(Baremetal) IoT 디바이스의 펌웨어는 펌웨어 업데이트 파일 및 Flash De-soldering 등으로 추출이 가능 하며, 이를 역공학(Reverse Engineering) 툴을 이용한 정적 분석을 통해 실행 흐름을 분석하여 취약점을 찾을 수 있다. 이를 해결하기 위해 일반적인 컴퓨팅 시스템에는 정적 분석을 어렵게 하기 위한 다양한 소스코드 기반의 난독화 방법이 존재한다. 그러나, 기존 바이너리 코드 난독화에 대한 연구는 다양하게 전개된 사례는 있으나, 베어메탈 기반의 IoT 디바이스를 타겟으로 하는 난독화 기법의 연구 전개는 미비한 상황이다.이를 해결하기 위해 LLVM Pass를 이용한 난독화된 바이 너리 코드를 실행하는 코드의 삽입을 통해 베어메탈 IoT 디바이스 펌웨어의 바이너리 코드를 난독화 하는 방안을 제안한다. |

**1. 서론**

**1.1. 연구배경**

최근 들어 IoT 기술을 적용한 가전제품을 포함한 여러 기기 사용이 증가되고 있고, 그에 따른 보안의 중요성이 강조되고 있다. 하지만 IoT 디바이스 제조사들의 보안을 고려하지 않은 개발로 인해 IoT 디바이스에 대한 공격이 증가하고 있다. 과학기술 정보통신부에 따르면 2015년부터 2019년 상반기까지 loT 보안 취약점 신고 건수는 총 1,414건으로 3 배 이상 건수가 증가하였다[1]. IoT 디바이스들의 펌웨어는 플래시 De-soldering 등을 통해 추출이 가능하고, 추출한 펌웨어는 정적 분석 방법을 통해 코드의 흐름을 파악할 수 있다 [2]. 특히, Ghidra, IDA Pro와 같은 프로그램은 바이너리 파일을 손쉽게 분석할 수 있도록 디컴파일등의 기능도 지원한다[3]. 위의 도구들을 활용하여 누구나 쉽게 펌웨어의 분석이 가능하므로 펌웨어의 분석을 어렵게 하는 난독화 기법이 요구된다.

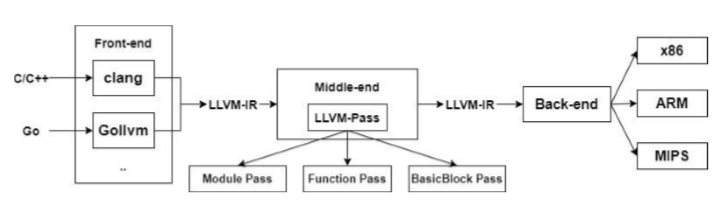
소프트웨어의 코드 정적 분석을 어렵게 하기 위해 Obfuscator-LLVM[4]에서는 다양한 소스코드 기반 난독화 기법을 제안하였다. 하지만 이러한 소스코드 기반 난독화 기법은 제어 흐름 그래프(Control Flow Graph)의 분석을 통해 쉽게 난독화 해제가 가능하다.

**1.2. 연구목표**

본 연구에서는 베어메탈 IoT 디바이스의 펌웨어 난독화를 위해 펌웨어의 바이너리 코드 자체를 난독화 하는 방안을 제안한다. 베어메탈 IoT 디바이스는 펌웨어가 하나의 바이너리 파일로 이루어져 있으며 이 펌웨어의 코드는 성능을 위해 플래시 메모리에서 실행되는 XIP(eXecute In Place)[5]기술을 이용하여 실행된다. 펌웨어 바이너리 코드 전체를 난독화 하여 수행할 경우 난독화를 해제하기 위해 펌웨어 코드 전체가 RAM에 올라가야 하므로 실행 시 오버헤드가 크게 발생할 수 있다. 이를 해결하기 위해 펌웨어 바이너리 코드의 주요 Instruction인 Return Instruction만 난독화 하여 제어 흐름을 알 수 없게 하면서 난독화 한 일부 Instruction만 RAM에 올려 난독화를 해제하여 실행시키는 방안을 제시한다.

**2. 관련연구**

**2.1. LLVM**



[그림 1] LLVM 구성도

LLVM (Low Level Virtual Machine)은 Chris Lattner에 의해 작성된 컴파일러 프로젝트이다[6]. LLVM은 다양한 프로그래밍 언어의 정적 컴파일과 동적 컴파일을 모두 지원할 수 있는 컴파일 환경을 제공한다. LLVM은 [그림 1]과 같이 Front-end, Middle-end, Back-end로 나누어진다. Front-end는 C, C++, Go 언어와 같은 다양한 프로그래밍 언어로 작성된 소스 코드를 분석 후 이 코드를 LLVM에서 사용하는 어셈블리언어와 유사한 중간 표현인 IR(Intermediate Representation) 형식의 명령어를 생성한다. LLVM에서 주로 사용되는 Front-end로는 C/C++와 같은 언어를 위한 Clang이 존재한다. Middle-end에서는 변환된 IR 코드를 최적화한다. 제어 흐름 그래프를 단순화하거나, 가짜 반복문을 제거하는 역할을 하며 이 과정은 LLVM의 Pass를 이용한 code instrumentation을 통해 이루어진다. Back-end에서는 Middle-end에서 변환된 IR 코드를 타겟의 아키텍처에 맞는 기계어 바이너리 코드로 변환해준다. 이 과정에서도 다양한 컴파일러상의 최적화가 가능하다.

**2.1.1. LLVM/Clang 빌드**

다음은 LLVM과 Clang을 빌드하는 과정을 나타낸다. Ubuntu 18.04 운영체제 기준으로 빌드를 수행한다.

$ sudo apt-get install cmake   
$ git clone https://github.com/llvm/llvm-project.git  
$ cd llvm-project  
$ mkdir build  
$ cd build  
$ cmake -DLLVM\_ENABLE\_PROJECTS=clang -G "Unix Makefiles" ../llvm  
$ make

빌드 시 메모리 부족으로 인한 오류가 발생할 수 있다. 이 경우 Swap 영역을 늘려주거나 물리 메모리를 증가시켜야 한다.

빌드 결과는 build/bin에 저장된다.

**2.1.2. LLVM Clang을 이용한 빌드 수행**

**Test Code:**

#include <stdio.h>  
  
int main() {  
 printf(“Hello World!\n”);  
 return 0;  
}

다음의 소스코드를 test.c 파일로 작성하였다고 가정하자.

**실행 파일 생성**

$ build/bin/clang test.c -o test

**오브젝트 파일 생성**

$ build/bin/clang test.c -c test.o

**IR 코드 생성**

$ build/bin/clang test.c -c -S -emit-llvm -o test.ll

생성된 IR 코드는 다음과 같다.

