Journal of The Korea Institute of Information Security & Cryptology ISSN 1598-3986(Print)
VOL.28, NO.3, Jun. 2018 ISSN 2288-2715(Online)
https://doi.org/10.13089/JKIISC.2018.28.3.605

소스 코드 난독화 기법에 의한 Clang 정적 분석 도구의 성능 영향 분석*

진 홍 주,[†] 박 문 찬, 이 동 훈[‡]
고려대학교 정보보호대학원

Analysis of Detection Ability Impact of Clang Static Analysis Tool by Source Code Obfuscation Technique*

Hongjoo Jin,[†] Moon Chan Park, Dong Hoon Lee[‡]
Graduate School for Information Security, Korea University

요 약

사물 인터넷 시장 규모의 급격한 성장에 따라 임베디드 시스템에서 가장 많이 쓰이는 언어인 C/C++ 언어의 사용도 증가하고 있다. C/C++ 언어의 코드 품질을 향상시키고 개발 비용을 절감시키기 위해 소프트웨어 개발 전반부에서 수행 가능한 소프트웨어 검증기법인 정적 분석을 사용하는 것이 좋다. 많은 프로그램들이 정적 분석을 사용하여 소프트웨어의 안전성을 검증하고 있으며 많은 정적 분석 도구들이 사용 및 연구되고 있다.

본 논문에서는 Clang 정적 분석도구를 사용하여 검증된 시험 코드에 대한 보안 약점 검출 성능을 확인한다. 또한 소스 코드 난독화 기법인 구획 난독화, 데이터 난독화, 제어 흐름 난독화 기법이 각각 적용된 시험 코드에 대한 정적 분석 결과와 원본 시험 코드에 대한 정적 분석 결과를 비교하여 소스 코드 난독화 기법에 따른 Clang 정적 분석 도구의 성능 영향을 분석한다.

ABSTRACT

Due to the rapid growth of the Internet of Things market, the use of the C/C++ language, which is the most widely used language in embedded systems, is also increasing. To improve the quality of code in the C/C++ language and reduce development costs, it is better to use static analysis, a software verification technique that can be performed in the first half of the software development life cycle. Many programs use static analysis to verify software safety and many static analysis tools are being used and studied.

In this paper, we use Clang static analysis tool to check security weakness detection performance of verified test code. In addition, we compared the static analysis results of the test codes applied with the source obfuscation techniques, layout obfuscation, data obfuscation, and control flow obfuscation techniques, and the static analysis results of the original test codes, Analyze the detection ability impact of the Clang static analysis tool.

Keywords: Internet of Things, Software Weakness, Static Analysis, Source Code Obfuscation

Received(03. 16. 2018), Modified(05. 23. 2018), Accepted(05. 23. 2018)
* 본 논문은 2017년도 동계학술대회에 발표한 우수논문을 개선 및 확장한 것임.
* 이 성과는 2018년도 과학기술정보통신부의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No. NRF-2017R1A2B3009643).
† 주저자, realredwine@korea.ac.kr
‡ 교신저자, donghlee@korea.ac.kr(Corresponding author)

Ⅰ. 서 론

각종 산업과 일상생활에 널리 활용되고 있는 사물인터넷 시장 규모는 매년 크게 성장하고 있으며, 사물인터넷 기기의 수 또한 급격하게 증가하고 있다〔1〕. IEEE Spectrum에서 발표한 Interactive: The Top Programming Languages 2017 (Fig. 1.)에 의하면 임베디드 시스템에서 C/C++ 언어가 가장 많이 사용되고 있으며〔2〕, 앞으로도 임베디드 소프트웨어 개발자를 위한 프로그래밍 언어로써 가장 선호하는 경향이 지속될 것이다. 다른 프로그래밍 언어와 차별화되는 C/C++ 언어의 가장 큰 장점은 바로 “low-level” 특성을 가진 고수준 언어라는 것이다〔13〕. C/C++ 언어는 임베디드 소프트웨어 프로그래머에게 고급 언어의 이점을 희생하지 않으면서 직접적인 하드웨어 제어를 제공한다. 하지만 “low-level” 특성으로 인해 C/C++ 언어는 버퍼 오버플로, 취약한 라이브러리의 사용, 안전하지 않은 포인터의 사용 등의 취약점을 가진다.

소프트웨어 취약점은 시스템 설계, 구현 또는 운영 관리에서 발생하는 결함 또는 약점으로〔16〕, 공격자와 같은 위협 요소가 시스템 내부에서 승인되지 않은 작업을 수행할 때 악용될 수 있는 약점이다. 네트워크로 연결된 컴퓨터 시스템의 소프트웨어는 외부에서 접근할 수 있으며, 악의적인 공격자가 소프트웨어 취약점을 이용하여 원격 공격을 수행할 수 있다〔17〕. 따라서 소프트웨어 개발 단계에서 결함을 찾아내는 방법과 이러한 취약점을 보완하는 것이 중요하다.

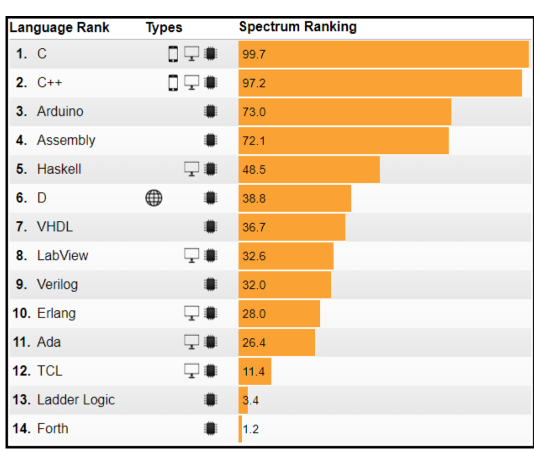

Fig. 1. Interactive: The Top Programming Languages 2017(IEEE Spectrum).

