KIPS Tr. Comp. and Comm. Sys.
Vol.5, No.4 pp.79~86 pISSN: 2287-5891

http://dx.doi.org/10.3745/KTCCS.2016.5.4.79

Development of Safe Korean Programming Language Using Static Analysis

Dohun Kang† · Yeoneo Kim†† · Gyun Woo†††

ABSTRACT

About 75% of software security incidents are caused by software vulnerability. In addition, the after-market repairing cost of the software is higher by more than 30 times than that in the design stage. In this background, the secure coding has been proposed as one of the ways to solve this kind of maintenance problems. Various institutions have addressed the weakness patterns of the standard software. A new Korean programming language Saesark has been proposed to resolve the security weakness on the language level. However, the previous study on Saesark can not resolve the security weakness caused by the API. This paper proposes a way to resolve the security weakness due to the API. It adopts a static analyzer inspecting dangerous methods. It classifies the dangerous methods of the API into two groups: the methods of using tainted data and those accepting in-flowing tainted data. It analyses the security weakness in four steps: searching for the dangerous methods, configuring a call graph, navigating a path between the method for in-flowing tainted data and that uses tainted data on the call graph, and reporting the security weakness detected. To measure the effectiveness of this method, two experiments have been performed on the new version of Saesark adopting the static analysis. The first experiment is the comparison of it with the previous version of Saesark according to the Java Secure Coding Guide. The second experiment is the comparison of the improved Saesark with FindBugs, a Java program vulnerability analysis tool. According to the result, the improved Saesark is 15% more safe than the previous version of Saesark and the F-measure of it 68%, which shows the improvement of 9% point compared to 59%, that of FindBugs.

Keywords : Secure Coding, Korean Programming Language, Safety Programming Language, Saesark

정적 분석을 이용한 안전한 한글 프로그래밍 언어의 개발

강 도 훈† · 김 연 어†† · 우 균†††

요 약

소프트웨어 보안 사고의 약 75%는 소프트웨어 취약점으로 인해 발생한다. 또한, 제품 출시 후 결함 수정 비용은 설계 단계의 수정 비용보다 30배 이상 많다. 이러한 배경에서, 시큐어 코딩은 유지 보수 문제를 해결하는 방법 중 하나로 제안되었다. 다양한 연구 기관에서는 소프트웨어 보안 약점의 표준 양식을 제시하고 있다. 새로운 한글 프로그래밍 언어 새싹은 언어 수준에서 보안 약점 해결 방법을 제안하였다. 그러나 이전 연구의 새싹은 API에 관한 보안 약점을 해결하지 못하였다. 본 논문에서는 API에 의한 보안 약점을 해결하는 방법을 제안한다. 이 논문에서 제안하는 방법은 새싹에 위험한 메소드를 검사하는 정적 분석기를 적용하는 것이다. 위험한 메소드는 오염된 데이터 유입 메소드와 오염된 데이터 사용 메소드로 분류한다. 분석기는 위험한 메소드 탐색, 호출 그래프 구성, 호출 그래프를 바탕으로 유입 메소드와 사용 메소드 간의 경로 탐색, 검출된 보안 약점 분석 순으로 4단계에 걸쳐 보안 약점을 분석한다. 이 방법의 효율성을 측정하기 위해 정적 분석기를 적용한 새로운 새싹을 이용하여 두 가지 실험을 실행하였다. 첫 번째 실험으로서 이전 연구의 새싹과 개선된 새싹을 Java 시큐어 코딩 가이드를 기준으로 비교하였다. 두 번째 실험으로써 개선된 새싹과 Java 취약점 분석 도구인 FindBugs와 비교하였다. 결과에 따르면, 개선된 새싹은 이전 버전의 새싹보다 15% 더 안전하고 개선된 새싹의 F-measure는 68%로써 FindBugs의 59%인 F-measure와 비교해 9% 포인트 증가하였다.

키워드 : 시큐어 코딩, 한글 프로그래밍 언어, 안전한 프로그래밍 언어, 새싹

1. 서 론

클라우드, 모바일, 빅데이터, SNS, IoT 등 새로운 플랫폼의 등장은 소프트웨어의 고도화의 원인 중 하나로 꼽히고 있다[1]. 세계적으로 소프트웨어 시장 규모는 점점 성장하고 있으며, 새로운 제품과 서비스의 혁신을 주도하고 있다. 하지만 새롭고 다양한 소프트웨어의 등장에 따라 보안 사고가 발생하는 비율도 증가하고 있다. 대표적인 보안 사고로 국외에서는 페이스북(Facebook)의 소프트웨어 버그에 의한 개인정보 유출 사고, 해커 그룹의 소니 픽처스(Sony Pictures) 공격에 인한 보안사고가 있다[2, 3]. 그리고 국내에서는 은행과 통신사의 개인 정보 유출 사고가 있다[4].

