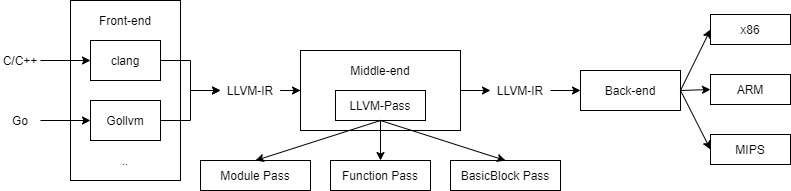
|  |
| --- |
| **LLVM을 기반의 BareMetal IoT 디바이스 펌웨어 난독화** |
| 박우진 이한솔 조진성  경희대학교 컴퓨터공학과  amdx1254@khu.ac.kr, mardi@khu.ac.kr, chojs@khu.ac.kr |
| LLVM-based Firmware Obfuscation on BareMetal IoT Device |
| Woojin Park Hansol Lee Jinsung Cho  Department of Computer Science and Engineering, KyungHee University |
| **요 약**  최근 다양한 IoT 디바이스의 사용이 증가되고 있다. 그로 인한 보안 위협도 증가되고 있다. 베어메탈(BareMetal) IoT 디바이스의 펌웨어는 펌웨어 업데이트 파일 및 Flash De-soldering 등으로 추출이 가능하며, 이를 역공학(Reverse Engineering) 툴을 이용한 정적 분석을 통해 실행 흐름을 분석하여 취약점을 찾을 수 있게 된다. 이러한 정적 분석을 어렵게 하기 위해 다양한 소스코드 기반의 난독화 방법이 존재한다. 하지만 베어메탈 IoT 디바이스의 펌웨어 바이너리 코드 자체를 난독화 할 수 있는 방안은 존재하지 않는다. 이를 해결하기 위해 LLVM Pass를 이용한 난독화된 바이너리 코드를 실행하는 코드의 삽입을 통해 베어메탈 IoT 디바이스 펌웨어의 바이너리 코드를 난독화 하는 방안 제안하며, 이를 구현한다. | | |

**1. 서 론**



[그림 1] LLVM 구성도

최근 들어 IoT 기술을 적용한 가전제품을 포함한 여러 기기 사용이 증가되고 있다. 보편화된 만큼 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 디바이스들은 역공학을 통한 펌웨어의 정적 분석을 통해 코드의 흐름을 파악할 수 있으며 이를 위한 도구도 존재한다. Ghidra, IDA Pro와 같은 프로그램은 역공학을 통해 바이너리 파일을 손쉽게 분석할 수 있도록 해주며 역컴파일(Decompile)도 가능하게 해주어 누구나 쉽게 펌웨어의 분석이 가능하다. 이처럼 4차 산업 혁명에 대비할 수 있도록 IoT 디바이스의 보안 문제는 반드시 해결해야 할 숙제이다.

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 분석이 가능하다.

본 논문에서는 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)[2]기술을 이용하여 실행된다. 실행 코드가 RAM에 복사되어 실행되지 않기 때문에 펌웨어 바이너리 코드 전체를 난독화 하여 수행할 경우 펌웨어 코드가 RAM 전체를 이용하게 되는 오버헤드가 크게 발생할 수 있다. 이를 해결하기 위해 펌웨어 바이너리 코드의 주요 instruction만 난독화 하여 RAM에 불러온 후 실행하는 방안을 제시한다.

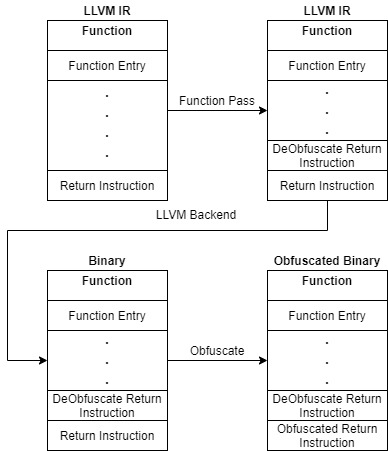
본 논문의 구성은 다음과 같다. 2장에서는 LLVM과 Obfuscator-LLVM에 대해 소개한다. 3장에서는 펌웨어의 바이너리 코드 난독화 방안에 대해 제시하고 4장에서는 실제 ARM Cortex-M3에 적용해보고 구현 결과를 제시한다. 마지막으로 5장에서는 결론을 맺고 향후 연구 방향을 제시한다.

**2. 관련 연구**

**2.1 LLVM**

LLVM (Low Level Virtual Machine)은 Chris Lattner 에 의해 작성된 컴파일러 프로젝트이다[3]. 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 어셈블리와 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 루프를 제거하는 일을 하며 이 과정은 LLVM의 Pass를 이용한 code instrumentation을 통해 이루어진다. LLVM Pass를 이용해 새로운 코드의 삽입도 가능하며 코드의 변환도 가능하다. LLVM Pass는 [그림 1]과 같이 Function, Module, Basic Block 등의 단위로 code instrumentation을 수행할 수 있다. Back-end에서는 Middle-end에서 변환된 IR 코드를 타겟의 아키텍처에 맞는 기계어 바이너리 코드로 변환해준다. 이 과정에서도 다양한 컴파일러 상의 최적화가 가능하다.

