Journal of The Korea Society of Computer and Information
Vol. 22 No. 3, pp. 17-25, March 2017
www.ksci.re.kr
https://doi.org/10.9708/jksci.2017.22.03.017

Evaluation of Static Analyzers for Weakness in C/C++ Programs using Juliet and STONESOUP Test Suites

Hyunji Seo*, Young-gwan Park**, Taehwan Kim***, Kyungsook Han****, Changwoo Pyo*****

Abstract

In this paper, we compared four analyzers Clang, CppCheck, Compass, and a commercial one from a domestic startup using the NIST's Juliet test suit and STONESOUP that is introduced recently. Tools showed detection efficacy in the order of Clang, CppCheck, the domestic one, and Compass under Juliet tests; and Clang, the domestic one, Compass, and CppCheck under STONESOUP tests. We expect it would be desirable to utilize symbolic execution for vulnerability analysis in the future. On the other hand, the results of tool evaluation also testifies that Juliet and STONESOUP as a benchmark for static analysis tools can reveal differences among tools. Finally, each analyzer has different CWEs that it can detect all given test programs. This result can be used for selection of proper tools with respect to specific CWEs.

▸ Keyword : Static Analyzer, Software Weakness, C/C++ Program, JULIET Test Suite, STONESOUP

I. Introduction

개인 미디어의 발전과 유무선 초고속 인터넷의 발전, 그리고 정보를 지능적으로 처리하는 인공지능 기술의 발전으로 사물인터넷에 대한 관심과 연구가 활성화 되고 있다[1]. 사물인터넷 기술이 적용되는 장비들은 임베디드 시스템으로 구성된다. 임베디드 시스템에 사용되는 언어는 주로 C/C++인데, 이는 효율성 측면에서 다른 언어보다 하드웨어 제어에 특화되어 있으며, 임베디드 시스템의 제한적인 자원 환경을 최대한 효율적으로 이끌어 낼 수 있기 때문이다. 그러나 C/C++ 언어로 작성된 프로그램은 안전하지 않은 타입 시스템과 통제되지 않는 타입 변환, 포인터의 무제한적인 사용, 배열의 불완전한 처리로 인해 예상치 못한 오류가 발생할 수 있다.

C/C++ 프로그램에서 발생한 취약점 사례는 다양한 보안약점을 이용하고 있으며 그 수가 꾸준히 증가하고 있다. [Fig. 1]은 최근 7년간의 C/C++ 관련 CWE[2] 항목별 CVE[3] 발생 건수를 나타낸 그래프이다. 그래프의 추세를 분석해보면 CVE 발생 건수는 계속 증가하고 있으며, 새로운 CWE를 포함하는 CVE 발생 빈도도 높아지고 있다. 이에 따라 C/C++로 작성된 프로그램의 보안약점을 검출할 수 있는 도구의 중요도 역시 높아지고 있으며, 특히 취약점 발생 전 사전 분석이 가능한 정적 분석기를 활용하여 프로그램의 안전성이 검증되어야만 한다.

그러나 정적 분석 방식은 1) 오탐 또는 미탐, 2) 실행 시간 입력을 통한 검증 불가라는 두 가지 이유로 부정확한 결과가 발생할 수 있다. 이를 보완하기 위하여 다양한 분석 기술이 제시되었는데, 기호 실행(symbolic execution)[4]은 입력 값을 실제 구체적인 값이 아닌 기호로 가정하고 프로그램을 해석하며, 요약 해석(abstract interpretation)[5][6]은 분석 대상이 되는 프로그램에

• First Author: Hyunji Seo, Corresponding Author: Changwoo Pyo
*Hyunji Seo(s8488@mail.hongik.ac.kr), Dept. of Computer Engineering, Hongik University
**Young-gwan Park(byg0102@mail.hongik.ac.kr), Dept. of Computer Engineering, Hongik University
***Taehwan Kim(evenstar@mail.hongik.ac.kr), Dept. of Computer Engineering, Hongik University
****Kyungsook Han(khan@kpu.ac.kr), Dept. of Computer Engineering, Korea Polytechnic University
*****Changwoo Pyo(pyo@hongik.ac.kr), Dept. of Computer Engineering, Hongik University
• Received: 2017. 03. 13, Revised: 2017. 03. 17, Accepted: 2017. 03. 23.
• This work was supported by the core technology R&D project of the Agency for Defense Development of Korea(UD160013ED).
• This work was supported by 2016 Hongik University Research Fund

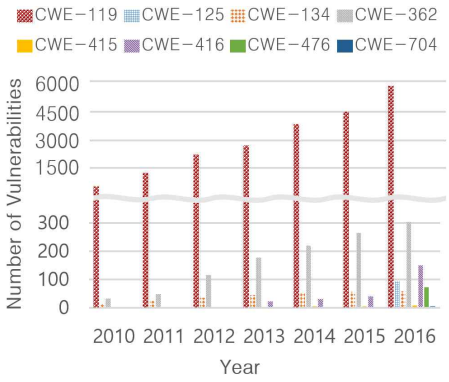


Fig. 1. The number of vulnerabilities with respect to CWE list related to C/C++

서 요약 값들의 변화를 관찰함으로써 프로그램의 실제 실행에 근사한 정보를 얻어내는 것이다. 각 기술들은 대부분 휴리스틱한 분석 방식을 활용하고 있으며, 특히 같은 범주의 기술을 사용하더라도 서로 다른 도구들 간에 사용하는 파라미터에 따라 결과가 달라진다. 그러므로 다른 기술이 적용된 정적 분석기들의 보안약점 탐지율을 비교하여 장단점을 분석하고, C/C++ 프로그램의 보안약점 탐지의 적합성에 대한 확인이 필요하다.