프로그램을 실행시키지 않고 소프트웨어 취약점을 발견하고 제거하기 위해 다양한 정적 테스팅(static testing) 기법이 사용된다. 소프트웨어 검사(software inspection), 정적 분석(static analysis) 등의 정적 테스팅 기법을 사용하여 소프트웨어의 품질을 향상시킬 수 있다〔14〕. 수동적(manual) 정적 테스팅 기법인 소프트웨어 검사는 소프트웨어 문서 및 코드의 결함을 탐지하는 효과적인 방법이지만 일반적으로 여러 명의 프로그래머가 필요하므로 비용이 많이 든다〔15〕. 정적 분석은 주로 정적 분석 도구(static analysis tool)를 사용하여 자동화된 정적 테스팅을 수행하기 때문에 비용 측면에서 효율적이다.

C/C++ 언어에서 빈번하게 발생하는 언어 규칙 위반과 결함을 탐지하고, 취약점을 보완하기 위해 정적 분석 도구를 사용하는 것이 효과적이다. 정적 분석 도구는 많은 수의 프로그램을 비교적 짧은 시간에 분석할 수 있으며 소프트웨어 개발 생명 주기의 전반부에 사용가능하기 때문에 개발 비용을 절감할 수 있는 장점이 있다. 반면에 소스 코드의 크기가 커지거나 경로(path)가 복잡해지면 정적 분석 도구의 성능에 따라 보안 약점을 검출 못할 수 있다. 그렇기 때문에 소스 코드에 난독화 (obfuscation) 기법이 적용되어 데이터 흐름 또는 제어 흐름이 복잡해지면 일반 소스 코드보다 정적 분석이 어려워진다.

본 논문에서는 검증된 시험 코드에 대한 Clang 정적 분석 도구의 검출 성능을 확인한다. 또한 소스 코드 난독화 기법이 적용된 시험 코드에 대한 Clang 정적 분석 도구의 검출 성능이 달라짐을 확인하고, 이를 분석하여 Clang 정적 분석 도구의 성능 영향을 파악한다. 본 논문의 2장에서 관련 연구를 소개하고, 3장에서는 Clang 정적 분석 도구와 정적 분석 대상 시험 코드인 줄리엣 시험 코드에 대하여 소개한다. 4장에서는 소스 코드 난독화 기법을 설명하고, 줄리엣 시험 코드에 소스 코드 난독화 기법을 적용하는 과정을 보여준다. 5장에서는 원본 시험 코드와 난독화 기법이 적용된 시험 코드 집합에 대하여 Clang 정적 분석 도구로 정적 분석을 수행하여 보안 약점을 검출한 결과를 보여주고, 이를 분석한다. 6장에서는 5장에서 분석한 결과를 토대로 결론을 맺는다.

II. 관련 연구

정적 분석은 효율적인 소스 코드 분석이 가능하지만 분석 결과가 정적 분석 도구의 성능에 좌우되는 단점이 있다. 그렇기 때문에 정적 분석 도구의 정확성과 기능성을 분석하는 연구가 계속해서 진행되고 있다.

Nathaniel Ayewah 등[20]은 자바(java) 소스 코드에서 결함을 탐지하는 정적 분석 도구인 FindBugs[21]의 성능과 사용자 편의성을 분석하였다. 이 논문에서는 실제 사용되는 자바 코드에 대하여 FindBugs 정적 분석 도구를 사용하여 검출 성능을 살펴보았고, 사용자 설문을 통해 개선점을 제시하였다.

Kostyantyn Vorobyov 등[22]은 C/C++ 프로그램에 대한 바운드 모델 체커(bounded model checker)인 CBMC[23]와 C/C++ 언어를 대상으로 하는 정적 분석 도구인 Parfait[24]을 비교하는 연구를 진행하였다. 이 연구에서는 서로 다른 정적 테스팅 기법을 사용하는 정적 분석 도구와 바운드 모델 체커의 성능을 비교하기 위해 BegBunch[25] 벤치마킹 프레임 워크를 사용하였다. 또한 BegBunch 벤치마크에 대한 CBMC와 Parfait의 버그 검출 결과를 분석하여 두 분석 도구의 성능을 비교하였다.

소프트웨어 품질 개선과 보안성 향상을 위해 NIST(National Institute of Standards and Technology)는 2004년부터 SAMATE (Software Assurance Metrics And Tool Evaluation)[6] 프로젝트를 시작했다. SAMATE 프로젝트에서 Vadim Okun 등[25][26][27][28][29]은 2008년부터 상용 정적 분석 도구에 대한 연구 및 개선을 위해 SATE(Static Analysis Tool Exposition)를 실시하였다. 초기의 SATE는 오픈 소스 프로그램의 알려진 결함에 대하여 정적 분석 도구들의 검출 결과를 다방면으로 분석하였다. 이후 특정 보안 약점에 대한 정적 분석 도구의 검출 성능을 객관적으로 측정하기 위해 개발된 시험 코드 집합을 사용하였다.

정적 분석 도구에 대한 연구는 주로 상용 정적 분석 도구의 성능과 정확도를 측정하는 방향으로 진행되었다. 또한 성능과 정확도에 대한 객관적인 수치를 얻기 위해 오픈 소스 프로그램의 알려진 결함을 이용하거나, 의도적으로 결함을 내포한 시험 코드를 사용하였다. 본 연구에서는 오픈 소스 정적 분석 도구의 보안 약점 검출 성능을 확인하기 위해 시험 코드를 사용한다. 성능 확인을 통해 정적 분석 도구가 어떤 종류의 결함을 얼마나 잘 검출할 수 있는지 파악한다. 또한 소스 코드 난독화 기법이 정적 분석 도구의 검출 성능에 끼치는 영향을 파악하기 위해 상용 소스 코드 난독화 도구를 사용하여 시험 코드에 난독화 기법을 적용한다. 난독화 기법이 적용된 시험 코드에 대한 정적 분석 도구의 검출 성능 변화를 파악 및 분석한다. 이를 통해 오픈 소스 정적 분석 도구의 지속적인 개발 및 보완에 필요한 정보를 제공할 수 있다.