※ 이 논문은 2013년도 정부(교육부)의 재원으로 한국연구재단의 지원을 받아 수행된 기초연구사업임.
† 준 회 원 : 부산대학교 전기전자컴퓨터공학과 석사과정
†† 준 회 원 : 부산대학교 전기전자컴퓨터공학과 박사과정
††† 종신회원 : 부산대학교 전기컴퓨터공학과 교수/LG 스마트제어센터
Manuscript Received : February 18, 2016
Accepted : March 24, 2016
* Corresponding Author : Gyun Woo(woogyun@pusan.ac.kr)

이러한 보안 사고를 예방하기 위해 침입 방지 시스템, 침입 차단 시스템 등 다양한 보안 방법이 적용되고 있다. 하지만 가트너의 보고서에 의하면, 소프트웨어 보안 사고는 소프트웨어의 취약성에 의해서 75% 발생한다[5]. 또한 국립표준기술연구소(NIST)의 연구 결과에 의하면 제품 출시 후 결함 수정 비용은 설계 단계에서의 결함 수정 비용보다 30배 이상 많다고 한다[6]. 이러한 문제를 해결하기 위해 개발 단계에서부터 보안 약점을 제거하는 방법인 시큐어 코딩(secure coding)이 주목받고 있다.

시큐어 코딩은 소프트웨어 개발 과정 중 소스 코드 구현 단계에서 보안 약점을 배제하기 위한 보안 방법이다. 국외의 CERT[7], OWASP[8], CWE[9] 등의 기관에서는 시큐어 코딩 가이드를 제시하기 위한 연구를 진행하고 있다. 국내에서는 행정자치부에서 시큐어 코딩 가이드와 소프트웨어 보안 약점 진단 가이드를 제시하고 있다. 또한, 항공기나 자동차 등의 산업에서는 JSF(Joint Strike Fighter)[10], MISRA C[11] 등의 코딩 가이드를 제시하여 안전한 소프트웨어 개발을 위한 노력을 기울이고 있다.

이전 연구에서 우리는 한글 프로그래밍 언어 새싹을 개발하였다[12]. 그리고 행정자치부에서 제시한 Java 시큐어 코딩 가이드의 보안 약점을 재분류하고 프로그래밍 언어 수준에서 보안 약점을 해결하기 위해 예약어를 제공하였다[13]. 그리고 새싹은 행정자치부에서 제시한 Java 시큐어 코딩 가이드의 보안 약점 83개 중 33개를 해결하였다.

본 논문에서는 해결하지 못한 보안 약점 중 API를 사용하여 발생하는 보안 약점을 해결하는 방안을 제안한다. API에 의한 보안 약점은 API의 메소드 형식이 정해져 있기 때문에 프로그래밍 언어 차원에서는 해결하기 어렵다. 이에 우리는 보안 약점이 발생할 수 있는 API를 조사하고, 정적 분석 방법을 통해 조사한 API를 사용하여 데이터를 조작하거나 사용한 경우 보안 약점으로 판단한다. 제안한 방식의 성능 평가를 위해 보안 약점 분석기 FindBugs와 비교 실험하였다.

본 논문의 구성은 다음과 같다. 2장에서는 보안 약점 표준 연구 및 보안 약점 분석 도구를 소개하며 3장에서는 분석기 설계에 관하여 설명한다. 4장에서는 개선된 새싹의 안전성을 평가한다. 마지막으로 5장에서 결론을 맺는다.

2. 관련 연구

2.1 보안 약점 표준 연구

CERT(Computer Emergency Response Team)는 인터넷 사용 시에 나타나는 문제들로 인한 필요성 때문에 1988년 11월에 DARPA(The Defense Advanced Research Projects Agency)에 의해 구성되었다[7]. 시큐어 코딩 표준을 2006년에 생각해 처음으로 CERT C 시큐어 코딩 표준을 2007년 발표하였다. CERT 시큐어 코딩 표준은 다른 기관의 보안 약점 표준과 다르게 C, C++, Java 등 언어별 시큐어 코딩 표준을 제공하여 안전한 프로그램을 개발할 수 있도록 한다.

OWASP(The Open Web Application Security Project)는 오픈 소스 웹 애플리케이션 보안 프로젝트이다[8]. 주로 웹에 관한 정보 노출, 악성 파일 및 스크립트, 보안 취약점 등을 연구하며, 10대 웹 애플리케이션의 취약점(OWASP TOP 10)도 선정하여 발표하고 있다. OWASP TOP 10은 웹 애플리케이션 취약점 중에서 빈도가 많이 발생하고, 보안상 영향을 크게 줄 수 있는 것들 10가지를 선정하여 2004년부터 3년을 주기로 발표하고 있다[14].