이 논문에서는 4개의 정적 분석기와 이들을 평가하기 위한 2개의 벤치마크를 선정하여 C/C++ 프로그램의 보안약점 탐지에 적합한 기술과 도구를 비교 분석한다. 도구에는 기호 실행 방식을 사용하는 Clang[7], 요약 해석 방식을 사용하는 국내의 Q 정적 분석기(Q 정적 분석기는 국내 상용 분석도구로, 실명을 쓰기에는 문제가 있을 수 있어 Q라고 씀)가 포함되었으며, 이들과는 다른 방식으로 동작하는 CppCheck[8]와 Compass[9]도 추가하였다. 비교 실험에서는 2개의 벤치마크에서 C/C++ 관련 테스트 모음만을 분류하여 평가를 진행하였다. 각 테스트 모음은 벤치마크에 의해 특정 보안약점이 삽입되어 있으므로 이를 정적 분석기가 검출하는지에 대한 여부로 성능을 측정할 수 있다. 각 벤치마크로는 NIST SAMATE(Software Assurance Metrics And Tool Evaluation) 사이트[10]에 게재된 STONESOUP(Securely Taking On New Executable Software of Uncertain Provenance)[11]의 테스트 모음과 줄리엣 테스트 모음(juliet test suites)[12]을 사용하여 실험을 진행하였다.

Table 1. Example programs provided by STONESOUP

Example programs	Description
GNU Tree	Directory listing tool that produces a depth indented listing of files
FFMpeg	Multi-media data processor
Gimp	Image manipulation tool
GNU grep	File search tool
Apache Subversion	Version control system
OpenSSL	Cryptographic library
Postgre SQL	Relational database system
Wireshark	Network protocol analyzer

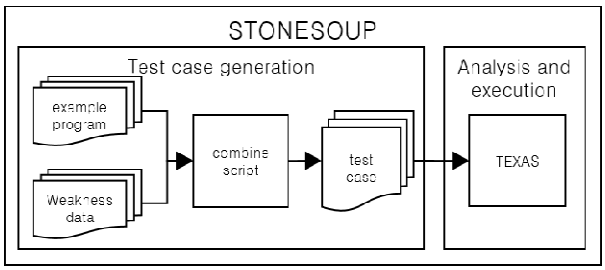


Fig. 2. STONESOUP architecture

이 논문의 구성은 다음과 같다. 2절에서는 2개의 벤치마크에 대하여 설명한다. 3절에서는 평가 대상이 되는 4개의 정적 분석기에 대하여 설명한다. 4절에서는 각 정적 분석기별로 테스트 모음들을 분석한 실험 결과를 정리한다. 5절에서는 실험 결과를 기반으로 각 정적 분석기들 간 비교 및 평가를 한다.

II. Benchmark

1. STONESOUP

NIST SAMATE[10] 프로젝트는 소프트웨어 분석기와 도구에 사용된 기법을 평가하는 방법을 개발하고 이를 통해 분석기의 결함을 검출한다. SAMATE에 게시된 STONESOUP[11]은 사용자의 안전한 프로그램 사용을 위해 출처가 불분명한 프로그램 소스 코드의 보안약점을 분석하고, 이 보안약점을 완화시켜 실행 가능한 파일 형태로 제공해주는 시스템이다. 이 벤치마크는 IARPA[13]에서 진행된 연구 프로젝트 중 하나로, Columbia University, GrammaTech Inc., Kestrel Institute와 같은 단체에서 보안약점을 완화하고 평가하는 과정을 함께 연구하였다[14][15][16]. 전체적인 시스템 구조는 [Fig. 2]와 같이 테스트 케이스 생성부와 분석 및 실행부로 나누어 볼 수 있다.

테스트 케이스 생성부에서는 소스 코드 형태의 예제 프로그램과 보안약점을 내포하는 파일들의 조합으로 테스트 케이스를 생성한다. 예제 프로그램으로는 C로 작성된 프로그램 8개가 있

Table 2. CWE list related to C/C++ in STONESOUP

Weakness	CWE ID
Concurrency handling	363 367 412 414 543 609 663 764 765 820 821 833 831 828 479
Injection	078 088 089
Number handling	190 191 194 195 196 197 369 682 839
Resource drains	400 459 674 774 789 834 835 401 771 773 775
Memory corruption	120 124 126 127 129 134 170 415 416 590 761 785 805 806 822 824 843
Null pointer	476

으며 [Table 1]에서 이 프로그램 목록을 확인할 수 있다. 보안약점 데이터는 [Table 2]에서 확인할 수 있는데, 이는 STONESOUP 내에서 자체적으로 분류한 C 언어 관련 6개의 보안약점 항목과 관련 56개의 CWE ID에 대한 분류표이다.

분석 및 실행부에서는 TEXAS(Test and Evaluation eXecution and Analysis System)를 통해 진행된다. TEXAS는 생성된 테스트 케이스를 분석하고 보안약점을 완화시킨 실행가능 파일로 변환하는 과정을 돕는 프레임워크로, 크게 분석과 실행 과정으로 나뉜다. 이 실험에서는 정적 분석기의 평가를 위해 TEXAS를 배제하고 STONESOUP의 테스트 케이스 생성부만을 사용하였다.

2. Juliet test suites

줄리엣 테스트 모음[12]은 미국 국가안보국의 CAS (Center for Assured Software)에서 개발되었다. 미국 국립표준기술연구소에서 정적 분석기의 성능을 평가하기 위한 시험코드로 줄리엣 테스트 모음을 채택했다. 이 연구소의 보고서[17]에 따르면 줄리엣 테스트 모음을 활용하여 평가한 정적 분석기는 CppCheck[8], LDRA Testbed[18], INFER[19], Parasoft C++ test[20], Red Lizard Software Goanna[21] 5개가 존재한다.