@.str = private unnamed\_addr constant [13 x i8] c"Hello World\0A\00", align 1  
  
; Function Attrs: noinline nounwind optnone uwtable  
define dso\_local i32 @main() #0 {  
entry:  
 %retval = alloca i32, align 4  
 store i32 0, i32\* %retval, align 4  
 %call = call i32 (i8\*, ...) @printf(i8\* getelementptr inbounds ([13 x i8], [13 x i8]\* @.str, i64 0, i64 0))  
 ret i32 0  
}

**2.1.3. LLVM Pass**

LLVM Pass는 LLVM의 Middle-end에서 사용되는 최적화 모듈이며, [그림 1]과 같이 Function, Module, Basic Block 등의 단위로 code instrumentation을 수행할 수 있다. 본 연구에서 소개할 바이너리 코드 난독화 기법은 LLVM의 Middle-end에서 적용되며, 바이너리 코드 난독화 및 난독화 된 코드 실행을 위한 Pass를 작성하여 구현하였다.

**2.1.3.1. LLVM Pass 작성**

llvm/lib/Transforms에 Code Instrumentation을 위한 새로운 Pass를 작성할 수 있다.

1. 작성하고자 하는 Pass 이름이Obfuscation이라면 llvm/lib/Transforms/Obfuscation directory를 생성한다.

2. llvm/lib/Transforms/CMakeLists.txt 에 Obfuscation Pass를 빌드하기 위한subdirectory를 추가한다.

add\_subdirectory(Utils)  
add\_subdirectory(Instrumentation)  
add\_subdirectory(AggressiveInstCombine)  
add\_subdirectory(InstCombine)  
add\_subdirectory(Scalar)  
add\_subdirectory(IPO)  
add\_subdirectory(Vectorize)  
add\_subdirectory(Hello)  
add\_subdirectory(ObjCARC)  
add\_subdirectory(Coroutines)  
add\_subdirectory(CFGuard)  
**add\_subdirectory(Obfuscation)**

3. 빌드를 위해 llvm/lib/Transforms/Obfuscation/CMakeLists.txt 파일을 작성한다.

add\_llvm\_library( **LLVMObfuscation** MODULE  
 **Obfuscation.cpp**   
  
 DEPENDS  
 intrinsics\_gen  
 PLUGIN\_TOOL  
 opt  
 )

LLVMObfuscation이라는 이름의 라이브러리로 Pass를 생성하며, Obfuscation.cpp는 실제 Pass의 동작을 가지는 소스코드이다.

4. 실제 Pass의 작동을 기술하는 Obfuscation.cpp 소스코드를 작성한다.

// Obfuscation.cpp  
#include "llvm/Pass.h"  
#include "llvm/IR/Function.h"  
#include "llvm/IR/Module.h"  
#include "llvm/Support/Alignment.h"  
#include "llvm/Support/raw\_ostream.h"  
using namespace llvm;  
  
namespace {  
 struct Obfuscation : public FunctionPass {  
 static char ID;  
  
 Obfuscation() : FunctionPass(ID) { }  
 bool runOnFunction(Function &F) override {   
 errs() << F.getName() << '\n';  
 return false;  
 }   
  
 }; // end of struct Obfuscation  
} // end of anonymous namespace

char Obfuscation::ID = 0;  
  
static RegisterPass<Obfuscation> X("obfus", "Obfuscation Pass", false, false);

Function Pass로 Pass를 작성하였고, IR 코드에 Pass 적용 시 IR코드의 각 Function들에 대해 Obfuscation.cpp에 정의한 code instrumentation 방법이 적용되게 된다.

C++의 Class 형식의 구조체를 사용하며, runOnFunction 멤버함수에 Pass 적용시 수행되는 작업 절차를 정의하며 위 소스코드는 파라미터로 들어온 함수의 이름을 출력하는 Pass를 작성한 코드이다. 또한 RegisterPass를 이용해 opt의 obfus 파라미터로 Pass를 수행할 수 있도록 해준다.

스크린샷, 그리기이(가) 표시된 사진

자동 생성된 설명

[그림 2] Pass Function, Module, BasicBlock Class

[그림 2]는 각각 Module Class, Function Class, BasicBlock Class의 상관관계를 보여준다. Module Class내의 List에는 Function들이 존재하고 Iterator class를 이용해 Function들을 iteration할 수 있다. 또한 Function Class내의 List에는 Basic Block들이 존재하고 Iterator class를 이용해 Basic Block들을 iteration할 수 있다. 또한 BasicBlock 내의 List에는 각각의 IR형태의 Instruction Object들이 존재하며, BasicBlock의 Iterator class를 이용해 Instruction 들을 iteration할 수 있다.

Instruction Class의 getParent() 멤버함수 호출로 Instruction의 부모 BasicBlock Object를 가져올 수 있고, BasicBlock Class의 getParent() 멤버함수 호출로 Basic Block의 부모 Function Object를 가져올 수 있으며 Function Class의 getParent() 멤버함수 호출로 Function의 부모 Module Object를 가져올 수 있다.

5. LLVM을 새로 빌드 후 IR 코드에 Pass 적용

$ build/bin/opt -load build/lib/LLVMObfuscation.so -obfus test.ll -S -o test.ll

LLVMObfuscation Pass라이브러리를 로드 한 후 obfus Pass를 수행하도록 한다. 위에서 만든 test.ll 파일에 Pass적용 후 결과를 test.ll로 출력시킨다.

위에서 작성한 Pass는 함수 이름만 출력하므로, “main”만이 출력된다.

**2.2. Obfuscator-LLVM**

oLLVM(Obfuscator LLVM)[4]은 Pascal에 의해 개발된 난독화 도구이다. oLLVM은 LLVM IR에 대해 난독화를 하는 패스(Pass)로서 구현되어 있어서 프로그램언어에 의존하지 않고 타겟 아키텍쳐로부터 독립적이라는 장점이 있다. 현재 oLLVM에는 Bogus Control Flow, Instructions Substitution, Control Flow Flattening의 3종류의 난독화가 존재한다.

**2.2.1 가짜 제어 흐름(Bogus Control Flow)**

BCF는 Collberg가 제안한 불투명 술어(Opaque Predicates)을 확장시킨 난독화이다. 이 방법은 현재 베이직 블럭(basic block) 전에 하나의 베이직 블럭을 추가하여 함수 호출 그래프를 수정하는 것이다. 이 새로운 베이직 블럭은 불투명 술어를 포함하고 원래 베이직 블럭으로 조건 분기를 하게 한다.

컴파일러 옵션

1. -mllvm -bcf : BCF 패스를 추가한다.
2. -mllvm -bcf\_loop=3 : 패스가 활성화되면, 함수에 3회 적용한다. 디폴트 값은 1
3. -mllvm -bcf\_prob=40 : 패스가 활성화되면, 베이직 블럭은 40%의 확률로 난독화된다. 디폴트 값은 30이다.

예시

|  |
| --- |
| #include <stdlib.h>  int main(int argc, char\*\* argv) {  int a = atoi(argv[1]);  if(a == 0)  return 1;  else  return 10;  return 0;  } |

아래와 같은 IR로 변환된다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

가짜 제어 흐름 패스 후, 아래와 같은 제어 그래프를 얻을 수 있다.