III. 배경지식

3.1 Clang 정적 분석 도구

LLVM[3] 컴파일러의 Clang[4]은 C/C++ 언어를 대상으로 특정 언어나 아키텍처에 종속적이지 않은 중간 언어(intermediate representation)를 생성하는 프론트엔드이다. 또한 Clang은 C/C++ 프로그램에 대하여 정적 분석을 수행할 수 있는 모듈인 Clang 정적 분석 도구[18]를 가지고 있으며, 이를 이용해 소스 코드의 보안 약점을 검출할 수 있다.

Clang의 정적 분석 옵션에는 체커(checker)를 선택하는 부분이 있으며, 검출하고자 하는 보안 약점에 알맞은 체커를 선택하여 정적 분석을 수행할 수 있다. 체커는 특정 보안 약점에 대해 정적 분석을 수행하는 기능별로 분류되어 있으며 8개의 체커 집합에 속한 95개 체커가 존재한다[19]. 체커의 개발 방법과 체커 관련 소스 코드는 공개되어있으며 새로운 체커의 개발과 기존 체커의 보완 및 수정은 계속해서 진행되고 있다. Fig. 2.는 alpha.unix.cstring.OutOfBounds 체커를 이용해 버퍼 오버플로 보안 약점을 찾아 경고를 표시하는 예이다. clang 명령어에 정적 분석 옵션을 추가하고, 원하는 체커를 선택

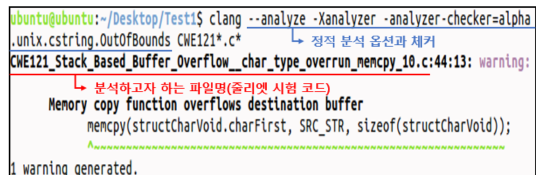

Fig. 2. Using Clang Static Analysis Tool to Detect Security Weakness.

하여 정적 분석 대상 소스 코드에 대하여 정적 분석을 수행하였다. clang은 소스 코드에서 보안 약점이 발견된 위치를 표시해주고, 해당 보안 약점에 대한 경고 메시지와 보안 약점이 검출된 코드 부분을 표시해준다.

3.2 줄리엣 시험 코드

본 논문에서 사용한 정적 분석 대상 코드는 줄리엣 시험 코드(juliet test suite v1.3 for C/C++)[5]로 NIST SAMATE 프로젝트의 SATE에서 정적 분석 도구들의 성능을 평가하기 위해 사용한 시험 코드이다. 줄리엣 시험 코드는 CWE(Common Weakness Enumeration)[7]를 기준으로 118개 보안 약점 집합으로 분류되어있다. 각 집합은 보안 약점을 유발하는 결함 유형(flaw type)을 1개 이상 가지고 있으며 줄리엣 시험 코드는 1,617개의 결함 유형을 포함한다.

줄리엣 시험 코드의 생성 규칙은 다음과 같다. 결함 유형별로 생성한 시험 코드를 베이스 라인(baseline)으로 하여 01번으로 번호를 부여한다. 베이스 라인 시험 코드에서 제어 흐름을 변형하여 02번부터 22번까지의 시험 코드를 파생시키고, 데이터 흐름을 변형하여 31번부터 84번까지의 시험 코드를 생성한다. 제어 흐름 변형은 조건문이나 반복문을 추가하여 경로를 복잡하게 만드는 방식을 사용하고, 데이터 흐름 변형은 지역 변수, 전역 변수, 함수의 매개변수 및 인수 등 사용하는 데이터에 접근하기 위한 경로가 복잡해지는 방법을 사용한다. 베이스 라인인 01번 시험 코드에서 숫자가 증가할수록 소스 코드의 복잡도가 증가하는 양상을 보인다.

IV. 난독화 적용

난독화 기법에 따른 Clang 정적 분석 도구의 성능 영향을 분석하기 위해 줄리엣 시험 코드에 난독화 기법을 적용한다. 난독화 기법은 적용 대상에 따라서 바이너리 난독화와 소스 코드 난독화로 나눌 수 있다. Clang 정적 분석 도구는 소스 코드에 대하여 정적 분석을 수행하므로 줄리엣 시험 코드에 소스 코드 난독화 기법을 적용한다. 소스 코드 난독화 기법은 구획 난독화(layout obfuscation), 데이터 난독화 (data obfuscation), 제어 흐름 난독화

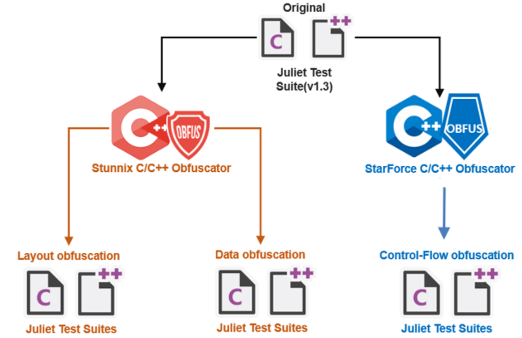

Fig. 3. Using Stunnix C/C++ Obfuscator and StarForce C/C++ Obfuscator to generate a set of test codes with three source code obfuscation techniques(Layout obfuscation, Data obfuscation, Control-Flow obfuscation).

(control flow obfuscation), 예방 난독화(preventive obfuscation)로 구분할 수 있다[8].