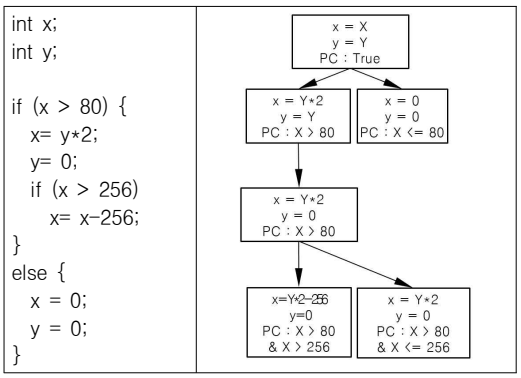
줄리엣 테스트 모음은 CWE 항목별로 분류되어 있으며, CWE 내에서 다양한 유형으로 구성되어 있다. 코드가 많기 때문에 정적 분석기를 평가할 때 보안약점 검출 성능에 대한 구체적인 결과를 얻을 수 있다. 또한 줄리엣 테스트 모음은 다른 정적 분석기와 객관적인 성능 비교를 가능하게 한다. 줄리엣 테스트 모음은 독립성을 가지며 CWE를 기반으로 JAVA와 C/C++로 나뉜다. 이 논문에서는 C/C++만을 대상으로 하기 때문에 JAVA와 관련된 코드는 제외하였다. C/C++를 대상으로 하는 줄리엣 테스트 모음에는 61,387개의 코드가 존재하고 CWE는 119개 항목이 존재한다.

III. Static analyzer

1. Clang static analyzer

Clang[7] 프로젝트는 LLVM[22][23]의 전단부를 위한 프로젝트로 컴파일러와 정적 분석기를 포함한다. 소스는 일리노이 대학교 오픈 소스 라이선스로 이용할 수 있다. 운영체제는 맥 OS, 리눅스, 윈도우 모두 이용 가능하다. 이 실험에서는 운영체제로 리눅스를 사용했고 버전은 3.7.0을 사용했다. 이후 문서에서 언급하는 Clang은 정적 분석기만을 말한다.

Clang에서는 기호 실행[4]을 이용하여 코드에 대한 정적 분석을 수행한다. 기호 실행은 [Fig. 3]의 (b)에서 볼 수 있듯이 입력 값을 실제 구체적인 값이 아닌 기호로 가정한다. 프로그램



(a) source code (b) symbolic execution tree

Fig. 3. Example of symbolic execution

과 실행 경로를 동시에 해석하며 분기문을 만나면 기호에 대한 제약식을 생성하여 덧붙이고 모든 경로에 대해 분석을 진행한다. 예를 들어, (a)의 첫 번째 if문을 만나면 (b)의 기호 실행 트리에서 참인 경로와 거짓인 경로로 나눠지게 되고 경로 조건(PC, Path Condition)이 추가된다. 이후 나눠진 경로 모두에 대해 분석을 진행한다. 그러므로 결함이 검출될 때까지 해석한 프로그램 상태를 통해 보안약점의 유형을 파악할 수 있다.

Clang의 체커는 크게 Alpha 집합과 Default 집합으로 나누어져있다. Alpha 집합은 개발 중인 체커들의 집합으로 구성되어 있고 Default 집합은 Alpha 집합에서 검증이 끝난 체커들로 구성되어 있으며 core, cpluscplus, deadcode, security, unix, osx 6개의 패키지로 구성되어 있다. Clang의 체커는 총 96개가 있으며, 그 중 C/C++에 관련된 56개 체커들로 검출할 수 있는 CWE 목록은 [Table 3]과 같다.

2. Q static analyzer

Q는 그래프 DB에 대한 질의어를 통해 보안약점을 검사하는 국내에서 개발된 상용 도구이다. 온더플라이 정적분석 프레임워크를 통한 결함 추적 시스템을 통하여 분석과 결함 추적을 실시간으로 연동해 빠르게 분석하는 것을 목표로 하고 있다. 현재 온더플라이 정적분석 기능은 개발 중인 상태이다.

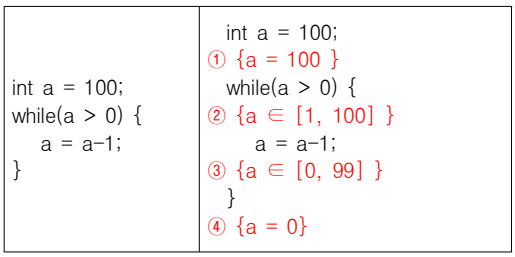
Q에서는 C/C++ 및 JAVA로 이루어진 코드를 대상으로 정적 분석을 수행하며, 행정자치부에서 공고한 47개의 주요 보안약점을 탐지한다. 또한 사용자를 관리자, PM, 일반 사용자 계정으로 구분하여 기능을 부여함으로써 프로젝트 관리 개념을 도입하였으며, 웹 기반으로 접속 가능한 IP를 등록함으로써 미리 등록된 사용자만 접근 가능하도록 하여 보안성을 높였다. 보안 검사 이후에는 프로그램 분석 후 분석 대상 파일 정보, 분석 결과 요약 정보와 카테고리에 따른 분석 결과, 경고 목록, 경고에 대한 상세 정보 및 시큐어코딩 기반 수정 가이드를 출력해 줌으로써 사용자가 필요로 하는 정보를 찾아보기 편리하도록 시각적인 인터페이스를 구성하고 있다.

