|  |
| --- |
| **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.

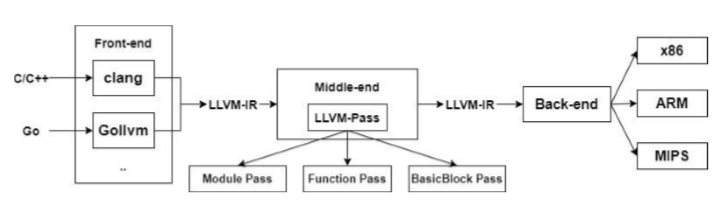
소프트웨어의 코드 정적 분석을 어렵게 하기 위해 Obfuscator-LLVM[4]에서는 다양한 소스코드 기반 난독화 기법을 제안하였다. 하지만 이러한 소스코드 기반 난독화 기법은 제어 흐름 그래프(Control Flow Graph)의 분석을 통해 쉽게 난독화 해제가 가능하다.

**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.

LLVM Pass는 LLVM 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.

**2.2. Obfuscator-LLVM**

Obfuscator-LLVM[4]은 LLVM으로 작성된 오픈소스 코드 난독화 컴파일러이다. LLVM Pass를 이용해 작성되었다. 소프트웨어의 역공학을 통한 분석의 속도를 늦추기 위해 개발되었으며, 많은 곳에서 사용되고 있다. ADD, XOR, OR와 같은 연산을 동일한 결과가 나오는 다양한 연산들로 조합하여 분석이 어렵게 하거나, 기존 제어 흐름(Control Flow)에 가짜 제어 흐름을 추가하여 새로운 코드를 집어넣는 등 다양한 소스코드 기반의 난독화 기법을 사용하여 소프트웨어 사용자의 역공학을 통한 정적 분석을 어렵게 한다.

**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.

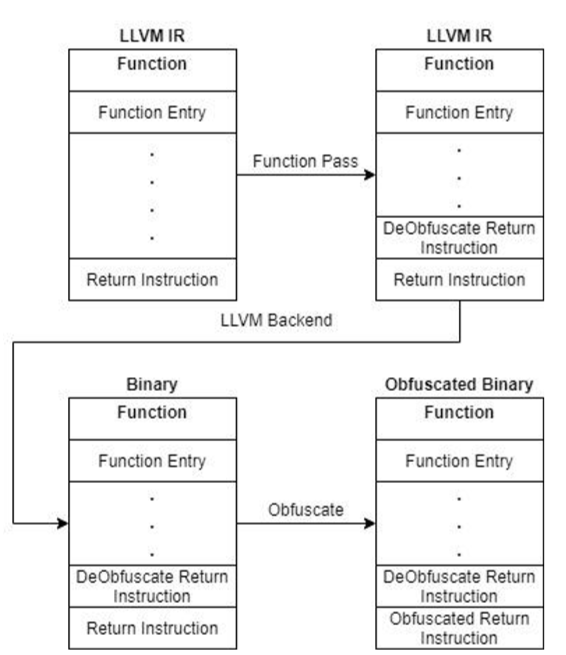
**2.3.2. 해결 방안**

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**



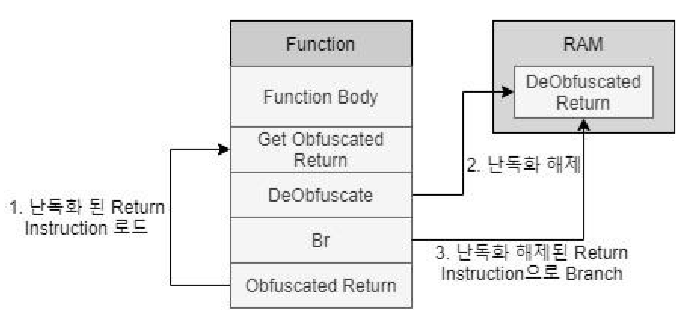
[그림 2] 바이너리 코드 난독화 및 LLVM Pass 수행 과정

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 2]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 3] 난독화 코드 실행

[그림 3]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.