텍스트이(가) 표시된 사진

자동 생성된 설명

**2.2.2 명령어 치환(Instructions Substitution)**

이 방법은 산술 또는 논리 연산자를 기능적으로 동등하게 대체한다. 하지만 더 복잡한 명령어 열로 대체된다. 여러 동등한 명령어 열이 가능하다면, 하나는 랜덤으로 선택된다. 이 난독화 방법은 만들어진 코드를 re-optimizing하여 쉽게 제거 될 수 있기 때문에, 많은 보안을 추가하지 않는다.

컴파일러 옵션

1. -mllvm -sub : 명령 치환은 이 옵션을 사용하면 된다.
2. -mllvm -sub\_loop=3 : 패스가 활성화되면 함수에 3회 적용한다. 디폴트 값은 1이다.

예시

- 덧셈

|  |
| --- |
| %0 = load i32\* %a, align 4  %1 = load i32\* %b, align 4  %2 = sub i32 0, %1  %3 = sub nsw i32 %0, %2 |

|  |
| --- |
| %0 = load i32\* %a, align 4  %1 = load i32\* %b, align 4  %2 = sub i32 0, %0  %3 = sub i32 0, %1  %4 = add i32 %2, %3  %5 = sub nsw i32 0, %4 |

|  |
| --- |
| %0 = load i32\* %a, align 4  %1 = load i32\* %b, align 4  %2 = add i32 %0, 1107414009  %3 = add i32 %2, %1  %4 = sub nsw i32 %3, 1107414009 |

|  |
| --- |
| %0 = load i32\* %a, align 4  %1 = load i32\* %b, align 4  %2 = sub i32 %0, 1108523271  %3 = sub i32 %2, %1  %4 = add nsw i32 %3, 1108523271 |

- 뺄셈

|  |
| --- |
| %0 = load i32\* %a, align 4  %1 = load i32\* %b, align 4  %2 = sub i32 0, %1  %3 = add nsw i32 %0, %2 |

|  |
| --- |
| %0 = load i32\* %a, align 4  %1 = load i32\* %b, align 4  %2 = sub i32 %0, 1057193181  %3 = sub i32 %2, %1  %4 = add nsw i32 %3, 1057193181 |

- AND

|  |
| --- |
| %0 = load i32\* %a, align 4  %1 = load i32\* %b, align 4  %2 = xor i32 %1, -1  %3 = xor i32 %0, %2  %4 = and i32 %3, %0 |

- OR

|  |
| --- |
| %0 = load i32\* %a, align 4  %1 = load i32\* %b, align 4  %2 = and i32\* %0, %1  %3 = xor i32 %0, %1  %4 = or i32 %2, %3 |

- XOR

|  |
| --- |
| %0 = load i32\* %a, align 4  %1 = load i32\* %b, align 4  %2 = xor i32 %0, -1  %3 = and i32 %1, %2  %4 = xor i32 %1, -1  %5 = and i32 %0, %4  %6 = or i32 %3, %5 |

**2.2.3 제어 흐름 평탄화(Control Flow Flattening)**

이 패스의 목적은 프로그램의 제어 흐름 그래프를 완전하게 평단화하는 것이다. 제어 구조를 분기로 하여금 평단화하고 알고리즘의 특징을 읽기 어렵게 하는 방법이다.

컴파일러 옵션

1. -mllvm -fla : 이 옵션을 사용하여 CFF를 사용할 수 있다.
2. -mllvm -split : 베이직 블럭 분리한다. 함께 적용했을 때, 평탄화를 향상시킨다.
3. -mllvm -split\_num=3 : 패스가 활성화되면, 각 베이직 블럭에서 3회 적용된다. 디폴트 값은 1이다.

예시

|  |
| --- |
| #include <stdlib.h>  int main(int argc, char\*\* argv) {  int a = atoi(argv[1]);  if(a == 0)  return 1;  else  return 10;  return 0;  } |

평탄화 패스는 아래와 같이 변환된다.

|  |
| --- |
| #include <stdlib.h>  int main(int argc, char\*\* argv) {  int a = atoi(argv[1]);  int b = 0;  while(1) {  switch(b) {  case 0 :  if(a == 0)  b = 1;  else  b = 2;  break;  }  }  return 0;  } |

모든 베이직 블럭들은 나누어지고 무한 루프에 삽입되었다. 그리고 프로그램 흐름은 switch와 변수 b에 의해 제어된다.

스크린샷이(가) 표시된 사진

자동 생성된 설명

평탄화 후, 아래와 같은 명령어 흐름을 얻을 수 있다.

텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

Pure C의 예시 사이에서 차이점은 IR의 버전이 완전히 평탄화 되었다는 것이다.

**2.3. 기존 연구의 문제점 및 해결 방안**

**2.3.1. 기존 연구의 문제점**

위에서 언급한 Obfuscator-LLVM[4]은 가짜 제어 흐름 추가, 의미 없는 소스코드의 삽입 등 소스코드 기반으로만 난독화를 실시한다. 소스코드 기반으로 난독화 된 바이너리 파일은 제어 흐름 그래프(Control Flow Graph) 분석을 자동화하여 쉽게 원래의 실행 흐름을 파악할 수 있다는 단점이 존재한다.

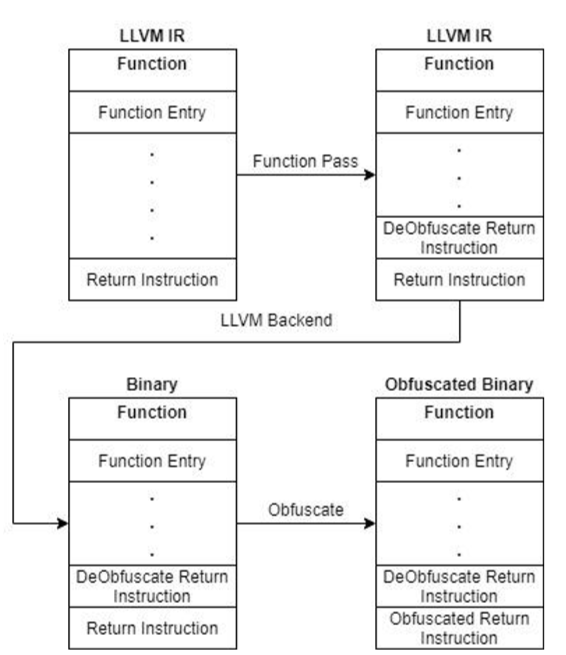
**2.3.2. 해결 방안**

제어 흐름 그래프 분석을 어렵게 하거나 바이너리 파일의 코드를 사용자가 완전히 분석을 할 수 없도록 하기 위해서는 바이너리 파일의 코드 자체를 난독화 하는 기법이 필요하다. 이를 위해 본 연구에서는 LLVM의 Pass를 이용하여 컴파일 된 바이너리 코드를 수행하기 전 난독화 해제 과정을 거친 후 코드를 수행하는 코드를 삽입한 후에 이러한 바이너리 파일의 코드를 난독화 하고자 한다.