Q에서는 추상 구문 트리나 제어 흐름 그래프, 데이터 흐름 그래프와 같은 그래프 정보를 기반으로 질의어를 수행하며 조

Table 3. CWE list that can be evaluated by Clang

CWE ID	CWE Name	CWE ID	CWE Name
CWE-077	Improper Neutralization of Special Elements used in a Command	CWE-456	Missing Initialization of a Variable
CWE-120	Buffer Copy without Checking Size of Input ('Classic Buffer Overflow')	CWE-457	Use of Uninitialized Variable
CWE-121	Stack-based Buffer Overflow	CWE-463	Deletion of Data Structure Sentinel
CWE-122	Heap-based Buffer Overflow	CWE-466	Return of Pointer Value Outside of Expected Range
CWE-131	Incorrect Calculation of Buffer Size	CWE-467	Use of sizeof() on a Pointer Type
CWE-135	Incorrect Calculation of Multi-Byte String Length	CWE-416	Use After Free
CWE-170	Improper Null Termination	CWE-468	Incorrect Pointer Scaling
CWE-190	Integer Overflow or Wraparound	CWE-469	Use of Pointer Subtraction to Determine Size
CWE-193	Off-by-one Error	CWE-476	NULL Pointer Dereference
CWE-194	Unexpected Sign Extension	CWE-477	Use of Obsolete Functions
CWE-195	Signed to Unsigned Conversion Error	CWE-561	Dead Code
CWE-227	Improper Fulfillment of API Contract ('API Abuse')	CWE-562	Return of Stack Variable Address
CWE-242	Use of Inherently Dangerous Function	CWE-563	Assignment to Variable without Use ('Unused Variable')
CWE-243	Creation of chroot Jail Without Changing Working Directory	CWE-570	Expression is Always False
CWE-244	Improper Clearing of Heap Memory Before Release ('Heap Inspection')	CWE-571	Expression is Always True
CWE-250	Execution with Unnecessary Privileges	CWE-587	Assignment of a Fixed Address to a Pointer
CWE-266	Incorrect Privilege Assignment	CWE-590	Free of Memory not on the Heap
CWE-271	Privilege Dropping/Lowering Errors	CWE-665	Improper Initialization
CWE-272	Least Privilege Violation	CWE-667	Improper Locking
CWE-330	Use of Insufficiently Random Values	CWE-676	Use of Potentially Dangerous Function
CWE-331	Insufficient Entropy	CWE-690	Unchecked Return Value to NULL Pointer Dereference
CWE-332	Insufficient Entropy in PRNG	CWE-704	Incorrect Type Conversion or Cast
CWE-337	Predictable Seed in PRNG	CWE-754	Improper Check for Unusual or Exceptional Conditions
CWE-338	Use of Cryptographically Weak Pseudo-Random Number Generator	CWE-762	Mismatched Memory Management Routines
CWE-369	Divide By Zero	CWE-763	Release of Invalid Pointer or Reference
CWE-377	Insecure Temporary File	CWE-764	Multiple Locks of a Critical Resource
CWE-400	Uncontrolled Resource Consumption ('Resource Exhaustion')	CWE-765	Multiple Unlocks of a Critical Resource
CWE-401	Improper Release of Memory Before Removing Last Reference	CWE-805	Buffer Access with Incorrect Length Value
CWE-404	Improper Resource Shutdown or Release	CWE-824	Access of Uninitialized Pointer
CWE-415	Double Free	CWE-908	Use of Uninitialized Resource

건에 일치하는 정보를 찾거나 새로운 그래프 정점 및 간선을 추가하며 분석을 진행한다. 값의 범위에 대한 분석에는 요약 해석 방식[5][6]을 사용하며, 이는 분석 대상이 되는 프로그램에서 실제 도메인을 요약한 요약 도메인을 통해 요약 값들의 변화를 관찰함으로써 프로그램의 실제 실행에 근사한 정보를 얻어내는 방식이다. [Fig. 4]는 요약 해석의 과정을 나타내는 예이다. (a)는 변수 a가 반복문을 통해 100에서 0까지의 값을 갖게 되는 예제 코드이며, (b)에서 이 a 변수가 가질 수 있는 값의 범위를 나타내었다. 나타낸 값의 범위로부터 요약 도메인이 생성된다. 요약 해석 방식은 프로그램의 실제 실행 의미를 요약하기 때문에 프로그램 상의 오류를 많이 검출해낼 수 있다는 것이 장점이다. 단, C 언어 관련 부분은 라이브러리가 존재하지 않는 경우 검사가 이루어지지 않는 문제를 가지고 있다.



(a) source code (b) abstract interpretation

Fig. 4. Example of abstract interpretation

3. CppCheck static analyzer

Daniel Marjamaki가 처음 만들고 주도적으로 개발한 CppCheck[7]는 C/C++ 언어 대상으로 분석을 수행하는 정적 분석기로 주로 메모리 누수, 초기화하지 않은 변수, 사용하지 않는 함수와 같은 항목을 검출하며 다량의 파일을 빠르게 검사하기 위해 사용된다. 이클립스 플러그인 버전과 GUI 버전으로 사용할 수 있으며 분석 결과는 XML, CSV, HTML 형식을 지원한다. 크로스 플랫폼 기반으로 운영되며 대부분의 운영체제에서 사용이 가능하다.

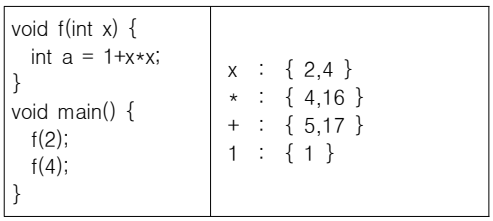
CppCheck는 제어 흐름 분석 방식[24]을 사용하여 대상 코드로부터 제어 흐름 그래프를 얻어내고 이를 분석한다. 제어 흐름 그래프는 프로시저 간의 함수 호출이나 분기 명령과 같이 프로그램의 실행에 따라 달라지는 실행 경로들을 포함하여 생성된다.