CWE(Common Weakness Enumeration)는 소프트웨어 보안 및 품질 강화를 위해 개발 시에 참고할 수 있도록 전 세계 소프트웨어 취약점을 표준화한 목록이다[9]. 미국 국토안전부와 미국 국립 표준기술연구소 산하 비영리 기관인 MITRE에서 후원하고 있다. CWE/SANS Top 25는 일반적인 수행 환경을 위한 주요 보안 약점 목록으로서 2009년부터 25개의 주요 보안 약점을 선발하여 발표하고 있으며, 2011년에 가장 최근 버전이 발표되었다[15]. 2011년에는 중요 보안 약점의 선별을 위하여 중요도 정량평가 방법인 CWSS(Common Weakness Scoring System)가 사용되었다.

2.2 취약점 분석 도구

취약점 분석 도구는 소프트웨어의 보안 약점을 분석하는 도구다. 보안 약점을 해결하기 위해 국내외에서 다양한 취약점 분석 도구가 연구되고 있다. 국외에서 개발한 취약점 분석 도구로는 Fortify[16], Coverity Code Advisor[17], AppScan[18], FindBugs[19, 20] 등이 있다. Fortify는 캘리포니아에 기반을 둔 소프트웨어 보안 업체인 Fortify에서 개발한 취약점 분석 도구다. 2010년에 HP에서 인수하여 현재 HP 기업 보안 상품의 일부로 기업 고객을 위한 제품과 서비스를 제공한다. Coverity Code Advisor는 Coverity에서 개발한 취약점 분석 도구로 정적 분석과 동적 분석으로 C, C++, Java로 작성된 소스 코드의 결함 및 보안 취약점을 분석한다. AppScan은 1988년 애플리케이션 방화벽으로 시작되었는데 초기 이름은 AppShield였다. 현재는 IBM을 통해 웹 애플리케이션 보안 테스트 및 모니터링 서비스로 제공되고 있다. FindBugs는 오픈소스 정적 분석기로 Java의 소스 코드가 아닌 바이트 코드에서 작동한다.

국내에서 개발한 취약점 분석 도구는 BigLook[21]. Code-Ray[22], SecurityPrism[23], SPARROW SCE[24] 등이 있다. BigLook은 이븐스타에서 개발한 취약점 분석 도구로 C, C++, Java 등 다양한 언어의 정적 분석을 지원한다. Code-Ray는 트리니티 소프트에서 개발한 취약점 분석 도구로 기능은 BigLook과 비슷하다. SecurityPrism은 지티원에서 개발하고 SPARROW SCE는 파수닷컴에서 개발한 취약점 분석 도구로 시맨틱(Semantic) 기반의 정적 분석 기술을 적용하여 다양한 언어를 지원한다.

이처럼 다양한 취약점 분석 도구가 개발되었다. 하지만 취약점 분석 도구는 C나 Java와 같이 일반적으로 많이 사용하는 프로그래밍 언어만 지원한다. 그래서 새싹과 같이

새롭게 개발된 프로그래밍 언어는 기존의 취약점 분석 도구를 사용해 취약점을 분석할 수 없다. 그러므로 새싹의 보안 약점을 분석하기 위해서는 새싹용 취약점 분석기가 필요하다. 이에 본 논문에서는 정적 분석 방법을 사용하여 새싹의 보안 약점을 분석하는 취약점 분석기를 개발한다.

3. 분석기 설계

새싹은 프로그래밍 언어 수준에서 보안 약점을 해결하기 위해 제안된 프로그래밍 언어이다[25]. 새싹에서는 새로운 프로그래밍 언어의 단점인 라이브러리의 부족을 해결하기 위해 Java 라이브러리를 사용할 수 있도록 하고 있다. 그래서 새싹에서는 Java와 마찬가지로 API에 의한 보안 약점이 발생할 수 있다. API는 사용하는 형식이 정해져 있고 API의 코드 내부를 확인하기 어려우므로 프로그래밍 언어 수준에서 문법적으로 API에 의한 보안 약점을 해결하기 어렵다. 그러므로 본 논문에서는 보안 약점이 발생할 수 있는 Java API를 조사하여 위험한 메소드 목록을 구성하고 오염 분석[26]을 통해 보안 약점을 분석한다. 구체적으로 위험한 메소드 호출 탐색, 호출 그래프 구성, 유입 메소드와 사용 메소드 간 경로 탐색, 보안 약점 분석 순으로 4단계 분석 과정을 수행한다.