**3. 프로젝트 내용**

**3.1. 기존 제안 방안**

**3.1.1 구조**



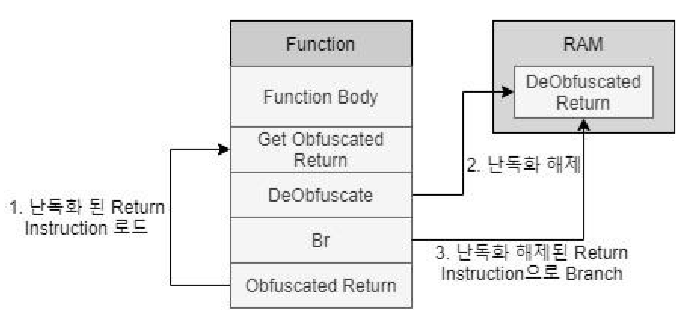
[그림 3] 바이너리 코드 난독화 및 LLVM Pass 수행 과정

베어메탈 IoT 디바이스는 대부분 ARM 아키텍처를 사용하는 CPU를 가진다. ARM 아키텍처의 경우 Instruction 집합을 단순화하여 실행한다. Instruction의 길이는 32비트로 단순화되어 있으며, 피연산자에 32비트 정수 값을 넣을 수 없어 함수 호출과 같은 분기 명령어의 경우 PC(Program Counter) 레지스터에 대한 오프셋 값을 이용해 수행된다. 따라서 분기 Instruction은 난독화를 해제 후 RAM에 해제된 난독화 코드를 올린 후 실행시킬 경우 PC레지스터와의 오프셋 값 변경으로 인한 명령어 코드의 실행이 불가능한 문제가 발생할 수 있다. 따라서 주요 Instruction 중 Return Instruction만을 난독화 하였다. ARM의 경우 함수 호출 시에 LR(Link Register) 레지스터에 되돌아갈 주소를 저장한 후 Return Instruction 실행 시에 LR 레지스터를 PC에 저장하는 방식으로 분기가 이루어지므로 Return Instruction 난독화 시에도 문제가 발생하지 않는다.   
 [그림 3]에서는 바이너리 코드를 난독화 하는 방안에 대한 전체적인 구조를 보여주며, LLVM의 Function Pass를 이용한다.   
 1. Function Entry, Return Instruction  
 - Function Entry는 각 함수의 시작 번지를 나타내고 Return Instruction은 함수가 종료할 때 마지막으로 실행되는 이전 함수 분기 명령어이다.   
 2. Deobfuscate Return Instruction

- 난독화된 Return Instruction의 난독화를 해제시키는 코드이다.  
 3. Obfuscate Return Instruction

- 난독화된 Return Instruction이다.  
 Function Pass를 작성할 경우 각 IR 코드 내에 각 Function 단위로 Code Instrumentation이 가능하다. 따라서 Function Pass를 이용해 함수 단위로 IR 레벨에서 Return Instruction 전에 난독화를 해제하는 코드를 추가한다. 그 후 LLVM의 Back-end를 이용해 ARM 바이너리 코드를 생성한다. 이후 생성된 바이너리 코드는 독립적인 난독화 프로그램을 이용해 Return Instruction에 대한 난독화를 진행한다.

**3.1.2 난독화 코드 실행**



[그림 4] 난독화 코드 실행

[그림 4]에서는 난독화 된 코드의 구조와 이러한 코드를 실행하는 순서를 보여준다. 상세한 코드 실행 순서는 다음과 같다. 첫번째 단계로 난독화 된 Return Instruction의 바이너리 코드를 읽어 RAM으로 로드 한다. 이 때 함수 호출 시에 생성된 Stack을 이용한다. 두번째 단계로 난독화 해제 연산을 통해 RAM 내의 난독화 된 Return Instruction의 난독화를 해제한다. 마지막 단계로 복구된 Return Instruction이 저장 되어있는 주소로 Branch하여 Return Instruction을 수행하고 다음 명령어를 수행한다.

**3.1.3 Pass 코드 작성**

2.1.3.1절에서 Return Instruction Obfuscation Pass를 LLVMObfuscation.so 라이브러리에 빌드하는 환경을 제시하였다. 다음은 ReturnObfuscation.cpp의 runOnFunction함수의 일부이다.

bool inserted = false;  
for (auto &BB : F) {  
 for (auto &I : BB) {  
 if(!inserted) {  
 ptr\_ret\_array = new AllocaInst(return\_array, NULL, "ret\_ptr", &I);  
 ptr\_ret\_array->setAlignment(MaybeAlign(1));  
 ptr\_this\_ret = new AllocaInst(ret\_func\_ptr, NULL, "ptr", &I);  
 ret\_array\_ptr = new AllocaInst(ret\_func\_ptr, NULL, "ptr2", &I);  
 ptr\_i = new AllocaInst(IntegerType::get(mod->getContext(), 32), NULL, "i", &I);  
   
 inserted=true;   
 }  
 if (I.getOpcode() == Instruction::Ret) {  
 instructions.push\_back(&I);  
 }  
 }  
}

Function Pass로 Pass를 작성하였고, 각각 함수내의 처음 BasicBlock의 첫 Instruction 전에 난독화된 return instruction을 가리키는 포인터, 난독화 해제한 return instruction을 저장할 배열의 포인터, 반복문 사용을 위한 i 값에 대한 포인터를 Allocation하는 Instruction을 생성한다.

그 후 함수 내 모든 Instruction을 확인하며 Return Instruction을 만날 경우 instructions vector에 push해준다.

AllocaInst에 의한 생성 결과의 예시는 다음과 같다.

%ptr = alloca [33 x i8], align 4

위 IR코드는 %ptr에 33 크기의 1바이트(8비트) 배열을 allocation해주는 코드이다.

for (auto &I : instructions) {  
 BasicBlock \*BB = I->getParent();  
 // One Instruction Basic Block has only one ret instructions  
 if (!BB->size() < 2)  
 {  
 BasicBlock \*retblock = BB->splitBasicBlock(I->getIterator(), BB->getName() + ".RetBlock");  
 RetBlocks.push\_back(retblock);  
 } else {  
 RetBlocks.push\_back(BB);  
 }   
}

위의 코드는 모든 Return Instruction을 가진 BasicBlock에서 Return Instruction만을 따로 BasicBlock을 나누는 과정이다. BasicBlock Class의 splitBasicBlock 멤버함수를 이용해 나눌 수 있으며, Basic Block의 이름을 지정해 줄 수 있다. 이 때 나눈 후 Return Instruction Basic Block의 이전 Basic Block은 Return Instruction의 Basic Block로 Branch하는 코드가 삽입된다.

이 과정은 Return Basic Block을 따로 나누어 Block의 Address를 구하여 난독화 된 Return Instruction에 접근하기 위하여 사용된다.

다음은 Basic Block의 Block Address를 가져와 Constant형태로 저장하는 코드이다.