구획 난독화 기법은 함수나 변수 등 식별자 이름을 변환하는 방식을 사용한다[11]. 프로그램의 성능에 적은 영향을 주는 장점이 있으며 간단하게 구현할 수 있다. 데이터 난독화 기법은 데이터의 형태나 구조를 변환하는 변수 분할, 문자열 암호화, 인코딩 형식 변환 등의 방법을 사용한다. 데이터가 사용되는 목적을 감출 수 있는 장점이 있으며 데이터 분석에 소요되는 시간을 증가시키기도 한다[30]. 제어 흐름 난독화 기법은 프로그램의 제어 흐름을 복잡하게 만들어 실제 프로그램의 제어 흐름 구조를 감추는 방식을 사용한다[31]. 제어 흐름 실제 프로그램 실행에 영향을 주지 않는 더미 코드를 삽입하는 계산 난독화(computation obfuscation) 기법을 주로 사용한다[8]. 불투명 술어(opaque predicate) 기법은 입력 값에 상관없이 조건문의 결과가 항상 상수 true 또는 상수 false로 평가되는 술어를 사용한다[32]. 일반적인 제어 흐름 난독화 기법은 계산 난독화 기법과 불투명 술어 기법을 함께 사용하여 제어 흐름 구조를 복잡하게 만든다. 예방 난독화의 경우 이미 알려진 역난독화 방법에 대한 역난독화 방지 기법을 적용하는 방법을 사용하는 난독화 기법이다[9]. 그렇기 때문에 정적 분석 대상 시험 코드의 난독화에 사용하지 않았다.

본 논문에서는 원본 줄리엣 시험 코드 집합과 구획 난독화, 데이터 난독화, 제어 흐름 난독화 기법이 적용된 3개의 시험 코드 집합, 총 4개의 집합을 정적 분석 대상 소스 코드로 사용한다. 구획 난독화와

데이터 난독화가 적용된 2개의 시험 코드 집합을 생성하기 위해 Stunnix 난독화 도구를 사용하며, 제어 흐름 난독화가 적용된 시험 코드 집합은 StarForce 난독화 도구을 사용하여 얻는다. Fig. 3.는 Stunnix와 StarForce 난독화 도구를 사용하여 원본 줄리엣 시험 코드에 소스 코드 난독화 기법을 적용하는 과정을 나타낸다.

4.1 Stunnix 난독화 도구

줄리엣 시험 코드에 구획 난독화와 데이터 난독화 기법을 적용하기 위해 Stunnix 난독화 도구(Stunnix C/C++ Obfuscator v4.7)를 사용한다[10]. 다른 난독화 도구들은 난독화 정도를 사용자가 설정하여 난독화를 수행하는 경우가 많았고, 그렇기 때문에 구획 난독화와 데이터 난독화 등 여러 난독화 기법이 함께 적용된다. 각각의 난독화 기법에 의한 Clang 정적 분석 도구의 성능 영향을 확인하기 위해 Stunnix 난독화 도구를 선택하였다. Stunnix 난독화 도구의 경우 식별자(identifier), 문자열과 상수 값, 공백 문자와 주석 등에 대한 세부적인 난독화 옵션을 선택할 수 있기 때문에 구획 난독화와 데이터 난독화를 분리하여 진행할 수 있다. 하지만 제어 흐름 난독화 기능이 없기 때문에 이를 위해 다른 도구를 사용한다.

Stunnix 난독화 도구를 이용하여 원본 줄리엣 시험 코드 집합에 식별자 치환, 주석 제거, 공백 문자 섞기 등의 구획 난독화 기법을 적용한다. 또한 원본 줄리엣 시험 코드 집합에 데이터 분할, 연산자 추가 등의 데이터 난독화 기법을 적용한다.

4.2 StarForce 난독화 도구

줄리엣 시험 코드에 제어 흐름 난독화를 적용하기 위해 Starforce[12] 난독화 도구(StarForce C++ Obfuscator v5.1.3)를 사용한다. Starforce 난독화 도구는 소스 코드에 제어 흐름 난독화를 적용하기 위해 프로그램의 원래 실행 흐름에 영향을 주지 않는 더미 코드를 삽입한다. 더미 코드는 기존 프로그램의 제어 흐름 사이에 레이블과 goto문을 삽입하여 구현한다. 또한 그렇기 때문에 난독화 이후 소스 코드 크기의 변화가 적은 Stunnix 난독화 도구와 달리 소스 코드 크기가 15~100배 정도 증가한다.

Starforce 난독화 도구를 이용하여 원본 줄리엣 시험 코드 집합에 더미 코드 역할을 하는 반복문과 조건문을 이용한 제어 흐름 난독화 기법을 적용한다. 하지만 제어 흐름 난독화 과정에서 기본적으로 식별자 치환, 변수 분할을 함께 적용하기 때문에 제어 흐름 난독화 기법만이 적용된 시험 코드 집합을 얻을 수 없었다. Starforce 난독화 도구는 사용자가 난독화 레벨(level)을 선택하여 난독화를 진행할 수 있으며, 레벨 0부터 레벨 4까지 난독화 레벨을 조절할 수 있다. 가장 난독화 정도가 심한 레벨 4의 경우 소스 코드의 크기가 원본에 비해 100배 이상 증가하였고, 실험에 적당한 레벨 3을 선택하여 제어 흐름 난독화를 적용하였다.

V. 시험 코드에 대한 정적 분석

정적 분석은 리눅스 운영체제에서 수행하였으며 LLVM/Clang 5.0.1 버전의 정적 분석 도구를 사용하였다. Clang의 95개 체커 중 31개의 체커가 줄리엣 시험 코드 집합의 보안 약점을 검출할 수 있었고, 118개의 CWE로 분류된 시험 코드 집합 중 39개의 집합, 총 35,008개 시험 코드에 대하여 정적 분석을 수행하였다. Fig. 1. 같이 특정 시험 코드에서 보안 약점을 Clang의 체커가 검출하면 파일 이름과 위치, 해당 위치의 코드를 출력하여 표시해준다. 또한 체커마다 보안 약점을 검출하였을 때 출력하는 경고 메시지를 가지고 있기 때문에 해당 코드 부분에서 어떤 보안 약점이 검출되었는지 알 수 있다.