3.1 API의 위험한 메소드 목록

API의 메소드를 잘못 사용할 경우 보안 약점이 발생할 수 있다. 예를 들면, 데이터 통신에서 수신한 데이터를 검증 없이 사용하면 사용한 기능에 따라 사용자가 원치 않는 동작을 할 수 있다. 그러므로 우리는 보안 약점이 발생할 수 있는 API의 메소드를 조사하고 오염된 데이터를 유입하는 API와 오염된 데이터 사용하는 API로 분류한다. 오염된 데이터 유입 API는 Table 1과 같이 데이터 통신의 수신, 파일 읽기 등과 같이 외부에서 데이터를 가져오는 API다. 오염된 데이터 사용 API는 Table 2와 같이 데이터 통신의 전송, DB 나 파일의 쓰기 등 데이터를 사용하는 API다.

Table 1. Tainted Data Inflowing API

Category	Tainted Data Inflowing API Name
Network	java.net.URL.set()
	java.io.InputStreamReader.read()
	java.io.InputStream.read()
	java.io.BufferedReader.readLine()
File	java.util.Properties.getProperty()
	java.io.FileInputStream.read()
	java.io.FileReader.read()
System	java.lang.System.getenv()
	java.lang.System.getProperties()

Table 2. Tainted Data Using API

Category	Tainted Data Using API Name
Network	java.net.Socket.connect()
	java.io.OutputStream.write()
	java.io.OutputStreamWriter.write()
File	java.io.FileOutputStream.write()
	java.io.FileWriter.write()
System	java.lang.Runtime.exec()
DB	java.spl.DriverManager.Connection()
	java.sql.PreparedStatement.executeQuery()
	java.sql.Statement.execute()

오염된 데이터 유입 API를 네트워크, 파일, 시스템으로 분류한다. 그리고 오염된 데이터 사용 API는 네트워크, DB, 파일, 시스템으로 분류한다. 네트워크는 데이터 통신과 관련된 API의 메소드다. 그리고 파일은 파일의 데이터를 조작하는 API의 메소드고 시스템은 시스템의 정보를 가져오거나 시스템 명령어를 수행하는 API의 메소드다. 마지막으로 DB는 DBMS를 조작하는 API의 메소드다.

3.2 위험한 메소드 호출 탐색

소프트웨어를 구성하는 모든 클래스에서 오염된 데이터 유입 메소드와 오염된 데이터 사용 메소드가 있는지 탐색한다. 각각의 메소드 코드가 위험한 메소드에 대한 호출을 포함하는지 탐색한다. Fig. 1의 소스 코드로 예를 들면, 소스 코드에서는 파일에 있는 데이터를 읽어오기 위해 오염된 데이터 유입 메소드인 Properties.getProperty()를 사용한다. 그리고 DB에 연결하기 위해 오염된 데이터 사용 메소드인 DriverManager.getConnection()을 사용하고 작성한 쿼리문을 실행하기 위해 Statement.execute()를 사용한다.

```
void insertData() throws IOException, SQLException {
    streamFileput = new FileInputStream(filePath);
    properties.load(streamFileput);
    data = properties.getProperty("data");
    dbConnection = DriverManager.getConnection(url, user, pwd);
    sqlStatement = dbConnection.createStatement();
    Boolean result = sqlStatement.execute("insert into users "
            + "(status) values ('updated') where name='" + data + "'");
}
```

Fig. 1. Source Code Including Vulnerability

소스 코드에서 사용한 위험 메소드를 탐색하고 탐색한 메소드를 오염된 데이터 유입 메소드 리스트와 오염된 데이터 사용 메소드 리스트에 분리해 각각의 리스트에 저장한다. Fig. 1의 소스 코드로 예를 들면, 위험한 메소드인 Properties.getProperty()를 오염된 데이터 유입 메소드 리스트에 저장한다. 그리고 DriverManager.getConnection()와 Statement.execute()를 오염된 데이터 사용 메소드 리스트에 저장한다. Table 3은 Fig. 1의 소스 코드에서 사용한 위험한 메소드를 저장한 각각의 리스트다.

Table 3. Dangerous API List

Tainted Data Inflowing API	Tainted Data Using API
Properties.getProperty()	Statement.execute()
	DriverManager.getConnection()

3.3 메소드 호출 그래프 구성

저장한 리스트의 메소드를 바탕으로 호출 그래프를 구성한다. 호출 그래프는 오염된 데이터 사용에 접근하는 메소드 리스트를 순회하며 구성한다. 본 논문에서는 소프트웨어 전체에 대한 호출 그래프를 생성하지 않는다. 소프트웨어 전체에 대한 호출 그래프를 생성하면 탐색 범위가 넓어 분석 성능이 떨어질 수 있기 때문이다. 따라서 호출 그래프는 오염된 데이터를 사용한 메소드를 루트로 하고 호출 그래프를 작성한다. Fig. 2는 Fig. 1의 소스 코드를 바탕으로 구성된 호출 그래프다.