CppCheck에서 소스 파일은 전처리기를 거친 후에 토큰화되는데, 예를 들어 'abc=ab+c;'와 같은 코드는 'abc', '=', 'ab', '+', 'c', ';' 와 같이 6개의 토큰으로 나누어진다. 이후 체커는 토큰 리스트를 순회하면서 오류를 탐지한다. 이 때 토큰들은 구문트리로 관리된다. 체커가 실행되기 전에 CppCheck는 가능한 값을 추적하고 이 값은 토큰에 저장된다.

[Fig. 5]의 (b)는 (a)에 있는 함수 f의 토큰이 가질 수 있는 값들을 나타낸다. f(2) 일 때, 'x'는 2를 가질 수 있으며 '*', '+'는 각각 '4', '5'를 가질 수 있다. 같은 방식으로 f(4)에서도 가능한 값을 구할 수 있다. 이러한 방식을 활용하여 0으로 나눗셈 연산을 수행하려는 경우에 대한 보안약점을 검출할 수 있게 된다.

4. Compass static analyzer

Compass[9]는 LLNL(Lawrence Livemore National Laboratory)에서 개발한 로즈 컴파일러[25]의 전단부를 사용하여 작동하는 정적 분석기이다. 로즈 컴파일러는 중간표현으로 AST를 사용하며 C/C++, Fortran, JAVA, OpenMP와 같은 언어를 지원한다. 많은 도구들이 보안약점 기반의 검사를 수행하는데 비해, Compass는 CERT 코딩 표준을 기반으로 해당 표준의 준수 여부를 검사하기 위한 목적을 가지며, 메모리나 문자열, 입출력 관련 검사 기능이 우수하다. 로즈 컴파일러를 이용하여 대상 코드를 파싱하고, 체커를 설정함에 따라 어떤 코딩



(a) source code (b) Set of values in the token of the function f

Fig. 5. Example of tokenizing code in CppCheck

규칙에 대한 검사를 수행할지 결정하게 된다. 이후 검사 결과를 얻어내면 이를 보완하여 안전한 실행 파일까지 생성할 수 있다.

Compass는 C/C++, Fortran으로 된 소스 코드에 대한 검사를 101개의 체커를 통해 수행하는데, 이 때 체커로 검사할 수 있는 대상은 소스 코드뿐만이 아니라 오브젝트 파일도 가능하다. Compass 체커들의 집합은 기본적으로 제공되는 체커들과 이후에 추가된 사용자 정의 체커들로 이루어진다. 체커들은 사용자가 확장할 수 있으며, AST에 영향을 주지 않는 범위에서 분석 모듈을 작성해 체커 목록에 추가 요청을 할 수 있다.

IV. Experimental process and results

이 장에서는 3장에서 설명한 정적 분석기 4개를 사용하여 2개의 벤치마크에 대한 평가 결과를 보인다. 각 벤치마크에서 C/C++ 관련 항목에 대한 테스트 모음만을 사용하였으며 분석기별 요구하는 환경에 따라 실험을 수행했다. 실험을 위해 각 벤치마크에서 평가 대상으로 하는 CWE 항목마다 테스트 모음을 생성 및 정리하였다. 줄리엣 테스트 모음의 경우 독립적인 코드로 존재하지만, STONESOUP의 경우 실험 환경에 맞게 테스트 모음을 생성할 필요가 있다.

STONESOUP 테스트 케이스 생성부에서는 예제 프로그램과 보안약점 데이터를 스크립트를 통해 결합시켜 테스트 모음을 생성한다. [Fig. 2]에서의 예제 프로그램은 [Table 1]에서 8개의 프로그램에 해당하며, diff 파일 형태로 구성된 보안약점 데이터는 결합 스크립트에 의해 예제 프로그램을 패치하는데 사용된다. 정적 분석기의 평가를 위해서는 문제가 되는 소스 코드 및 이와 관련된 헤더 파일만이 필요하므로 이외의 파일은 전부 삭제하는 과정을 결합 스크립트에 추가하였다. 생성된 테스트 모음은 같은 보안약점을 가지는 집합 단위로 재구성하여 분석을 수행하였다. [Table 4]와 [Table 5]는 각 정적 분석기를 통해 STONESOUP과 줄리엣 테스트 모음에서 평가 가능한 보안약점과 맵핑한 CWE 항목을 나타낸다.

평가 기준은 분석기가 벤치마크에 포함된 CWE 항목을 정확히 검출할 때에만 정탐으로 분류하였다. 예를 들어, STONESOUP 테스트 모음이 생성될 때 CWE-476이 포함되어 있다면, 정적 분석기가 삽입된 보안약점 코드 위치에서 CWE-476

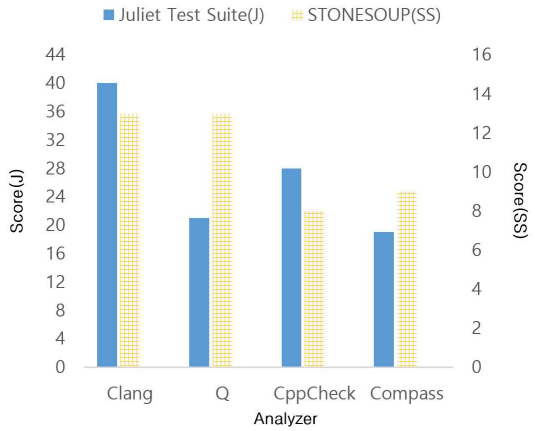


Fig. 6. Scoring records based on true positive rate for CWE list

을 검출해야만 정탐이 된다. 즉, 검출된 CWE 항목의 코드 위치가 다르거나 다른 CWE 항목을 검출하면 정탐이 아니므로 오탐 가능성이 있다고 볼 수 있다.