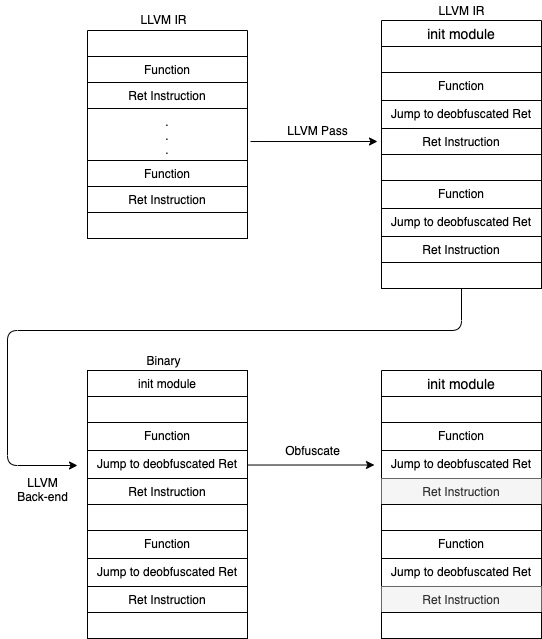
**3.1.3 문제점**

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 4] 변경된 바이너리 코드 난독화 및 LLVM Pass 수행 과정

[그림 4]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

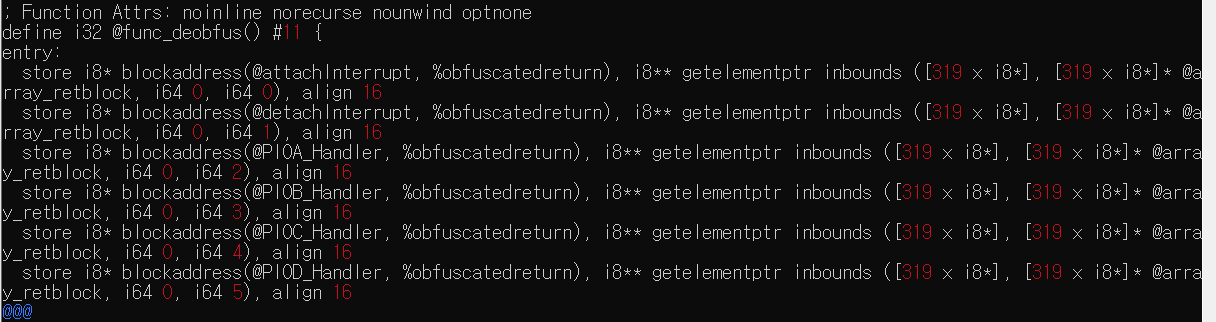
자동 생성된 설명**

[그림 5] 변경된 난독화 코드 수행 과정

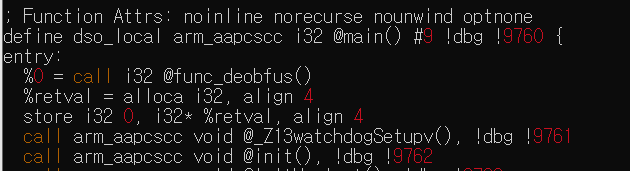
[그림 5]에서는 변경된 난독화 코드의 구조와 이러한 코드를 실행하는 순서를 보여준다. 상세한 코드 실행 순서는 다음과 같다. 우선 Initialization 모듈에서 모든 함수들의 Return Instruction을 난독화 해제하여 SRAM의 전역변수 영역에 Return Table의 형태로 삽입한다. 그 후 각각의 함수 실행 시에 원래의 Return Instruction을 수행 시키는 것이 아닌 Return Table에 존재하는 난독화 해제된 Return Instruction으로 분기한 후 수행시킨다.

**4. 프로젝트 결과**

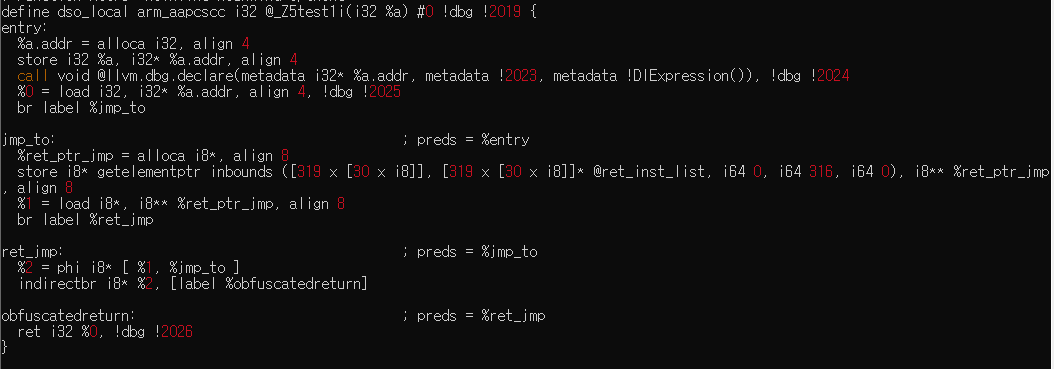
**4.1. 연구 결과**

****

[그림 6] 삽입된 Initialization Module



[그림 7] Initialization Module 수행



[그림 8] 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 6]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 7]은 func\_deobfus 함수를 main함수에서 호출하는 코드를 보여준다. [그림 8]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.