3.4 유입 메소드와 사용 메소드 간 경로 탐색

호출 그래프를 바탕으로 오염된 데이터 유입 메소드 노드와 오염된 데이터 사용 메소드 노드를 잇는 경로를 탐색한다. 경로 탐색 알고리즘은 Dijkstra의 최단 경로 탐색 알고리즘을 사용한다. 경로를 찾으면 코드 배열을 구성한다. 코드 배열은 호출 그래프에서 찾은 경로에 있는 메소드들의 코드 나열이다. 코드 배열은 보안 약점 검출에서 보안 약점인지 판단하는 데 사용된다.

단순히 호출 그래프만으로 유입 메소드와 사용 메소드 간 경로 탐색이 어렵다. 그러므로 클래스의 동적, 정적 속성에 대한 접근을 고려한다. Fig. 1의 코드를 예로 들면, Properties.getProperty()를 사용해 파일의 데이터를 변수 data에 할당한다. 그리고 Statement.execute()에서 변수 data를 이용한 쿼리문을 작성하고 실행한다. 변수에 의해 오염된 데이터가 이동할 수 있다.

3.5 보안 약점 분석

보안 약점 검출은 코드 배열을 바탕으로 분석한다. 분석기는 애플리케이션 코드 내에 존재하는 유출 가능한 경로에 대해 병렬적으로 보안 약점 분석을 실행한다. 보안 약점 검출은 메소드에 따라 단계별로 진행되고 코드 배열 내의 모든 메소드를 실행했을 때 마지막에 오염 데이터 사용 메소드에 오염된 데이터가 전달되는지 확인한다.

보안 약점을 분석할 때 반복문은 반복 횟수에 따라 데이터의 값이 변할 수 있다. 하지만 반복문의 반복 횟수만큼 실행하면 분석 시간이 오래 걸릴 수 있다. 또한, 반복 횟수를 동적으로 할당해 반복 횟수를 알 수 없거나 무한 루프와 같은 문제가 발생할 수 있다. 분석기는 데이터의 값이 변하지 않는 고정점(fixed point)을 찾을 때까지만 실행한다. 그리고 무한 루프와 같이 고정점이 없을 수 있는데, 이때는 5번 실행한 결과를 바탕으로 분석한다. 한 연구에 따르면 대표적인 흐름 그래프의 평균 깊이는 약 2.75이다[27]. 그리고 수행해야 하는 반복 횟수의 상한은 통상 흐름 그래프의 깊이에 2를 더한 값이라는 사실[28]이 알려져 있다. 즉, 평균적인 반복 횟수는 약 5번이라고 볼 수 있다.

3.6 해결한 보안 약점

이전 연구에서 제안한 시큐어 코딩 가이드의 항목 중 플랫폼과 라이브러리가 있다[25]. 플랫폼은 시스템 정보 유출이나 시스템 API 사용에 의한 보안 약점이다. 라이브러리는 시스템 API가 아닌 다른 API 사용에 의한 보안 약점이다.

플랫폼 보안 약점의 예로는 운영체제 명령어 삽입, 시스템 정보 노출 등등이 있다. 해결한 플랫폼 보안 약점 중 운영체제 명령어 삽입에 관하여 설명한다. 운영체제 명령어 삽입은 적절한 검증 절차를 거치지 않은 입력값이 운영체제 명령어의 일부 또는 전부로 구성되어 시스템 동작 및 운영에 악영향을 미치는 보안 약점이다. Fig. 3은 Java에서 발생할 수 있는 운영체제 명령어 삽입 예제 코드다.

```
Properties props = new Properties();
String fileName = "file_list";
FileInputStream in = new FileInputStream(fileName);
props.load(in);
String version = props.getProperty("dir_type");
String cmd = new String("cmd.exe /K \"rmanDB.bat \"");
Runtime.getRuntime().exec(cmd + " c:\\prog_cmd\\" + version);
```

Fig. 3. OS Injection in Java

Fig. 3은 외부값인 파일의 dir_type 값을 cmd.exe 명령어를 통해 rmanDB.bat의 인자로 사용하는 예제다. dir_type 값이 사용자가 의도한 값이 아닐 경우 비정상적인 동작을