Constant**\*** retBlockAddress **=** BlockAddress**::**get(BB);

다음은 Return Basic Block의 Block Address를 처음에 Allocation한 현재 난독화된 Return Instruction을 가리키는 포인터에 Store하는 Instruction을 삽입하는 코드이다.

StoreInst**\*** ret\_store **=** **new** StoreInst(retBlockAddress, ptr\_this\_ret, false, **&**B);

다음은 난독화 된 Return Instruction을 난독화 해제하기 위해 decryption 하는 BasicBlock을 만드는 코드이다. 이 경우 원래의 Basic Block의 predecessor(현재의 BasicBlock으로 Branch하는 Basic Block)들의 branch명령어를 지운 후 decryption하는 BasicBlock으로 branch하는 명령어를 생성시키는 코드이다.

BasicBlock의 Terminator을 불러와 삭제 후, 새로운 Terminator인 decryption하는 BasicBlock으로 branch하는 명령어를 생성시킨다.

BasicBlock\* decrypt\_start = BasicBlock::Create(mod->getContext(), "dec\_start", &F, BB);  
for (BasicBlock\* preds : predecessors(BB)) {  
 preds->getTerminator()->eraseFromParent();  
 BranchInst::Create(decrypt\_start, preds);  
}

다음 코드는 GetElementPtrInst Instruction을 통해 난독화를 해제하여 넣을 return array 배열의 0번째 위치를 가져온 후 ret\_array\_ptr에 store하는 Instruction을 생성하는 코드를 삽입한 후 return block address을 ptr\_this\_ret 포인터에 store하는 Instruction을 생성한다.

std::vector<Value\*> ptr\_to\_retarray\_indices;  
ptr\_to\_retarray\_indices.push\_back(const\_int\_0);  
ptr\_to\_retarray\_indices.push\_back(const\_int\_0);  
GetElementPtrInst\* ptr\_to\_retarray = GetElementPtrInst::Create(return\_array, ptr\_ret\_array,  
 ptr\_to\_retarray\_indices, "arrayidx", decrypt\_start);  
ptr\_to\_retarray->setIsInBounds(true);  
StoreInst\* store\_to\_ret\_ptr = new StoreInst(ptr\_to\_retarray, ret\_array\_ptr, false, decrypt\_start);  
store\_to\_ret\_ptr->setAlignment(MaybeAlign(4));  
StoreInst\* void\_17 = new StoreInst(retBlockAddress, ptr\_this\_ret, false, decrypt\_start);

StoreInst**\*** store\_i\_0 **=** **new** StoreInst(const\_int\_0, ptr\_i, false, decrypt\_start);

그 이후 난독화 해제를 위한 for문의 조건부분을 생성한다. Decrypt\_cond라는 Basic Block을 생성 후 이 BasicBlock에서는 i값 로드, i값 20과 비교 후 branch하는 명령어를 생성하게 된다.

BasicBlock**\*** decrypt\_cond **=** BasicBlock**::**Create(mod**->**getContext(), "dec\_cond", **&**F, BB);  
BranchInst**::**Create(decrypt\_cond, decrypt\_start);  
LoadInst**\*** ldr\_i\_data **=** **new** LoadInst(ptr\_i, "", false, decrypt\_cond);  
ldr\_i\_data**->**setAlignment(MaybeAlign(4));  
ICmpInst**\*** cmp\_i\_with\_20 **=** **new** ICmpInst(**\***decrypt\_cond, ICmpInst**::**ICMP\_SLT, ldr\_i\_data, const\_int\_20, "cmp");  
  
BasicBlock**\*** decrypt\_ing **=** BasicBlock**::**Create(mod**->**getContext(), "dec\_ing", **&**F, BB);  
BasicBlock\* decrypt\_add = BasicBlock::Create(mod->getContext(), "dec\_add", &F, BB);  
BasicBlock\* decrypt\_end = BasicBlock::Create(mod->getContext(), "dec\_end", &F, BB);  
BranchInst**::**Create(decrypt\_ing, decrypt\_end, cmp\_i\_with\_20, decrypt\_cond);

다음은 i값을 로드하는 instruction을 삽입 후 현재 난독화 된 return instruction의 i번째 인덱스의 포인터를 가져오는 명령어를 삽입한 결과이다. ldr\_func\_ptr\_idx에 로드한다.

LoadInst\* ldr\_i\_data\_2 = new LoadInst(ptr\_i, "", false, decrypt\_ing);  
ldr\_i\_data\_2->setAlignment(MaybeAlign(4));  
LoadInst\* ldr\_ptr\_this\_ret = new LoadInst(ptr\_this\_ret, "", false, decrypt\_ing);  
ldr\_ptr\_this\_ret->setAlignment(MaybeAlign(4));  
GetElementPtrInst\* get\_func\_ptr\_idx = GetElementPtrInst::Create(cast<PointerType>(ldr\_ptr\_this\_ret->getType()->getScalarType())->getElementType(), ldr\_ptr\_this\_ret, ldr\_i\_data\_2, "arrayidx1", decrypt\_ing);  
get\_func\_ptr\_idx->setIsInBounds(true);  
LoadInst\* ldr\_func\_ptr\_idx = new LoadInst(get\_func\_ptr\_idx, "", false, decrypt\_ing);  
ldr\_func\_ptr\_idx->setAlignment(MaybeAlign(1));

다음은 은 앞서 본 Pass 코드들을 조합하여 i값 로드 및 현재 난독화된 Return Instruction의 위치의 값 가져오기, 그 값에 특정 연산을 한 후 난독화를 해제하고, 이 값을 난독화 해제 된 Return Instruction 배열에 집어 넣는다. 그 이후 Indirect Branch명령어를 이용해 난독화 해제 된 Return Instruction 배열의 첫번째 주소로 점프 하는 IR코드를 삽입 시키게 된다.

LoadInst\* ldr\_i\_data\_3 = new LoadInst(ptr\_i, "", false, decrypt\_ing);

ldr\_i\_data\_3->setAlignment(MaybeAlign(4));

std::vector<Value\*> ptr\_retn\_array\_indices;

ptr\_retn\_array\_indices.push\_back(const\_int\_0);

ptr\_retn\_array\_indices.push\_back(ldr\_i\_data\_3);

GetElementPtrInst\* get\_retn\_array\_data\_idx = GetElementPtrInst::Create(return\_array, ptr\_ret\_array, ptr\_retn\_array\_indices, "arrayidx2", decrypt\_ing);

get\_retn\_array\_data\_idx->setIsInBounds(true);

StoreInst\* str\_retn\_array\_data\_idx = new StoreInst(ldr\_func\_ptr\_idx, get\_retn\_array\_data\_idx, false, decrypt\_ing);

str\_retn\_array\_data\_idx->setAlignment(MaybeAlign(1));

LoadInst\* ldr\_i\_data\_4 = new LoadInst(ptr\_i, "", false, decrypt\_ing);

ldr\_i\_data\_4->setAlignment(MaybeAlign(4));

std::vector<Value\*> ptr\_retn\_array\_indices2;

ptr\_retn\_array\_indices2.push\_back(const\_int\_0);

ptr\_retn\_array\_indices2.push\_back(ldr\_i\_data\_4);

