임베디드 시스템의 보안성 향상을 위한 LLVM 기반의 소스코드 난독화 도구 설계

하재현[1], 곽동규[1]
[1]쿤텍
jhyun@coontec.com, kwak@coontec.com

Design of Source Code Obfuscation Tool based LLVM to improve security in Embedded System

Jae-Hyun Ha[1], Donggyu Kawk[1]
[1]Coontec

요 약

임베디드 시스템이 일상생활 및 각종 산업에 밀접하게 연관되어 개인 정보 및 국가 기술 등 지적 자산에 대한 보안의 필요성이 나타나고 있다. 이러한 문제점은 임베디드 시스템에 들어가는 소프트웨어의 역공학으로부터 초래된다. 따라서 본 논문은 소스 코드에 대해 제어 흐름 평탄화라는 난독화 알고리즘을 설계하는 방법을 제안한다. 이는 독자적으로 작성된 난독화 알고리즘이기 때문에 오픈 소스로 공개되어져 있는 다른 난독화 도구들에 비해 안전한 특징을 가진다. 제어 흐름 평탄화는 프로그램의 기능을 유지하면서 소스 코드의 정적 분석을 어렵게 하는 기법으로, 데이터를 탈취하려는 악의적인 행위를 사전에 예방할 수 있다.

본 논문에서 제안하는 제어 흐름 평탄화 알고리즘은 하나의 기본 블록으로 이루어진 단순한 소스 코드를 여러 개의 기본 블록으로 분할하고, 조건문을 통해 연결하는 방법을 사용하여 알고리즘의 복잡도를 높였다. 이처럼 새롭게 작성된 Pass를 통해 소스코드 난독화를 적용시켜 임베디드 시스템의 보안성을 향상시킬 수 있다.

1. 서론

임베디드 시스템은 IoT 제품뿐만 아니라 각종 산업 분야에서 일상생활과 밀접하게 연관되어 있다. 이는 임베디드 시스템의 소프트웨어 소스 코드 해킹으로 개인 정보, 국가 기술 등 중요 자료의 유출로 이어질 수 있다. 즉, 임베디드 소프트웨어의 소스 코드 역공학으로 인한 피해를 방지하기 위한 보안 기술의 도입이 필요하다.

임베디드 시스템은 대부분 C 언어로 프로그래밍 되어져 있다[1]. C언어는 어셈블리 언어 수준 정도로 하드웨어를 제어할 수 있으며, 포인터 기능을 사용하여 효율적인 메모리 관리가 가능하다. 또한 이식성이 뛰어나 다양한 플랫폼에서 사용할 수 있기 때문에 현재에도 임베디드 시스템 개발에 있어 C언어의 선호도가 상당히 높다.

본 논문은 임베디드 시스템의 소스 코드를 보호하기 위한 난독화 방법 중 제어 흐름 평탄화 방법을 제안한다. 제어 흐름 평탄화는 프로그램의 기능은 유지하되, 핵심 알고리즘의 복잡도를 증가시켜 정적 분석에 많은 시간을 소모하게 하는 난독화 기법이다. 본 논문에서는 제어 흐름 평탄화 기법을 설계하기 위해 LLVM을 사용한다. LLVM은 Clang을 통해 C/C++ 언어를 컴파일하여 중간언어(IR)를 생성한다. 제어 흐름 평탄화 난독화는 IR을 조작하는 Pass를 작성하여 구현이 가능하다.

기존에 존재하고 있는 LLVM 기반의 난독화 도구는 오픈 소스로 공개되어 있기 때문에 보안을 중요시하는 임베디드 시스템에 적용하기 적합하지 않다. 따라서 본 논문은 LLVM 기반의 제어 흐름 평탄화를 구현하는 새로운 Pass 설계 방법을 제안한다.

2. 관련 연구

LLVM 아키텍쳐는 프론트엔드, 미들엔드, 백엔드로 구성되어져 있다. 프론트엔드는 Scanner와 Parser의 기능을 하여 소스 코드를 어휘 및 구문 분석하고 중간언어인 IR을 생성한다. 미들엔드는 Pass를 통해 프론트 엔드에서 생성된 IR을 최적화한다[2]. 백엔드

는 최적화된 IR을 target machine에 맞는 기계어로 변환한다.

LLVM의 IR은 이식성이 높은 어셈블리로 설계되어 있어 C언어 기반 소프트웨어의 난독화 설계에 적합하다. IR은 소스코드의 모든 정보를 담고 있는 Module, 함수의 동작에 관여하는 Fuction, 중괄호로 표현된 함수 내 기본 블록인 Basic Block, 각종 명령어를 의미하는 Instructon가 계층구조를 이루고 있다. 그리고 IR은 Pass를 통해 소스 코드를 읽고 조작할 수 있다. 이 특성을 이용하여 제어 흐름 평탄화 Pass를 작성하여 소스 코드를 난독화할 수 있다.