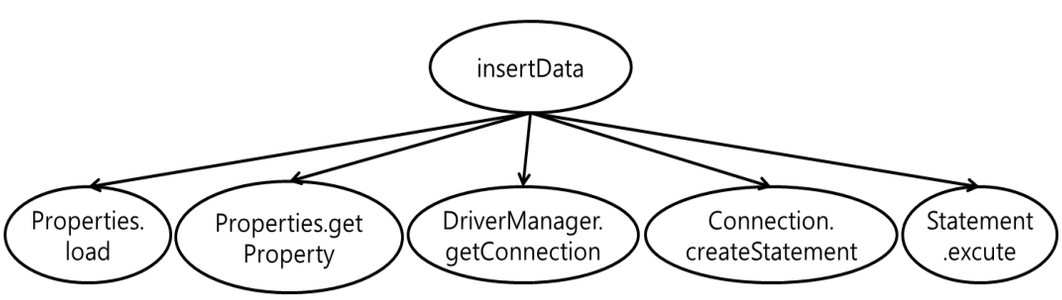

Fig. 2. Method Call Graph

수행할 수 있다. 새싹은 분석기를 통해 데이터 유입 메소드 호출인 pros.getProperty()와 데이터 사용 메소드 호출인 Runtime.getRuntime().exec()를 탐색한다. 그리고 호출 그래프를 구성하고 경로를 탐색한 후 보안 약점 분석을 통해 운영체제 명령어 삽입을 탐지할 수 있다. 자원 삽입을 포함한 플랫폼 보안 약점을 분석기를 통해 탐지할 수 있다.

라이브러리 보안 약점의 예로는 SQL 삽입, 민감한 데이터 노출 등등이 있다. 해결한 라이브러리 보안 약점 중 SQL 삽입에 관해 설명한다. SQL 삽입은 DB와 연동된 애플리케이션에서 공격자가 입력 형식 및 URL 입력란에 SQL 문을 삽입하여 DB로부터 정보를 열람하거나 조작하는 보안 약점이다. Fig. 4는 Java에서 발생할 수 있는 SQL 삽입 예제코드다.

```
String tableName = props.getProperty("jdbc.tableName");
String name = props.getProperty("jdbc.name");
String query = "SELECT * FROM '" + tableName +
        "' WHERE Name = '" + name + "'";
stmt = con.prepareStatement(query);
rs = stmt.executeQuery(query);
```

Fig. 4. SQL Injection in Java

Fig. 4는 외부로부터 tableName과 name 값을 입력받아 검증 없이 바로 SQL 문을 생성하는 예제 코드다. name 값으로 name' OR 'a'='a를 입력하면 SQL 문의 조건이 성립하게 되므로 tableName에 있는 모든 정보를 획득할 수 있다. 새싹은 운영체제 명령어 삽입 보안 약점과 같이 분석기를 통해 데이터 유입 메소드 호출인 pros.getProperty()와 데이터 사용 메소드 호출인 stmt.excuteQuery()를 탐색한다. 그리고 호출 그래프를 구성하고 경로를 탐색한다. 보안 약점 분석을 통해 SQL 삽입을 분석할 수 있다.

4. 안전성 평가

이 장에서는 정적 분석기가 탑재된 개선된 새싹의 안전성을 평가하기 위해 시행한 두 번의 실험에 관하여 설명한다. 처음에는 이전 연구에서 프로그래밍 언어 수준에서 보안 약점을 해결한 새싹과 개선된 새싹을 비교한다. 두 번째는 취약점 정적 분석기인 FindBugs와 개선된 새싹을 비교한다.

4.1 분석기 유무에 따른 성능 비교

이전 연구의 새싹과 개선된 새싹의 안전성을 비교하기 위해 이전 연구에서 제안한 시큐어 코딩 가이드[25]를 바탕으로 해결한 보안 약점의 수를 비교한다. 제안한 시큐어 코딩 가이드의 보안 약점은 행안부에서 제안한 Java 시큐어 코딩 가이드를 바탕으로 구성하였다.

비교 결과는 Table 4와 같이 이전 연구의 새싹은 83개의 보안 약점 중 33개를 해결하였다. 그리고 개선된 새싹은 83개의 보안 약점 중 46개를 해결하였다. 분석기를 통해 플랫폼 보안 약점 2개와 라이브러리 보안 약점 11개를 해결해 총 13개를 해결하였다. 즉 개선된 새싹은 해결 취약점 개선 측면에서 이전 연구의 새싹보다 15% 안전하다고 볼 수 있다. 전체적으로 시큐어 코딩 가이드에 따라 점검해야 하는 취약점 수가 55%나 줄어들었다.