각각 35,008개의 C/C++ 시험 코드로 이루어진 4개의 시험 코드 집합에 대한 Clang 정적 분석 도구의 검출 결과는 Table 1~3에서 확인할 수 있다. Table은 기본적으로 보안 약점 종류를 나타내는 CWE와 시험 코드의 수, Clang이 보안 약점을 검출한 시험 코드 수, 검출 비율로 구성된다. 구체적인 CWE 명칭은 Table 1.에 명시한 후 Table 2, Table 3에는 CWE ID로 간략하게 표기하였다. Table 2와 Table 3에는 감소율 항목이 추가되었고, 이는 원본 시험 코드에 대한 검출 수 대비 난독화가 적용된 시험 코드에 대한 검출 수 감소 비율을 나타낸다. Table 1~3에서 검출 비율과 감소율 항목은 소수점 셋째 자리에서 반올림하여 표기하였다.

Table 1. Detection results of Clang static analysis tool for original Juliet test code.

CWE 이름	시험 코드 수	검출 수	검출 비율
CWE-121 Stack Based Buffer Overflow	4,944	761	15.39 %
CWE-122 Heap Based Buffer Overflow	5,892	1,011	17.16 %
CWE-124 Buffer Underwrite	2,048	554	27.05 %
CWE-126 Buffer Overread	1,452	27	1.86 %
CWE-127 Buffer Underread	2,048	564	27.54 %
CWE-188 Reliance on Data Memory Layout	36	18	50.00 %
CWE-194 Unexpected Sign Extension	1,152	284	24.65 %
CWE-195 Signed to Unsigned Conersion Error	1,152	366	31.77 %
CWE-196 Unsigned to Signed Conversion Error	18	0	0 %
CWE-197 Numeric Truncation Error	864	0	0 %
CWE-242 Use of Inherently Dangerous Function	18	0	0 %
CWE-369 Divide by Zero	864	54	6.25 %
CWE-377 Insecure Temporary File	144	18	12.5 %
CWE-400 Resource Exhaustion	720	240	33.33 %
CWE-401 Memory Leak	1,658	848	51.15 %
CWE-404 Improper Resource Shutdown	384	22	5.73 %
CWE-415 Double Free	962	462	48.02 %
CWE-416 Use After Free	459	210	45.75 %
CWE-457 Use of Uninitialized Variable	948	415	43.78 %
CWE-467 Use of sizeof on Pointer Type	54	54	100 %
CWE-468 Incorrect Pointer Scaling	37	0	0 %
CWE-469 Use of Pointer Subtraction to Determine Size	36	0	0 %
CWE-476 NULL Pointer Dereference	348	204	58.62 %
CWE-561 Dead Code	2	1	50.00 %
CWE-562 Return of Stack Variable Address	3	3	100 %
CWE-563 Unused Variable	512	230	44.92 %
CWE-570 Expression is Always False	16	0	0 %
CWE-571 Expression Always True	16	1	6.25 %
CWE-587 Assignment of a Fixed Address to a Pointer	18	0	0 %
CWE-588 Attempt to Access Child of a Non-structure Pointer	80	0	0 %
CWE-590 Free of Memory not on the Heap	2,680	96	3.58 %
CWE-665 Improper Initialization	193	0	0 %
CWE-675 Duplicate Operations on Resource	192	27	14.06 %
CWE-680 Integer Overflow to Buffer Overflow	576	144	25.00 %
CWE-681 Incorrect Conversion between Numeric Types	54	0	0 %
CWE-761 Free of Pointer not at Start of Buffer	576	210	36.46 %
CWE-762 Mismatched Memory Management Routines	3,564	2,010	56.40 %
CWE-773 Missing Reference to Active File Descriptor or Handle	144	44	30.56 %
CWE-775 Missing Release of File Descriptor or Handle after Effective Lifetime	144	44	30.56 %
합 계	35,008	8,922	25.49 %

5.1 원본 줄리엣 시험 코드 정적 분석

원본 줄리엣 시험 코드에 대하여 정적 분석을 수행한 결과는 Table 1.에서 확인할 수 있으며 35,008개 코드 중 25.49%인 8,922개 시험 코드에 대하여 보안 약점을 검출하였다. 정적 분석 결과는 본 논문의 3.2절에서 설명한 줄리엣 시험 코드의 구성 방식의 영향을 받았다. Clang 정적 분석 도구는 상대적으로 복잡도가 낮고, 제어 흐름을 변형한 하위 번호의 시험 코드(01번~22번)에 대해서는 우수한 검출 능력을 보였다. 하지만 하위 번호의 시험 코드보다 복잡도가 높고, 데이터 흐름을 변형한 상위 번호의 시험 코드(31번~84번)에 대해서는 보안 약점을 잘 검출하지 못했다. 이는 시험 코드의 복잡도가

Table 2. Detection results of Clang static analysis tool for Juliet test code with layout obfuscation and data obfuscation.