Clang은 C/C++ 언어 등을 위한 컴파일러로 LLVM을 백엔드로 사용하는 프론트엔드이다. 고급 언어를 Clang 프론트엔드에서 컴파일한 후 미들엔드인 opt 모듈에서 IR을 조작하는 Pass를 작성하여 임베디드 시스템에 사용되는 코드의 난독화가 가능하다.

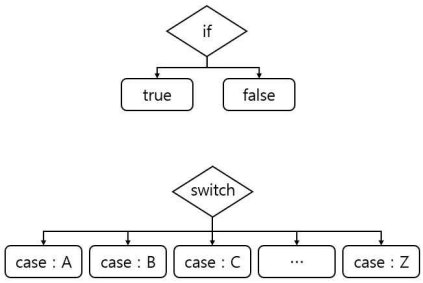
Obfuscator LLVM은 LLVM 기반의 바이너리 난독화 도구이다. 이 도구에서는 Instruction Substitution, Control Flow Flattening, Control Flow Splitting, Bogus Control Flow의 4가지 난독화 기법을 제시한다[3].

Obfuscator LLVM은 간단한 명령만으로 소스 코드를 난독화를 할 수 있는 장점을 가진다. 하지만 오픈 소스로 배포되어 있어 알고리즘 분석 및 역공학이 보다 쉽게 가능하여 각종 산업 및 다양한 IoT 제품에 적용하기 적합하지 않다. 따라서 소스 코드를 난독화 하는 독자적인 LLVM 최적화 Pass를 작성하여 임베디드 시스템의 보안적인 요소를 강화할 필요가 있다.

3. 자료흐름 평탄화 Pass 설계

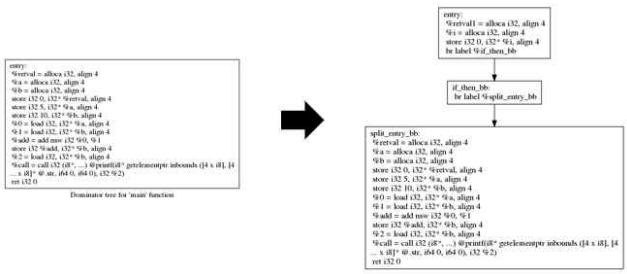
제어 흐름 평탄화는 소스 코드 난독화 기법 중 하나로 기능적으로는 동일하지만 정적분석을 어렵게 하기 위해 알고리즘의 복잡도를 증가시킨다. 알고리즘의 복잡도를 증가시키기 위해서 소스 코드를 switch-case문으로 변환하여 제어 흐름을 여러 단계로 나누거나 소스 코드의 각종 분기문과 반복문들을 switch-case문으로 변환시켜 검증해야 하는 경우의 수를 늘리는 방법이 있다.

그림 1은 if문과 switch-case문의 자료 흐름도를 보여준다. if문은 true/false의 2가지 경우의 수를 고려하지만, switch-case문은 case의 수만큼 검증해야 하므로 많은 시간을 정적 분석에 소모하게 한다.



(그림 1) if문과 switch-case문의 자료 흐름도

제어 흐름 평탄화를 하기 위해서는 IR의 기본 블록 단위에서 소스 코드의 switch-case문으로의 변환이 수행된다. 그림 2와 같이 본 논문은 entry라는 기본 블록을 대상으로 제어 흐름 평탄화가 수행된다. 기본 블록을 여러 기본 블록으로 분리시키는 것은 알고리즘의 복잡도를 증가시킨다. 여러 개로 분리된 기본 블록들은 무조건 분기로 연결하거나 조건문을 생성하여 참일 경우 해당 기본 블록으로 이동하게 한다. 이 과정에서 조건문의 false에 해당되는 기본 블록을 생성하여 더미 기본 블록으로 연결시킬 경우, 알고리즘의 복잡도는 향상된다.



(그림 2) 기본 블록을 여러 기본 블록으로 Split한 CFG

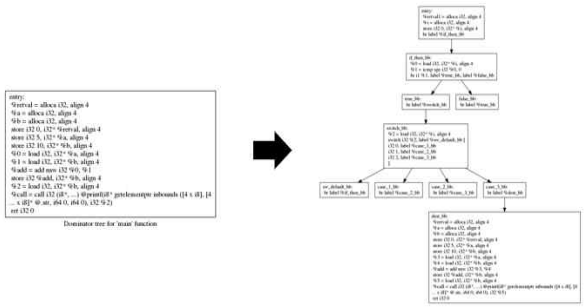
이후 split_entry_bb는 기본 블록은 N개의 case 기본 블록으로 나누고 분할된 기본 블록들은 알고리즘의 제어 흐름을 뒤섞는다. 그림 3은 switch-case문을 생성하는 알고리즘의 예시이며, 필요한 만큼 case의 수를 설정할 수 있다.

```
IRBuilder<> builder_switch(split_entry_bb);
BasicBlock *sw_default_bb = BasicBlock::Create(f.getContext(),
"sw_default_bb", &f, true_bb);
SwitchInst *switch_inst = builder_switch.CreateSwitch(var_switch,
sw_default_bb, bb.size());
switch_inst->addCase(builder_bodyBB.getInt32(0),case_1_BB);
switch_inst->addCase(builder_bodyBB.getInt32(1),case_2_BB);
switch_inst->addCase(builder_bodyBB.getInt32(2),case_3_BB);
```

(그림 3) switch - case 생성 알고리즘 예시

생성한 case 블록들은 split_entry_bb를 분할한 블록

들이지만, 의미 없는 더미 코드를 추가하여 제어 흐름을 복잡하게 할 수 있다[4]. 또한 switch의 조건을 문자열 난독화 하거나 변수를 비트 및 시프트 연산을 하여 알고리즘의 복잡도를 증가시킬 수 있다[5].