GetElementPtrInst\* get\_retn\_array\_data\_idx2 = GetElementPtrInst::Create(return\_array, ptr\_ret\_array, ptr\_retn\_array\_indices2, "arrayidx3", decrypt\_ing);

get\_retn\_array\_data\_idx2->setIsInBounds(true);

LoadInst\* ldr\_retn\_array\_data\_idx2 = new LoadInst(get\_retn\_array\_data\_idx2, "", false, decrypt\_ing);

ldr\_retn\_array\_data\_idx2->setAlignment(MaybeAlign(1));

CastInst\* cast\_retn\_array\_data\_idx2 = new ZExtInst(ldr\_retn\_array\_data\_idx2, IntegerType::get(mod->getContext(), 32), "conv", decrypt\_ing);

BinaryOperator\* xor\_retn\_array\_data\_idx2 = BinaryOperator::Create(Instruction::Xor, cast\_retn\_array\_data\_idx2, const\_int32\_133, "xor", decrypt\_ing);

CastInst\* trun\_retn\_array\_data\_idx2 = new TruncInst(xor\_retn\_array\_data\_idx2, IntegerType::get(mod->getContext(), 8), "conv4", decrypt\_ing);

LoadInst\* ldr\_i\_data\_5 = new LoadInst(ptr\_i, "", false, decrypt\_ing);

ldr\_i\_data\_5->setAlignment(MaybeAlign(4));

std::vector<Value\*> ptr\_retn\_array\_indices4;

ptr\_retn\_array\_indices4.push\_back(const\_int\_0);

ptr\_retn\_array\_indices4.push\_back(ldr\_i\_data\_5);

GetElementPtrInst\* get\_retn\_array\_data\_idx4 = GetElementPtrInst::Create(return\_array, ptr\_ret\_array, ptr\_retn\_array\_indices4, "arrayidx5", decrypt\_ing);

get\_retn\_array\_data\_idx4->setIsInBounds(true);

StoreInst\* str\_retn\_array\_data\_idx4 = new StoreInst(trun\_retn\_array\_data\_idx2, get\_retn\_array\_data\_idx4, false, decrypt\_ing);

str\_retn\_array\_data\_idx4->setAlignment(MaybeAlign(1));

BranchInst::Create(decrypt\_add, decrypt\_ing);

LoadInst\* ldr\_i\_data\_6 = new LoadInst(ptr\_i, "", false, decrypt\_add);

ldr\_i\_data\_6->setAlignment(MaybeAlign(4));

BinaryOperator\* add\_i\_data\_4 = BinaryOperator::Create(Instruction::Add, ldr\_i\_data\_6, const\_int\_1, "", decrypt\_add);

StoreInst\* str\_i\_data\_4 = new StoreInst(add\_i\_data\_4, ptr\_i, false, decrypt\_add);

str\_i\_data\_4->setAlignment(MaybeAlign(4));

BranchInst::Create(decrypt\_cond, decrypt\_add);

LoadInst\* ldr\_ret\_array = new LoadInst(ret\_array\_ptr, "", false, decrypt\_end);

ldr\_ret\_array->setAlignment(MaybeAlign(4));

BasicBlock\* dec\_jmp = BasicBlock::Create(mod->getContext(), "dec\_jmp", &F, BB);

BranchInst::Create(dec\_jmp, decrypt\_end);

PHINode\* ptr\_40 = PHINode::Create(ret\_func\_ptr, 1, "", dec\_jmp);

ptr\_40->addIncoming(ldr\_ret\_array, decrypt\_end);

IndirectBrInst \*void\_41 = IndirectBrInst::Create(ldr\_ret\_array, 1, dec\_jmp);

void\_41->addDestination(BB);

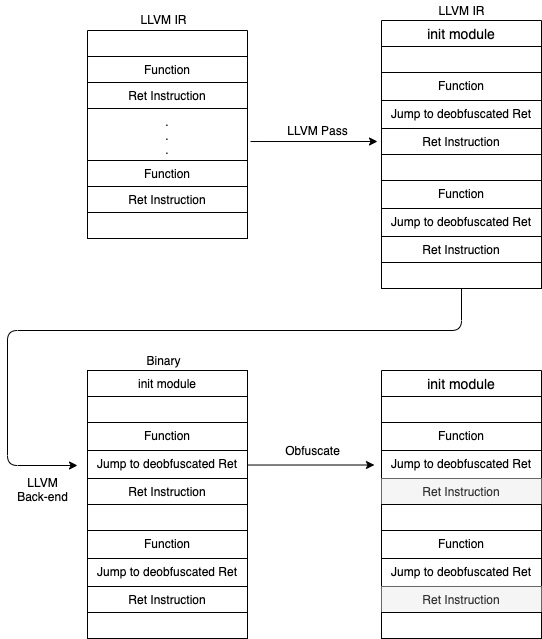
**3.1.4 문제점**

위의 방법에는 오버헤드가 증가하는 문제점이 존재한다. 함수 단위로 Return Instruction의 코드를 RAM으로 로드 한 후 Return Instruction을 수행할 때마다 난독화 해제 연산을 수행할 경우 함수 호출시마다 특정 연산을 수행해야 하기 때문에 많은 오버헤드가 발생한다. 이를 해결하기 위해 IoT 디바이스의 시작 시에 난독화를 미리 해제한 후 함수 호출시에 이러한 코드를 실행시키는 방안이 필요하다

.

**3.2. 제안 방안**

**3.2.1 구조**



[그림 5] 변경된 바이너리 코드 난독화 및 LLVM Pass 수행 과정

[그림 5]는 LLVM을 이용한 바이너리 코드를 난독화 하는 새로운 방안을 제시한다. 우선 각각 바이너리에 존재하는 모든 함수들의 난독화 된 Return Instruction을 난독화 해제시킨 후 SRAM의 전역변수 영역에 삽입하는 initialization 모듈을 함수 형태로 삽입시킨 후 main 함수에서 이를 호출하도록 한다. 그 후 Function Pass를 이용해 각각 함수별로 난독화가 해제된 Return Instruction으로 점프 시키는 코드를 삽입한다. 그 후 난독화 프로그램을 통해 바이너리 파일의 Return Instruction들을 모두 난독화 시킨다.

**3.2.2 난독화 코드 실행**

**스크린샷이(가) 표시된 사진

자동 생성된 설명**

[그림 6] 변경된 난독화 코드 수행 과정

[그림 6]에서는 변경된 난독화 코드의 구조와 이러한 코드를 실행하는 순서를 보여준다. 상세한 코드 실행 순서는 다음과 같다. 우선 Initialization 모듈에서 모든 함수들의 Return Instruction을 난독화 해제하여 SRAM의 전역변수 영역에 Return Table의 형태로 삽입한다. 그 후 각각의 함수 실행 시에 원래의 Return Instruction을 수행시키는 것이 아닌 Return Table에 존재하는 난독화 해제된 Return Instruction으로 분기한 후 수행시킨다.