CWE ID	구획 난독화	검출 수	검출 비율	감소율	데이터 난독화	검출 수	검출 비율	감소율
CWE-121	4,944	761	15.39 %	-	4,944	761	15.39 %	-
CWE-122	5,892	1,011	17.16 %	-	5,892	1,011	17.16 %	-
CWE-124	2,048	554	27.05 %	-	2,048	554	27.05 %	-
CWE-126	1,452	27	1.86 %	-	1,452	27	1.86 %	-
CWE-127	2,048	564	27.54 %	-	2,048	564	27.54 %	-
CWE-188	36	18	50.00 %	-	36	18	50.00 %	-
CWE-194	1,152	284	24.65 %	-	1,152	284	24.65 %	-
CWE-195	1,152	366	31.77 %	-	1,152	366	31.77 %	-
CWE-196	18	0	0 %	-	18	0	0 %	-
CWE-197	864	0	0 %	-	864	0	0 %	-
CWE-242	18	0	0 %	-	18	0	0 %	-
CWE-369	864	54	6.25 %	-	864	54	6.25 %	-
CWE-377	144	18	12.5 %	-	144	18	12.5 %	-
CWE-400	720	240	33.33 %	-	720	240	33.33 %	-
CWE-401	1,658	848	51.15 %	-	1,658	848	51.15 %	-
CWE-404	384	22	5.73 %	-	384	22	5.73 %	-
CWE-415	962	462	48.02 %	-	962	462	48.02 %	-
CWE-416	459	210	45.75 %	-	459	210	45.75 %	-
CWE-457	948	415	43.78 %	-	948	415	43.78 %	-
CWE-467	54	54	100 %	-	54	54	100 %	-
CWE-468	37	0	0 %	-	37	0	0 %	-
CWE-469	36	0	0 %	-	36	0	0 %	-
CWE-476	348	204	58.62 %	-	348	204	58.62 %	-
CWE-561	2	1	50.00 %	-	2	1	50.00 %	-
CWE-562	3	3	100 %	-	3	3	100 %	-
CWE-563	512	230	44.92 %	-	512	230	44.92 %	-
CWE-570	16	0	0 %	-	16	0	0 %	-
CWE-571	16	1	6.25 %	-	16	0	0 %	100 %
CWE-587	18	0	0 %	-	18	0	0 %	-
CWE-588	80	0	0 %	-	80	0	0 %	-
CWE-590	2,680	96	3.58 %	-	2,680	96	3.58 %	-
CWE-665	193	0	0 %	-	193	0	0 %	-
CWE-675	192	27	14.06 %	-	192	27	14.06 %	-
CWE-680	576	17	2.95 %	88.19 %	576	144	25.00 %	-
CWE-681	54	0	0 %	-	54	0	0 %	-
CWE-761	576	210	36.46 %	-	576	210	36.46 %	-
CWE-762	3,564	2,010	56.40 %	-	3,564	2,010	56.40 %	-
CWE-773	144	44	30.56 %	-	144	44	30.56 %	-
CWE-775	144	44	30.56 %	-	144	44	30.56 %	-
합 계	35,008	8,795	25.12 %	1.42 %	35,008	8,921	25.49 %	0.01 %

높을수록, 제어 흐름보다는 데이터 흐름이 변형될 경우 Clang 정적 분석 도구가 보안 약점을 검출하지 못하는 것을 보여준다.

Clang 정적 분석 도구는 버퍼 오버플로 관련 보안 약점보다 use-after-free 취약점과 같은 메모리 누수(memory leak) 관련 보안 약점에 대해서 준수한 검출 성능을 나타냈다. 메모리 누수 관련 보안 약점은 주로 동적으로 자원을 할당하는 API의 사용과 사용 후 자원 할당 해제에 대한 약점이다. 그렇기 때문에 메모리 할당 관련 함수와 메모리 할당 해제 관련 함수의 대응을 확인하는 간단한 방법으로 메모리 누수가 발생하는지 확인할 수 있고, 보안 약점 유무를 판단할 수 있다.

5.2 구획 난독화가 적용된 시험 코드 정적 분석

구획 난독화 기법이 적용된 시험 코드 집합에 대한 정적 분석 결과는 Table 2.의 2~5열에서 확인할 수 있다. 구획 난독화가 적용된 35,008개 시험 코드 중 8,795개의 시험 코드에서 보안 약점을 검출하였고 원본 시험 코드에 비해 1.42% 정도의 감소율을 나타내었다. 39개 보안 약점 중 Integer Overflow to Buffer Overflow에 대한 정적 분석 결과만 구획 난독화의 영향을 받았다. 이 보안 약점에 해당하는 시험 코드의 경우 힙 메모리 사용을 위한 동적 할당을 수행하는 API에 전달하는 인자에 대한 버퍼 오버플로를 코드의 결함으로 사용하였다. 여러 라이브러리의 함수를 사용하여 데이터 흐름이 복잡해지고, 구획 난독화의 적용이 추가되어 검출 성능이 떨어진 것으로 보인다.

5.3 데이터 난독화가 적용된 시험 코드 정적 분석

데이터 난독화 기법에 의해 난독화된 시험 코드에 대한 정적 분석 결과는 Table 2.의 6~9열에서 확인할 수 있다. 데이터 난독화가 적용된 35,008개 시험 코드 중 8,921개 시험 코드에서 보안 약점을 검출하였고, 원본 시험 코드에 비해 0.01%의 감소율을 나타내었다. 39개 보안 약점 중 Expression Always True에 대한 정적 분석 결과만 데이터 난독화의 영향을 받았다. Fig. 2.는 해당 보안 약점의 원본 줄리엣 시험 코드와 데이터 난독화가 적용된 시험 코드 부분이다. 조건문의 조건 값 2가 난독화되었고 경로 변화는 없다. 피연산자와 연산자의 복잡도가 증가함에 따라 Clang 정적 분석 도구가 연산을 완벽하게 수행하지 못하고 정적 분석을 진행했기 때문에 해당 보안 약점을 검출하지 못했다.