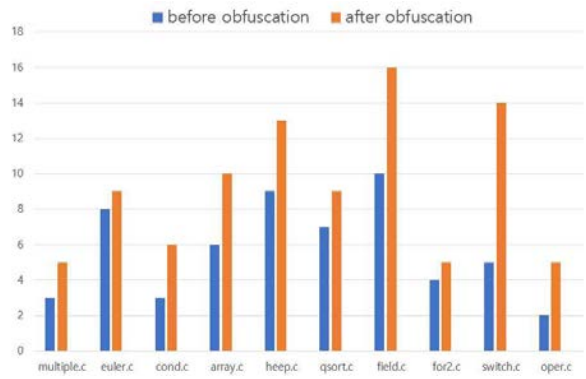


(그림 4) 자료 흐름 평탄화 Pass 적용 후 CFG

그림 4는 제어 흐름 평탄화 Pass를 적용한 결과를 나타내기 위하여 제어 흐름 그래프(CFG; Control Flow Graph)를 보여준다. Pass를 적용한 CFG는 기존의 CFG 보다 훨씬 제어 흐름이 복잡해 진 것을 확인할 수 있다.

4. 성능 평가

그림 5는 본 논문에서 구현한 자료 흐름 평탄화 Pass를 평가하기 위해 난독화 전과 후의 McCabe 순환 복잡도를 계산한 그래프이다[6].



(그림 5) 난독화 전/후의 McCabe 순환 복잡도 계산

자료 흐름 평탄화 Pass를 적용할 경우 원본 소스 보다 최대 2.8배로 평균 1.7배 복잡도가 증가하였다.

5. 결론 및 향후 연구 과제

임베디드 시스템은 일상 생활에 밀접하게 연관되어 있어 보안적인 측면이 중요시 여겨짐에 따라 역공학을 방지해야 하는 필요성이 있다. 본 논문은 이를 위해 임베디드 시스템에 사용 되는 소스 코드의 역공학을 방지하기 위한 제어 흐름 평탄화 난독화 알고리즘 설계 방법을 제안한다. 제어 흐름 평탄화를 구현하기 위한 환경으로는 LLVM과 Clang을 사용하였다. 이는 대체적으로 C언어로 프로그래밍 되어져 있는 임베디드 시스템에 적합하다. 그리고 재작성된 Pass의 적용은 IR 수준에서의 제어 흐름 평탄화를 가능하게 하였다. 그 결과 소스 코드의 특정 함수의 기본 블록은 switch-case문으로 변환되어 알고리즘의 복잡도가 증가되었음을 보여준다.

본 논문에서 구현한 제어 흐름 평탄화 Pass는 오픈 소스로 공개되어 있는 Obfuscator LLVM의 취약점을 보완하는 장점을 가진다. 이 Pass를 토대로 다양한 난독화 기법을 적용하여 더 복잡한 알고리즘을 구현하는 것을 향후 개발 목표로 한다.

참고문헌

[1] EETimes, “2019 Embedded Markets Study”, https://www.embedded.com/2019-embedded-markets-study-reflects-emerging-technologies-continued-c-c-dominance/.

[2] Ronny Tschüter, Johannes Ziegenbalg, Bert Wesarg, Matthias Weber, Christian Herold, Sebastian Döbel, Ronny Brendel, “An LLVM Instrumentation Plug-in for Score-P”, LLVM-HPC'17: *Proceedings of the Fourth Workshop on the LLVM Compiler Infrastructure in HPC*, no. 2, pp. 1-8, Nov. 2017.

[3] Pascal Junod, Julien Rinaldini, Johan Wehrli and Julie Michielin, “Obfuscator-LLVM — Software Protection for the Masses”, 2015 IEEE/ACM 1st International Workshop on Software Protection (SPRO), Oct. 2015.

[4] 이규호, 유재관, 김인성, 김태규, “무기체계 안티탬퍼링을 위한 소프트웨어 소스코드 난독화 도구 구현”, 정보과학회논문지, 제 46권, 제 5호, pp. 448-456, Mar. 2019.

[5] Chih-Fan Chen, Adrian Tang, “CONFUSE: LLVM-based Code Obfuscation”, 2013.

[6] Tímea László, Ákos Kiss, “Obfuscating C++ Programs via Control Flow Flattening”, 10th Symposium on Programming Languages and Software Tools (SPLST), June, 2009.