[Table 6]은 2개의 벤치마크에서 생성된 CWE 항목들에 대한 각 도구별 정탐율을 나타낸다. 벤치마크는 줄리엣 테스트 모음의 경우 J, STONESOUP의 경우 SS로 축약하여 분류하였고, CWE 항목에 대하여 검출하는 코드 개수와 전체 코드에 대한 정탐율을 괄호 안에 넣어 표기하였다. 도구에 해당 CWE를 검출할 수 있는 탐지 항목이 없으면 빈칸으로 표시하고, 모든 도구에 대하여 정탐율이 0%로 나타나는 CWE 항목은 표에 포함하지 않았다. [Table 6]에는 총 35개의 CWE에 대한 비교를 나타내었다.

이 논문에서는 분석기 간 성능 평가 기준으로 1) 3개 이상의 정적 분석기가 검출한 CWE 항목들을 선정하고, 2) 정탐율의 순위에 따라 4점부터 1점까지 점수를 부여하는 방식을 활용하였다. 예를 들어, [Table 6]에서의 CWE-401은 Clang, Q, CppCheck가 검출 가능하므로 평가 기준에 포함된다. 선정된 CWE 항목에는 줄리엣 테스트 모음 기준으로 CWE-121, CWE-122, CWE-195, CWE-242, CWE-401, CWE-415, CWE-416, CWE-467, CWE-476, CWE-562, CWE-571의 총 11개, STONESOUP 기준으로 CWE-401, CWE-415, CWE-416, CWE-476의 총 4개가 존재한다. 정탐율 순위에 따라 점수를 부여하면 객관적인 정적 분석기의 성능을 비교할 수 있다. CWE-195를 예로 들면, 정탐율이 1위인 Clang에 4점을 부여하고 CppCheck, Compass가 공동 2위이므로 각각 3점을 부여한다.

[Fig. 6]은 비교분석한 결과를 보여주는 그래프이다. 줄리엣 테스트 모음에서는 11개의 항목이 있으므로 만점이 44점이며, STONESOUP 테스트 모음은 4개의 항목이 있으므로 16점 만점이다. Clang이 줄리엣 테스트 모음을 대상으로 받은 점수는 CWE-195, CWE-242, CWE-415, CWE-416, CWE-467, CWE-476, CWE-562의 7개 항목에서 4점, CWE-121, CWE-122, CWE-401, CWE-571의 4개 항목에서 3점을 받아 총 40점으로 가장 높았다. 또한, STONESOUP 테스트 모음에서 받은 점수는 CWE-416 항목에서 4점, CWE-401, CWE-415, CWE-467의 3개 항목에서 3점을 받아 총 13점으로 가장 높았다. Q가 STONESOUP을 대상으로 받은 점수는 CWE-401, CWE-415의 2개 항목에서 4점, CWE-416 항목에서 3점, CWE-467 항목에서 2점을 받아 총 13점으로 Clang과 동점이다.

[Fig. 6]에서 볼 수 있듯이 Clang은 정탐율이 가장 높으며, [Table 6]에서도 다른 도구에 비해 분석 가능한 CWE 항목이 많다. 특히, CWE-242, CWE-467, CWE-562에 대한 모든 줄리엣 테스트 모음에서 100%의 정탐율을 보였다.

Q는 줄리엣 테스트 모음 기준으로 CWE-242에 대해서 100%의 정탐율을 보였다. STONESOUP 테스트 모음 기준으로는 CWE-415, CWE-674에 대해서 100%의 정탐율을 보인다.

CppCheck는 줄리엣 테스트 모음의 CWE-467, CWE-482, CWE-561, CWE-685에 대해서는 100%의 정탐율을 보인다. 줄리엣 테스트 모음 분석 결과 Clang에 비해 낮은 정탐율을 가진다. 그러나 Clang에서 분석하지 못한 CWE(e.g. CWE-482)에 대해서 분석이 가능하다.

Compass는 다른 도구에 비하여 분석 가능한 CWE 항목도 적고 정탐율도 낮은 수치를 갖는다. 그러나 줄리엣 테스트 모음에서는 CWE-242, CWE-478, CWE-587에 대해서는 100%의 정탐율을 보였다.

Table 4. CWE list that test codes generated by STONESOUP contain

Analyzer	CWE ID
Clang	124 126 127 194 195 369 400 401 415 416 476 764 765
Q	78 89 134 190 367 401 415 416 476 674
CppCheck	120 170 190 195 197 369 401 415 476 590 682 771 834
Compass	129 190 195 196 415 416 476 674 682 828 831

Table 5. CWE list that juliet test suite contains

Analyzer	CWE ID
Clang	121 122 124 126 127 194 195 242 369 377 400 401 404 415 416 457 467 468 469 476 561 562 563 571
Q	121 122 134 242 244 401 415 416 457 469 476 480 676
CppCheck	121 122 190 195 197 252 369 398 401 404 415 467 475 476 482 561 562 563 570 571 587 590 665 676 685 758 762
Compass	121 122 190 195 196 242 252 364 377 415 416 467 476 478 562 571 587 665 674 676 680 681 690

Table 6. Evaluation of Clang, Q, CppCheck, and Compass with juliet test suite (J) and test codes generated by STONESOUP (SS)