5.4 제어 흐름 난독화가 적용된 시험 코드 정적 분석

제어 흐름 난독화 기법에 의해 난독화된 시험 코드에 대한 정적 분석 결과는 Table 3.의 2~5열에서 확인할 수 있다. 제어 흐름 난독화가 35,008개 시험 코드 중 3,034개의 시험 코드에서 보안 약점을 검출 하였고 원본 시험 코드에 비해 65.99% 정도의 감소율을 나타내었다. 제어 흐름 난독화 기법이 적용된 시험 코드는 원본 시험 코드에 비해 경로가 복잡해지고, 경로의 길이가 크게 길어졌다. 그렇기 때문에 대부분의 보안 약점에 대한 검출 성능에 영향이 있었으며, Clang 정적 분석 도구가 준수한 검출 성능을 나타내었던 메모리 누수 관련 보안 약점에 대한 검출 비율이 크게 감소하였다. 이는 소스 코드에 대한 정적 분석을 수행할 때 모든 경로에 대하여 분석

Table 3. Detection results of Clang static analysis tool for Juliet test code with control-flow obfuscation.

CWE ID	제어 흐름 난독화	검출 수	검출 비율	감소율
CWE-121	4,944	583	11.79 %	23.39 %
CWE-122	5,892	754	12.80 %	25.42 %
CWE-124	2,048	381	18.60 %	31.23 %
CWE-126	1,452	13	0.90 %	51.85 %
CWE-127	2,048	400	19.53 %	29.08 %
CWE-188	36	0	0 %	100 %
CWE-194	1,152	144	12.5 %	49.30 %
CWE-195	1,152	210	18.23 %	42.62 %
CWE-196	18	0	0 %	-
CWE-197	864	0	0 %	-
CWE-242	18	0	0 %	-
CWE-369	864	0	0 %	100 %
CWE-377	144	0	0 %	100 %
CWE-400	720	0	0 %	100 %
CWE-401	1,658	168	10.13 %	80.19 %
CWE-404	384	8	2.08 %	63.63 %
CWE-415	962	20	2.08 %	95.67 %
CWE-416	459	23	5.01 %	89.05 %
CWE-457	948	169	17.83 %	59.28 %
CWE-467	54	0	0 %	100 %
CWE-468	37	0	0 %	-
CWE-469	36	0	0 %	-
CWE-476	348	0	0 %	100 %
CWE-561	2	1	50.00 %	-
CWE-562	3	0	0 %	100 %
CWE-563	512	18	3.52 %	92.17 %
CWE-570	16	0	0 %	-
CWE-571	16	0	0 %	100 %
CWE-587	18	0	0 %	-
CWE-588	80	0	0 %	-
CWE-590	2,680	0	0 %	100 %
CWE-665	193	0	0 %	-
CWE-675	192	0	0 %	100 %
CWE-680	576	102	17.71 %	29.17 %
CWE-681	54	0	0 %	-
CWE-761	576	0	0 %	100 %
CWE-762	3,564	0	0 %	100 %
CWE-773	144	20	13.89 %	54.55 %
CWE-775	144	20	13.89 %	54.55 %
합 계	35,008	3,034	8.67 %	65.99 %

```
void CWE571_Expression_Always_True__two_equals_two_01_bad()
{
    /* FLAW: This expression is always true */
    if (2 == 2)
    {
        printLine("Always prints");
    }
}
```

```
void CWE571_Expression_Always_True__two_equals_two_01_bad()
{if((0xe43+4357-0x1f46)==(0x22+881-0x391)){printLine(
"₩x41₩x6c₩x77₩x61₩x79₩x73₩x20₩x70₩x72₩x69₩x6e₩x74₩x73");}}
```

Fig. 4. Original Juliet test code and test code with data obfuscation(Expression Always True weakness).

을 수행하지 않기 때문인 것으로 보인다.

VI. 결 론

본 논문에서는 검증된 시험 코드를 이용하여 Clang 정적 분석 도구의 보안 약점 검출 성능을 확인하였다. 원본 줄리엣 시험 코드에 대한 보안 약점 검출 결과를 통해 Clang 정적 분석 도구가 어떤 종류의 보안 약점을 잘 검출하지 못하는지 파악할 수 있다. 이를 통해 Clang 정적 분석 도구의 성능 향상을 위해 어떤 체커를 우선적으로 개발 및 보완해야 하는지 참고할 수 있다. 또한 소스 코드 난독화 기법인 구획 난독화, 데이터 난독화, 제어 흐름 난독화 기법이 적용된 각각의 시험 코드 집합에 대한 검출 결과를 확인하였다. 정적 분석 결과에 대한 분석을 통해 Clang 정적 분석 도구가 난독화로 인해 분석이 어려워진 환경에서 보안 약점 검출 성능이 어떻게 저하되는지 파악할 수 있다.

Clang 정적 분석 도구의 검출 성능은 심볼을 치환하는 방식을 사용하는 구획 난독화 기법과 데이터를 변환하는 방식을 사용하는 데이터 난독화 기법에는 큰 영향을 받지 않았다. 하지만 많은 수의 더미 코드와 복잡한 분기를 사용한 제어 흐름 난독화 기법이 적용된 시험 코드에 대해서는 전반적으로 검출 성능이 크게 감소함을 보였다.

소스 코드 난독화 기법에 따른 Clang 정적 분석 도구의 성능 영향을 분석한 결과를 Clang의 체커 개발 및 성능 개선 과정에서 참고할 수 있을 것이라 생각한다. 더 나아가 난독화가 적용된 시험 코드가 가지는 특성을 이용하여 정적 분석 도구의 평가 기준으로 사용하거나 C/C++ 언어에 대한 시험 코드 개발 과정에서도 활용될 것으로 기대한다.

References

〔1〕 IHS Markit, https://www.ihs.com/industry/telecommunications.html.

〔2〕 IEEE Spectrum, https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2017.

〔3〕 C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," International Symposium on. IEEE, pp. 75-86, Mar. 2004.

〔4〕 Clang, https://clang.llvm.org.