Table 4. Evaluation of Previous Saesark and Improved Saesark

Category	Number of Vulnerabilities	Number of Vulnerabilities Resolved	
		Previous Saesark	Improved Saesark
Language	8	8	8
Compiler	5	5	5
Platform	5	0	2
Library	16	0	11
Logic	49	20	20
Total	83	33	46

4.2 개선된 새싹과 FindBugs(Ver. 3.0) 비교

FindBugs는 정적 분석기로 Java의 보안 약점을 분석한다. FindBugs를 선택한 이유는 오픈 소스로 상용화된 취약점 도구와 달리 공개되어 있기 때문이다. 그리고 FindBugs는 새싹과 같이 JVM에서 동작하기 때문이다. 비교에 사용할 테스트 코드로는 CWE/SANS Top 25[13]의 항목별로 Juliet(Ver. 1.2)[29, 30]과 CWE에서 제공하는 예제 코드를 참고하여 Java와 새싹으로 각각 보안 약점 코드 2개와 정상 코드 1개를 직접 작성하였다. 단, CWE/SANS Top 25의 항목 중 3, 20, 23번은 C, C++에서만 발생하는 보안 약점으로 새싹과 Java에서는 발생하지 않기 때문에 제외하였다. 그러므로 테스트 코드는 보안 약점 코드 44개와 정상 코드 22개로 총 66개다.

테스트 코드는 Fig. 5와 Fig. 6과 같이 작성하였다. Fig. 5의 소스 코드는 보안 약점 항목 중 잠재적으로 위험한 함수 사용을 Java로 작성한 보안 약점 코드다. Java에서 잠재적으로 위험한 함수로 정해진 Math.random()을 사용하고 있다. Fig. 6의 소스 코드는 Fig. 5의 소스 코드를 새싹으로 작성한 것이다.

```
public class UseOfPotentiallyDangerousFunction {
    public static void main(String[] args) {
        Dice dice = new Dice();
        System.out.println(dice.roledice());
    }
}
class Dice {
    public int roledice() {
        return (int)(Math.random() * 6 + 1);
    }
}
```

Fig. 5. Use of Potentially Dangerous Function in Java

Table 5. Evaluation of FindBugs and Improved Saesark

Category		FindBugs(Ver. 3.0)		Improved Saesark	
		Positive	Negative	Positive	Negative
Test Code	True	20	24	26	18
	False	4	18	7	15
Recall		45%		59%	
Precision		83%		79%	
F-measure		58%		67%	

```
@제한 난수()
시작(문자열[] 값) {
    주사위 주사위1 = 새 주사위();
    출력("%수\끝", 주사위1.굴리기());
}
클래스 주사위 {
    공개 정수 굴리기(){
        반환 (정수)(난수() * 6) + 1;
    }
}
```

Fig. 6. Use of Potentially Dangerous Function in Saesark

비교 실험은 작성한 Java 테스트 코드 66개를 FindBugs에서 분석을 수행한다. 그리고 새싹 테스트 코드 66개를 새싹에서 분석을 수행한다. 수행 결과를 재현율(recall)과 정밀도(precision), F-measure를 비교한다. 비교 결과는 Table 5와 같다.

Table 5에서 볼 수 있는 것처럼 보안 약점 코드 44개 중 FindBugs에서는 20개를 검출한 데 비해, 새싹은 24개를 검출하였다. 즉, FindBugs의 재현율은 45%인데 비해 새싹의 재현율은 59%인 것을 확인하였다. 새싹이 FindBugs보다 재현율이 높은 이유는 프로그래밍 언어 수준에서 보안 약점을 해결하였기 때문으로 보인다.

FindBugs의 정밀도는 83%인데 비해 새싹의 정밀도는 79%로 다소 낮았다. 새싹이 FindBugs보다 정밀도가 낮은 이유는 새싹에서 정상인 코드 3개를 보안 약점이 있는 코드로 판단했기 때문이다. 그러나 이는 보안 측면에서는 더 강화된 경고를 주는 것으로 생각할 수 있다. 재현율과 정밀도의 조화평균인 F-measure는 FindBugs가 59%, 새싹이 68%로 나타났다.

5. 결론 및 고찰

우리는 이전 연구에서 해결하지 못한 보안 약점 중 API에 의한 보안 약점을 정적 분석기를 통해 해결한다. 먼저, 위험한 API를 조사해 유입 메소드와 사용 메소드로 분류한다. 그리고 4단계에 걸쳐 분석한다. 먼저, 위험한 메소드를 탐색하고 저장한다. 그리고 저장한 메소드를 바탕으로 호출 그래프를 구성한다. 호출 그래프의 경로를 탐색하고 보안 약점 분석을 통해 보안 약점을 탐지한다.