**3.2.2 Pass 코드 작성**

[그림 5]와 [그림 6]의 새로운 방식으로 난독화 해제를 수행하기 위해 두가지 Pass를 작성하였다. 존재하는 함수들의 이름을 모두 받아오고 Return Instruction Basic Block을 Split하는 PreProcess Pass 및 난독화 해제 Initialization Module을 삽입하고, 각 함수별로 전역변수의 위치로 분기하는 코드를 삽입하는 Obfuscation Pass를 작성하였다.

**3.2.2.1 PreProcess Pass** PreProcess Pass에서는 functions.txt 파일 내에 각각 함수들의 이름을 저장하고, Return Instruction을 기준으로 SplitBasicBlock을 통해 Return Instruction만 들어가 있는 Basic Block을 “obfuscated return”이라는 이름을 생성한다.

std::vector<Instruction \*> instructions;

std::vector<BasicBlock \*> RetBlocks;

bool inserted = false;

std::ofstream functionFile("functions.txt", std::ios\_base::app);

if (functionFile.is\_open()) {

if (!F.getName().contains("\_\_cxx") && !F.getName().contains("\_GLOBAL"))

functionFile << F.getName().str() << "\n";

functionFile.close();

}

if (!F.getName().contains("\_\_cxx") && !F.getName().contains("\_GLOBAL")) {

for (auto &BB : F) {

for (auto &I : BB) {

if (I.getOpcode() == Instruction::Ret) {

instructions.push\_back(&I);

}

}

}

for (auto &I : instructions) {

BasicBlock \*BB = I->getParent();

// One Instruction Basic Block has only one ret instructions

if (!BB->size() < 2)

{

BasicBlock \*retblock = BB->splitBasicBlock(I->getIterator(), "obfuscatedreturn");

} else {

BB->setName("obfuscatedreturn");

}

}

}

**3.2.2.1 Obfuscation Pass**

다음은 Obfuscation 초반부의 코드이다. 우선 functions.txt파일에서 각각의 함수들을 불러온다. 그 후에 각 함수들의 난독화 된 Return Block의 Address를 불러와 vector에 입력한다.

function\_list.open("functions.txt");

std::vector<Constant \*> retblocks;

std::vector<Function \*> functions;

std::vector<Function \*> functions2;

std::string line;

while (getline(function\_list, line)) {

functions.push\_back(mod->getFunction(line));

}

count = 0;

bool inserted = false;

// 함수 별로 벡터에 집어넣기

for (auto &Fn : functions) {

inserted = false;

for (auto &BB : (\*Fn)) {

if (BB.getName().equals("obfuscatedreturn")) {

Constant \*retBlockAddress = BlockAddress::get(&BB);

retblocks.push\_back(retBlockAddress);

if (!inserted)

functions2.push\_back(Fn);

inserted = true;

}

}

}

for (auto &Fn : functions2) {

if (Fn->getName().equals(F.getName())) {

num\_this\_function = count;

}

count++;

}

num\_retblocks = retblocks.size();

다음은 Initialization Module인 난독화 해제 함수를 생성하는 과정이다. 이 과정은 이전의 제안방안과 같이 각각 함수에 해당하는 전역변수에 각각 함수별로 난독화된 Return Instruction을 난독화 해제하여 전역변수에 저장하는 IR코드를 생성시킨다.

Function \*Func\_deobfus = mod->getFunction("func\_deobfus");

if (!Func\_deobfus) {

Func\_deobfus = Function::Create(

Func\_deobfus\_type, GlobalValue::ExternalLinkage, "func\_deobfus", mod);

Func\_deobfus->setCallingConv(CallingConv::C);

}

if (Func\_deobfus->size() == 0) {

GlobalVariable \*gvar\_ret\_inst\_list =

new GlobalVariable(\*mod, array\_out, false, GlobalValue::CommonLinkage,

0, "ret\_inst\_list");

gvar\_ret\_inst\_list->setAlignment(MaybeAlign(16));

ConstantAggregateZero \*const\_array\_6 = ConstantAggregateZero::get(array\_out);

gvar\_ret\_inst\_list->setInitializer(const\_array\_6);

/\* 난독화된 Return Instruction들의 난독화를 해제한 후 전역변수 부분에 저장 \*/

}

다음 코드는 Initialization Module인 func\_deobfus 함수를 main함수의 맨 처음에 호출하는 Instruction을 생성하는 과정이다.

if (F.getName().equals("main")) {

for (auto &BB : F) {

for (auto &I : BB) {

CallInst \*int32\_25 = CallInst::Create(Func\_deobfus, "", &I);

int32\_25->setCallingConv(CallingConv::C);

int32\_25->setTailCall(false);

AttributeList int32\_25\_PAL;

int32\_25->setAttributes(int32\_25\_PAL);

break;

}

break;

}

}

다음 코드는 각각 함수에 해당하는 전역변수의 난독화 해제된 Return Instruction이 존재하는 포인터로 분기하는 명령어를 삽입한다. 분기 방법은 이전 코드와 유사하다.

if (!F.getName().contains("func\_deobfus") &&

!F.getName().contains("\_\_cxx") && !F.getName().contains("\_GLOBAL")) {

for (auto &BB : F) {

if (BB.getName().equals("obfuscatedreturn")) {

GlobalVariable \*gvar\_ret\_inst\_list =

mod->getGlobalVariable("ret\_inst\_list");

PointerType \*PointerTy\_31 =

PointerType::get(IntegerType::get(mod->getContext(), 8), 0);

BasicBlock \*jmp\_to =

BasicBlock::Create(mod->getContext(), "jmp\_to", &F, &BB);

BasicBlock \*ret\_jmp =

BasicBlock::Create(mod->getContext(), "ret\_jmp", &F, &BB);

AllocaInst \*ptr\_ptr3 =

new AllocaInst(PointerTy\_31, NULL, "ret\_ptr\_jmp", jmp\_to);

ptr\_ptr3->setAlignment(MaybeAlign(8));

for (BasicBlock \*preds : predecessors(&BB)) {

preds->getTerminator()->eraseFromParent();

BranchInst::Create(jmp\_to, preds);

}

std::vector<Constant \*> const\_ptr\_14\_indices;

const\_ptr\_14\_indices.push\_back(ConstantInt::get(

mod->getContext(), APInt(64, StringRef("0"), 10)));

const\_ptr\_14\_indices.push\_back(ConstantInt::get(

mod->getContext(),

APInt(64, StringRef(std::to\_string(num\_this\_function)), 10)));

const\_ptr\_14\_indices.push\_back(ConstantInt::get(

mod->getContext(), APInt(64, StringRef("0"), 10)));

Constant \*const\_ptr\_14 = ConstantExpr::getGetElementPtr(

array\_out, gvar\_ret\_inst\_list, const\_ptr\_14\_indices);

StoreInst \*void\_177 =

new StoreInst(const\_ptr\_14, ptr\_ptr3, false, jmp\_to);

void\_177->setAlignment(MaybeAlign(8));