〔5〕 SAMATE Software Assurance Reference Dataset, https://samate.nist.gov/SRD/around.php#juliet_documents.

〔6〕 SAMATE, https://samate.nist.gov/Main_Page.html.

〔7〕 CWE, https://cwe.mitre.org.

〔8〕 C. Collberg, C. Thomborson, and D. Low, "A Taxonomy of Obfuscating Transformations," Technical Report 148, Department of Computer Science, University of Auckland, Jul. 1997.

〔9〕 C. Wang, "A Security Architecture of Survivability Mechanisms," PhD thesis, Univ. of Virginia, School of Eng. and Applied Science, Oct. 2000.

〔10〕 Stunnix, http://stunnix.com/prod/cxxo.

〔11〕 G. Wroblewski, "General Method of Program Code Obfuscation," PhD thesis, Wroclaw University of Technology, Institute of Engineering Cybernetics, Oct. 2002.

〔12〕 Starforce, http://www.star-force.com/products/starforce-obfuscator/

〔13〕 M. Barr, Programming Embedded Systems in C and C++, Sebastopol, California: O'Reilly & Associates, Inc., Jan. 1999.

〔14〕 T. Nakashima, M. Oyama, H. Hisada, and N. Ishii, "Analysis of software bug causes and its prevention," Information

and Software Technology, vol. 41, no. 15, pp. 1059–1068, Dec. 1999.

[15] V. Mashayekhi, J.M. Drake, W.T. Tsai, and J. Riedl, "Distributed, Collaborative Software Inspections," IEEE Software, vol. 10, no. 5, pp. 66-75, Sep. 1993.

[16] R. Shirey, "Internet Security Glossary," Internet Engineering Task Force, RFC 2828, May. 2000.

[17] H. Wang, C. Guo, D. Simon, and A. Zugenmaier, "Shield: Vulnerability-driven network filters for preventing known vulnerability exploits," In Proceedings of ACM SIGCOMM, Portland, OR, Aug. 2004.

[18] Clang Static Analyzer, https://clang-analyzer.llvm.org.

[19] Clang Checker, https://clang-analyzer.llvm.org/available_checks.html.

[20] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," IEEE Software., vol. 25, no. 5, pp. 22–29, Sep. 2008.

[21] FindBugs, http://findbugs.sourceforge.net.

[22] K. Vorobyov and P. Krishna, "Comparing Model Checking and Static Program Analysis: A Case Study in Error Detection Approaches," in Proc. 5th Int. Workshop Syst. Softw. Verification, pp. 1–7. Mar. 2010.

[23] D. Kroening and M. Tautschnig, "CBMC–C bounded model checker," In International Conference on Tools and Algorithms for Construction and Analysis of Systems, LNCS 8413, pp. 389–391, 2014.

[24] C. Cifuentes and B. Scholz, "Parfait – designing a scalable bug checker," In Proceedings of the ACM SIGPLAN Static Analysis Workshop, pp. 4–11, Jun. 2008.

[25] V. Okun, R. Gaucher, and P.E. Black, "Static Analysis Tool Exposition(SATE) 2008," NIST Special Publication 500-279, National Institute of Standards and Technology, Jun. 2009.

[26] V. Okun, A. Delaitre, and P.E. Black, "The Second Static Analysis Tool Exposition (SATE) 2009," NIST Special Publication 500-287, National Institute of Standards and Technology, Jun. 2010.

[27] V. Okun, A. Delaitre, and P.E. Black, editors, "Report on the Third Static Analysis Tool Exposition (SATE) 2010," NIST Special Publication 500-283, National Institute of Standards and Technology, Oct. 2011.

[28] V. Okun, A. Delaitre, and P.E. Black, "Report on the Static Analysis Tool Exposition (SATE) IV," NIST Special Publication 500-297, National Institute of Standards and Technology, Jan. 2013.

[29] SATE V, https://samate.nist.gov/SATE5.html.

[30] D.E. Bakke, R. Parameswaran, D.M. Blough, A.A. Franz, and T.J. Palmer, "Data obfuscation: Anonymity and desensitization of usable data sets," IEEE Security and Privacy, vol. 2, no. 6, pp. 34-41, Nov. 2004.

[31] T.W. Hou, H.Y. Chen and M.H. Tsai, "Three Control Flow Obfuscation methods for Java Software," IEE Proceedings Software, vol. 153, no. 2, pp. 80-86, Apr. 2006.

[32] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," In Proc. 25th. ACM Symposium on Principles of Programming Languages (POPL 1998), pp. 184–196, Jan. 1998.

〈저자소개〉

진 홍 주 (Hongjoo Jin) 학생회원
2017년 8월: 홍익대학교 컴퓨터공학과 졸업
2017년 9월~현재: 고려대학교 정보보호대학원 석사과정
〈관심분야〉 정적 분석, 소프트웨어 난독화, 소프트웨어 보안

박 문 찬 (Moon Chan Park) 학생회원
2013년 2월: 서울시립대학교 수학과 졸업
2015년 2월: 고려대학교 정보보호대학원 석사 졸업
2015년 3월~현재: 고려대학교 정보보호대학원 박사과정
〈관심분야〉 소프트웨어 분석, 소프트웨어 난독화, 소프트웨어 보안

이 동 훈 (Dong Hoon Lee) 종신회원
1983년 8월: 고려대학교 경제학사 졸업
1987년 12월: Oklahoma University 전산학과 석사 졸업
1992년 5월: Oklahoma University 전산학과 박사 졸업
1993년 3월~1997년 2월: 고려대학교 전산학과 조교수
1997년 3월~2001년 2월: 고려대학교 전산학과 부교수
2001년 3월~현재: 고려대학교 정보보호대학원 교수
〈관심분야〉 암호프로토콜, 암호이론, USN이론, 키 교환, 익명성 연구, PET 기술