CWE	CWE Name	Total number of codes		Clang		Q		CppCheck		Compass	
		J	SS	J	SS	J	SS	J	SS	J	SS
78	OS Command Injection	4800	359				8 (2.2%)				
89	SQL Injection		357				32 (9.0%)				
121	Stack-based Buffer Overflow	4968		657 (13.2%)		1618 (32.6%)		7 (0.1%)		0 (0.0%)	
122	Heap-based Buffer Overflow	5922		972 (16.4%)		1432 (24.2%)		0 (0.0%)		0 (0.0%)	
124	Buffer Underflow	2048	95	458 (22.4%)							
126	Buffer Over-read	1452	92	27 (1.9%)							
127	Buffer Under-read	2048	95	422 (20.6%)							
134	Use of Externally-Controlled Format String	2880	23			13 (0.5%)	0 (0.0%)				
188	Reliance on Data/Memory Layout	36		18 (50.0%)							
194	Unexpected Sign Extension	1152	66	162 (14.1%)	0 (0.0%)						
195	Signed to Unsigned Conversion Error	1152	66	244 (21.2%)	0 (0.0%)			0 (0.0%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
242	Use of Inherently Dangerous Function	18		18 (100%)		18 (100%)				18 (100%)	
369	Divide By Zero	864	66	54 (6.3%)	0 (0.0%)			14 (1.6%)	0 (0.0%)		
377	Insecure Temporary File	144		36 (25.0%)						0 (0.0%)	
398	Indicator of Poor Code Quality	181						36 (19.9%)			
400	Resource Exhaustion	720	122	280 (38.9%)	0 (0.0%)						
401	Memory Leak	1658	61	74 (4.5%)	0 (0.0%)	53 (3.2%)	14 (23.0%)	308 (18.6%)	0 (0.0%)		
404	Improper Resource Shutdown or Release	384		87 (22.7%)				0 (0.0%)			
415	Double Free	962	24	594 (61.8%)	19 (79.2%)	120 (12.5%)	24 (100%)			0 (0.0%)	0 (0.0%)
416	Use After Free	459	24	383 (83.4%)	17 (70.8%)	116 (25.3%)	0 (0.0%)	341 (74.3%)	0 (0.0%)	0 (0.0%)	0 (0.0%)
457	Use of Uninitialized Variable	948		415 (43.8%)		216 (22.8%)					
467	Use of sizeof() on a Pointer Type	54		54 (100%)				54 (100%)		0 (0.0%)	
468	Incorrect Pointer Scaling	37		0 (0.0%)							
476	NULL Pointer Dereference	348	693	246 (70.7%)	347 (50.1%)	83 (23.9%)	0 (0.0%)	103 (29.6%)	84 (12.1%)	15 (4.3%)	475 (68.5%)
478	Missing Default Case in Switch Statement	18								18 (100%)	
482	Comparing instead of Assigning	18						18 (100%)			
561	Dead Code	2		1 (50.0%)				2 (100%)			
562	Return of Stack Variable Address	3		3 (100%)				1 (33.3%)		0 (0.0%)	

563	Unused Variable	512		240 (46.9%)				236 (46.1%)			
570	Expression is Always False	16						3 (18.8%)			
571	Expression is Always True	16		1 (6.3%)				3 (18.8%)		0 (0.0%)	
587	Assignment of a Fixed Address to a Pointer	18						0 (0.0%)		18 (100%)	
590	Free of Memory not on the Heap	2680	23					233 (8.7%)	0 (0.0%)		
674	Uncontrolled Recursion	2	62				62 (100%)				
685	Function Call With Incorrect Number of Arguments	18						18 (100%)			
762	Mismatched Memory Management Routines	3564						1588 (44.6%)			

V. Conclusion

이 논문에서는 4개의 정적 분석기를 대상으로 2개의 벤치마크를 선정하여 각 도구별 C/C++ 프로그램의 보안약점 탐지 성능을 평가 및 비교 분석하였다. 벤치마크는 줄리엣과 STONESOUP 테스트 모음을 활용하였으며, 이들이 커버하는 C/C++ 관련 CWE는 STONESOUP 테스트 모음이 56개, 줄리엣 테스트 모음이 119개이다.

벤치마크로부터 얻을 수 있는 탐지 가능한 CWE 항목이 서로 다르므로 정적 분석기들 간의 완전한 비교는 어렵지만, 줄리엣 테스트 모음 분석 실험에서의 수치 결과로 미루어 보면 기호 실행을 사용한 Clang이 전반적으로 가장 우수한 정탐율을 보였다. Clang은 검출 가능한 CWE 항목을 대상으로 정탐율을 측정한 결과 [Fig. 6]에서 가장 높은 점수인 40점과 13점을 보였다. 특히 줄리엣 테스트 모음에서는 CWE 항목 24개 중 23개를 탐지 가능했으며, 이 중 3개에서는 100%의 정탐율을 보였다. 이는 기호 실행 방식이 정적 분석에 효과적임을 증명한다.

CppCheck는 줄리엣 테스트 모음에서의 CWE 중 4개에서 100%의 정탐율을 보였으나, 탐지 가능한 항목은 20개 중 16개로 Clang보다 낮은 수치를 보였다. Q의 경우에는 CWE 항목 중 줄리엣 테스트 모음에서 1개, STONESOUP 테스트 모음에서 2개로 총 3개의 CWE에서 100%의 정탐율을 보였다. Q는 줄리엣 테스트 모음을 대상으로 CWE 항목 9개 모두에서 탐지가 가능하였으며, 다른 도구에서는 잘 검출하지 못한 STONESOUP 테스트 모음의 CWE도 높은 탐지율을 보이는 경우가 있었다. Compass는 [Fig. 6]에서 가장 낮은 점수인 19점을 보였다. 그러나 줄리엣 테스트 모음을 대상으로 CWE 항목 중 3개에 대해 100%의 정탐율을 보여주어 탐지 가능한 개수 대비 정탐율이 가장 높았다.