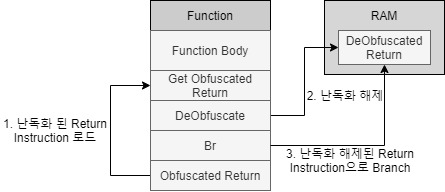


[그림 2] 바이너리 코드 난독화

본 논문에서 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 집어넣거나, 하나의 Basic Block을 여러 개로 나누는 등 다양한 소스코드 기반의 난독화 기법을 사용하여 소프트웨어 사용자의 역공학을 통한 정적 분석을 어렵게 한다.



[그림 3] 난독화 된 코드 실행

본 논문에서는 위 연구와 같이 소스코드 기반의 난독화 기법이 아닌 바이너리 코드 자체를 난독화 하는 방안을 제시한다.

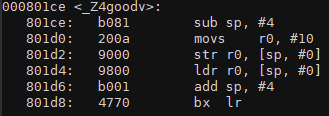
**3. 제안하는 방안**

본 절에서는 베어메탈 IoT 디바이스 펌웨어의 바이너리 코드를 난독화 하는 방안을 제시한다.

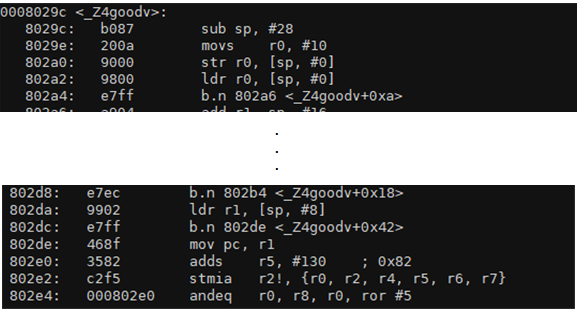
**3.1 구조**

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 명령어의 경우 PC(Program Counter) 레지스터에 대한 오프셋 값을 이용해 수행된다. 따라서 함수 호출 및 특정 주소로의 분기와 같은 Instruction는 난독화 할 경우 RAM에 불러와서 RAM으로 분기 후 실행 시 문제가 발생할 수 있다. 따라서 주요 Instruction 중 Return Instruction을 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 이루어져 난독화 시에 문제가 발생하지 않는다.

[그림 2]에서는 바이너리 코드를 난독화 하는 방안에 대한 전체적인 구조를 보여준다. LLVM의 Function Pass를 이용한다. Function Pass를 작성할 경우 각 IR 코드 내에 각 Function 단위로 Code Instrumentation이 가능하다. Function Pass를 이용해 IR 레벨에서 함수 단위로 난독화를 해제하는 코드를 추가한다. 그 후 LLVM의 Back-end를 이용해 ARM 바이너리 코드를 생성한다. 생성된 바이너리 코드는 Python 프로그램을 이용해 난독화 할 Return Instruction을 찾아 난독화 연산을 이용해 난독화 한다.



[그림 4] 난독화 전 코드



[그림 5] 난독화 결과 코드

**3.2 난독화 코드 실행**

[그림 3]에서는 삽입된 코드의 구조와 난독화 된 코드를 실행하는 순서를 보여준다. 삽입된 코드는 다음과 같다. 우선 난독화 된 Return Instruction의 코드를 RAM으로 로드 한다. 이 때 함수 호출 시에 생성된 Stack을 이용한다. 그 후 난독화 해제 연산을 통해 RAM 내의 난독화 된 Return Instruction을 복구한다. 그 후 복구된 Return Instruction의 주소로 Branch하여 Return Instruction을 수행하고 다음 명령어를 수행한다.

**4. 난독화 결과**

난독화된 바이너리 코드를 실제로 테스트하기 위해 ARM Cortex-M3 아키텍처 기반의 보드인 Arduino Due 보드를 이용해 테스트를 수행하였다. 난독화 결과를 확인하기 위해 objdump를 이용하여 어셈블리 코드를 확인하였다.

[그림 4], [그림 5]는 각각 good 함수가 난독화가 되기 전의 코드와 난독화가 된 후의 코드를 보여준다. good함수는 [그림 4]와 같이 정수 10을 반환하는 함수이며, [그림 5]에서 볼 수 있듯이 스택 복원 및 LR 레지스터 주소로 점프하는 명령어 4바이트가 난독화 된 것을 알 수 있다. 또한 [그림 5]에서 r1레지스터에 난독화가 해제된 Return Instruction 코드가 존재하는 스택 상의 주소를 넣고, PC에 r1레지스터의 값을 넣어 Return Instruction 코드를 실행할 수 있는 것을 알 수 있다.

**5. 결론 및 향후 계획**

본 논문에서는 LLVM을 이용하여 베어메탈 IoT 디바이스의 펌웨어의 바이너리 코드를 난독화 하는 방안을 제안하고 실제로 구현하였다. LLVM Pass를 이용한 Code Instrumentation을 통해 난독화 된 Return Instruction의 바이너리 코드를 복구하고 RAM에서 실행시키는 코드를 삽입하는 Pass를 작성하였다. 또한 ARM Cortex-M3 아키텍처를 사용하는 보드인 Arduino Due보드를 이용하여 실제로 난독화를 수행하고 테스트도 해보았다. 이를 통해 펌웨어 코드의 정적 분석 시 특정 함수가 반환하는 데이터를 알기 어렵게 보호할 수 있다.

본 논문에서는 Return Instruction의 바이너리 코드만 난독화 하였지만 이후에는 난독화 가능한 다른 Instruction의 바이너리 코드도 난독화 할 수 있으며, 이를 통해 함수에서 반환되는 데이터 뿐만 아니라 함수의 제어 흐름도 분석하기 어렵게 할 수 있다.

**참고 문헌**

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

[2] 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.

[3] 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.

[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.