LoadInst \*ptr\_178 = new LoadInst(ptr\_ptr3, "", false, jmp\_to);

ptr\_178->setAlignment(MaybeAlign(8));

BranchInst::Create(ret\_jmp, jmp\_to);

PHINode \*ptr\_181 = PHINode::Create(PointerTy\_31, 1, "", ret\_jmp);

ptr\_181->addIncoming(ptr\_178, jmp\_to);

IndirectBrInst \*void\_182 =

IndirectBrInst::Create(ptr\_181, 1, ret\_jmp);

void\_182->addDestination(&BB);

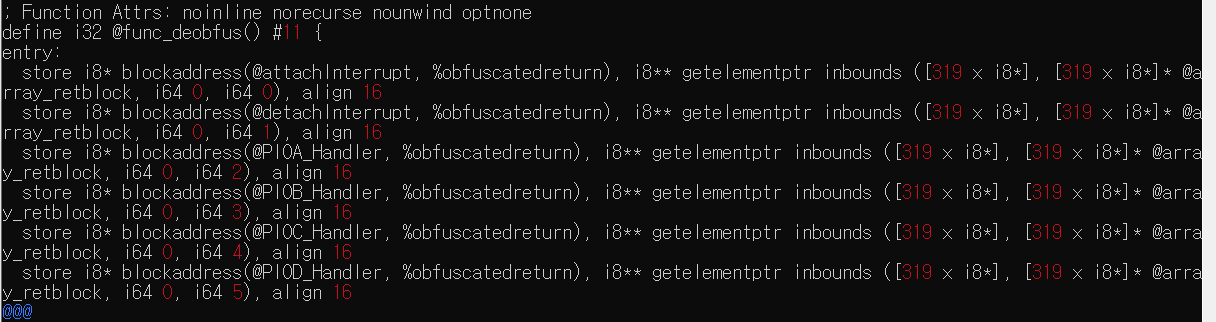
}

}

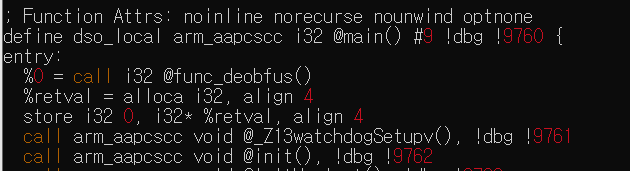
}

**4. 프로젝트 결과**

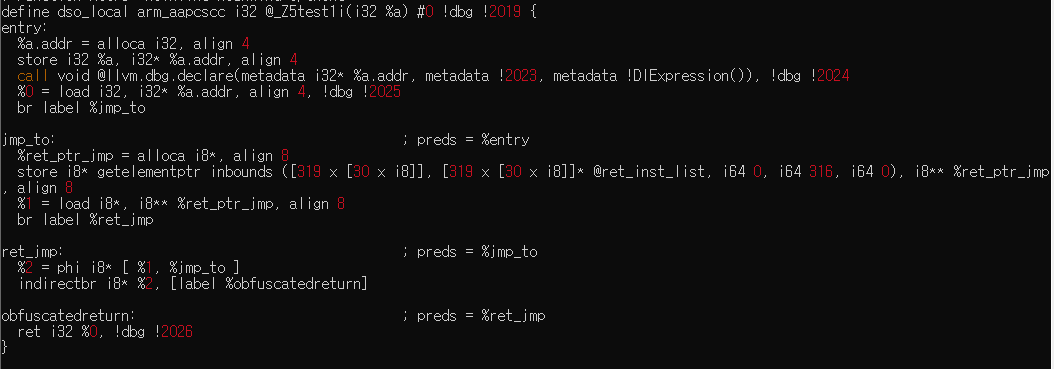
**4.1. 연구 결과**

****

[그림 7] 삽입된 Initialization Module



[그림 8] Initialization Module 수행



[그림 9] 난독화 해제된 Return Instruction으로 분기

이번 프로젝트를 통해 LLVM 컴파일러의 Pass 작성을 통해 베어메탈 IoT 디바이스의 펌웨어 바이너리 파일의 Return Instruction들을 난독화 하는 과정을 제안하였다. [그림 7]은 삽입된 Initialization 모듈(func\_deobfus)을 보여준다. 바이너리 파일의 각각의 함수들의 Return Instruction Block의 주소를 불러온 후 각각 난독화를 해제하여 ret\_inst\_list라는 Return Table에 입력하는 것을 알 수 있다. [그림 8]은 func\_deobfus 함수를 main함수에서 호출하는 코드를 보여준다. [그림 9]에서는 각각 함수의 Return Instruction을 수행할 때 ret\_inst\_list Return Table을 참조하여 이 곳으로 분기하여 난독화가 해제된 Return Instruction을 수행하는 것을 알 수 있다.

**5. 결론**

**5.1. 기대효과**

IoT 디바이스의 펌웨어 바이너리 코드의 Return Instruction의 난독화를 수행하였다. 이를 통해 일반 사용자가 역 공학 기법을 통해 정적 분석이 어렵도록 할 수 있으며, return instruction이 난독화 되기 때문에 ROP(Return Oriented Programming)공격 기법 방지에도 도움이 될 것이다.

**5.2. 추후 연구 방향**

현재는 Return Instruction의 난독화만 수행한다. 하지만 이후에는 난독화 해도 프로그램 수행에 영향을 받지 않는 다른 Instruction에 대한 난독화도 수행하여 더욱더 정적 분석이 어렵도록 할 수 있다. 또한 난독화 시에 난독화 해제 코드를 정적분석을 통해 확인할 수 있어 어떠한 방식으로 난독화 되어있는지 파악이 가능하다는 단점이 존재한다. 이는 HSM이나, ARM TrustZone 기술 등을 통해 난독화 해제 연산을 안전한 영역에서 수행하여 사용자가 코드를 확인할 수 없도록 할 수 있다.

**6. 참고문헌**

[1] 길민권, "최근 5년 간 신고된 IoT 보안취약점 1,400 건, 포상금은 4억 1천만원 넘어", 데일리시큐, 2019.

[2] Jung Wook (Matt) Oh, "Reverse Engineering Flash Memory for Fun and Benefit", Blackhat US, 2014.

[3] Roman Rohleder, "Hands-On Ghidra – A Tutorial about the Software Reverse Engineering Framework", Proceedings of the 3rd ACM Workshop on Software Protection, 2019.

[4] Junod, P, Rinaldini J, Wehrli J, Michielin J, "Obfuscator-LLVM: software protection for the masses", In Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, 2015.

[5] T. Benavides, J. Treon, J. Hulbert, W. Chang, "The Enabling of an Execute-In-Place Architecture to Reduce the Embedded System Memory Footprint and Boot Time", Journal of Computers, Vol. 3, No. 1, pp. 79-89, 2008.

[6] C. Lattner and V. Adve. "LLVM: A compilation framework for lifelong program analysis & transformation." In Proceedings of the 2004 International Symposium on Code Generation and Optimization, Palo Alto, California, March, 2004.