각 CWE 항목별 100%의 정탐율 결과는 보안약점에 따라 도구를 구분해서 활용하면 효과적인 탐지가 가능함을 보여준다. 반대로 정탐율이 낮거나 0%인 경우는 분석기의 탐지 알고리즘이 개선되거나 수정이 필요하다고 판단할 수 있다. 정탐율이 0%인 경우는 다음의 두 가지 상황으로 나누어 볼 수 있는데, 첫 번째는 외부로부터 유입되거나 동적으로 생성되는 값에 대한 분석에 대응하지 못하여 미탐이 발생하는 경우이다. 두 번째는 최근 추가된 보안약점에 대한 대응 미비이다. 첫 번째 경우에 대해서는 도구에 동적 분석을 함께 실행하는 하이브리드 분석 기법을 활용하면 보완이 가능하다. 두 번째 경우에 대해서는 제로-데이 취약점 사례를 유발하는 보안약점을 분석하여 이에 대응하는 업데이트를 지속적으로 수행하면 탐지율이 나아질 수 있다.

정적 분석기들은 공통적으로 STONESOUP 테스트 모음을 대상으로는 보안약점을 거의 검출하지 못하였다. STONESOUP 테스트 모음은 줄리엣 테스트 모음과 비교하여 보안약점을 내포한 코드 구조가 다소 복잡하고 여러 개의 파일들이 연결된 구성을 보일 때도 있어, 이러한 형태의 테스트 모음을 검사할 때는 도구들의 정탐율이 급격히 떨어졌다. STONESOUP 테스트 모음 중 100%의 정탐율이 나타난 경우는 예외적으로 단순한 구조를 가질 때였으며, 이러한 결과로부터 도구들의 분석 알고리즘이 복잡한 코드 구조에도 대응할 수 있도록 개선되어야 함을 확인하였다.

향후에는 정적 분석기 간 성능 비교 결과를 토대로 다양한 도구의 특성과 보안약점 분류에 따른 실험으로 확장하고자 한다. 이를 통해 도구나 보안약점 특성에 따른 효율성 및 활용도에 대한 더 자세한 평가가 가능하리라 기대한다.

REFERENCES

[1] C. Joo, and H. Na, “A Study of Research Trend about Internet of Things,” NIA(National Information society Agency), Vol. 22, No. 3, pp.3-15, Autumn 2015

[2] CWE, Common Weakness Enumeration, *http://cwe.mitre.org*

[3] CVE, Common Vulnerabilities and Exposures, *http://cve.mitre.org*
[4] C. Cadar, and K. Sen, "Symbolic execution for software testing: three decades later," Communications of the ACM, 56.2, pp.82-90, July 2013.
[5] P. Cousot, and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp.238-252, ACM, January 1977.
[6] S. Hendrik, and S. Kowalewski, "Static analysis of Sequential Function Charts using abstract interpretation," Emerging Technologies and Factory Automation (ETFA), pp.1-4, 2016 IEEE 21st International Conference on, September 2016.
[7] Clang, *http://Clang-analyzer.llvm.org*
[8] CppCheck, *http://CppCheck.sourceforge.net*
[9] Compass User Manual, *http://rosecompiler.org/Compass.pdf*
[10] NIST, *http://samate.nist.gov/SRD/testsuite.php*
[11] IARPA, STONESOUP(Securely Taking On New Executable Software of Uncertain Provenance)
[12] SAMATE, Juliet Test Suite v1.2 for C/C++ User Guide, National Security Agency
[1] [13] IARPA, *http://www.iarpa.gov*
[14] MINESTRONE, *http://nsl.cs.columbia.edu/projects/m- inestrone*
[15] PEASOUP, *http://www.grammatech.com/software-hardening/research*
[16] VIBRANCE, *http://stonesoup.kestrel.edu*
[17] NIST, Report on the Static Analysis Tool Exposition (SATE) IV
[18] LDRA Testbed, *http://www.ldra.com/en/testbedtbvision*
[19] INFER, *http://fbinfer.com*
[20] Parasoft C++ test, *http://www.parasoft.com/product/static-analysis-cc*
[21] Red Lizard Software Goanna, *http://redlizards.com*
[22] C. Lattner, and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization, pp.75, IEEE Computer Society, March 2004.
[23] B. C. Lopes, and R. Auler, "*Getting started with LLVM core libraries*," Packt Publishing Ltd, pp.73-104, 2014.
[2] [24] K. Cooper, and L. Torczon, "*Engineering a compil- er*," Elsevier, pp.231-232, 2011.
[25] ROSE compiler infrastructure, *http://rosecompiler.org*

Authors

Hyunji Seo received the B.S. degrees in Computer Engineering from Hongik University, Korea, in 2017. She is currently a master student in Hongik University. She is interested in software security and program testing.

Young-gwan Park received the B.S. degrees in Computer Engineering from Hongik University, Korea, in 2017. He is currently a master student in Hongik University. He is interested in software security, parallel computing and static analy- sis.

Taehwan Kim received the B.S., M.S. degrees in Computer Engineering from Hongik University, Korea, in 2012, and 2014. He is currently a Ph.D. candidate in Hongik University. He is interested in program instrumentation and software security.

Kyungsook Han received the B.S., M.S. and Ph.D. degrees in Computer Engine- ering from Hongik University, Korea, in 1993, 1995 and 2002. Dr. Han joined the faculty of the Department of Computer Engineering at Korea Polytechnic University, Gyeonggi-Do, Korea, in 2003. She is curre- ntly an Associate Professor in the Department of Computer Engineering, Korea Polytechnic University. She is interested in compiler optimization and software security.

Changwoo Pyo received the B.S. degree in Electronic Engineering and the M.S. degree in Computer Engineer- ing from Seoul National University in 1980 and 1982. He received the Ph.D. degree in Computer Science from the University of Illinois at Urbana-Champaign in 1989. Dr. Pyo joined the faculty of the Department of Computer Engineering at Hongik University, Seoul, Korea, in 1991. He is currently a Professor in the Department of Computer Engineering, Hongik University. He is interested in programming language, software security, program optimization, and parallel computing.