개발한 개선된 새싹의 안전성을 평가하기 위해 두 번의 실험을 시행하였다. 처음에는 이전 연구의 새싹과 비교를 하고 두 번째는 FindBugs와 비교하였다. 첫 번째 실험의 결과, 이전 연구의 새싹은 83개의 보안 약점 중 33개를 해결한데 비해 개선된 새싹은 46개를 해결하였다. 개선된 새싹은 이전 연구의 새싹보다 위험 코드 검출률이 15% 포인트 증가하였다. 두 번째 실험으로는 결함 코드 44개를 포함한 테스트 코드 66개를 FindBugs와 새싹에서 수행하였다. 그 결과 FindBugs의 F-measure는 59%이고 새싹의 F-measure는 68%다. 즉, 새싹의 위험 코드 검출률이 9% 포인트 증가했다고 볼 수 있다.

향후 연구로는 FindBugs보다 정밀도를 높이기 위한 연구를 생각해 볼 수 있다. 새싹은 CWE/SANS Top 25의 항목에서 위험한 종류의 파일 업로드, 무결성 검사 없이 코드 내려받기와 같이 파일의 위험성과 관련된 보안 약점을 해결하지 못하였다. 그리고 크로스 사이트 스크립팅과 크로스 사이트 요청 위조와 같이 웹 서버에서 발생하는 보안 약점을 해결하지 못하였다. 마지막으로 잘못된 권한 부여를 해결하지 못하였다. 해결하지 못한 보안 약점을 해결하기 위해 다른 정적 분석 방법이나 동적 분석 방법을 사용하는 방안도 생각할 수 있다. 그리고 다른 프로그래밍 언어와 비교해 새싹의 안전성에 대한 성능 평가를 생각할 수 있다.

References

[1] MSIP, SPRI, *Software Industry Annual Report*, 2014.
[2] I. H. Kim, Facebook users private information leaked six million people [Internet], http://news.inews24.com/php/news_view.php?g_serial=754079&g_menu=020600.
[3] A. Buncombe, “Sony Pictures hack: US intelligence chief says North Korea cyberattack was ‘most serious’ ever against US interests,” *The Independent*, 2015.
[4] S. W. Lee, “Study on the information system aduit check list for enhanced privacy,” MS. dissertation, Konkuk University, Seoul, ROK, 2015.
[5] T. Lanowitz, “Now is the time for security at the application level,” Gartner, 2005.

[6] G. Tassey, “The economic impacts of inadequate infrastructure for software testing,” National Institute of Standards and Technology, RTI Project 7007, 2002.
[7] J. McManus and D. Mohindra, The CERT Sun Microsystems Secure Coding Standard for java [Internet], http://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=34669015.
[8] OWASP, Welcome to OWASP [Internet], https://www.owasp.org/index.php/Main_Page.
[9] CWE, A community Developed Dictionary of Software Weakness Types [Internet], http://cwe.mitre.or/.
[10] JSF, The F-35 Lightning II Program [Internet], http://www.jsf.mil/.
[11] MISRA, The Motor Industry Software Reliability Association [Internet], http://www.misra.org.uk/.
[12] J. S. Cheon, D. H. Kang, and G. Woo, “A Concise Korean Programming Language Sprout,” *Journal of KIISE*, Vol.42, No.4, pp.496-503, 2015.
[13] D. H. Kang, Y. E. Kim, and G. Woo, “A Study on Improving Runtime Safety of a Sprout through Analysis of Java Secure Coding Guide,” *Proc. of the KIISE Korea Computer Congress 2015*, pp.1751-1753, 2015.
[14] OWASP, “OWASP Top 10 - 2013,” The Ten Most Critical Web Application Security Risks, 2013.
[15] B. Martin, M. Brown, A. Paller, and D. Kirby. “2011 CWE/SANS top 25 most dangerous software errors,” Common Weakness Enumeration, 2011.
[16] HP, IT Security in the Idea Economy [Internet], https://www.hpe.com/us/en/solutions/security.html.
[17] Coverity, Coverity Software Testing Platform [Internet], http://www.coverity.com/products/.
[18] IBM, IBM Security AppScan [Internet], http://www-03.ibm.com/software/products/en/appscan.
[19] FindBugs, FindBugs because it’s easy [internet], http://findbugs.sourceforge.net/findbugs2.html.
[20] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Q. Zhou, “Evaluating static analysis defect warnings on production software,” *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering,* ACM, pp.1-8, 2007.
[21] Evenstar, BigLook is the financial and enterprise security weaknesses SW diagnostic system optimized for enterprise environments [Internet], http://www.evenstar.co.kr/index-.php.
[22] Trinitysoft, The Trinitysoft is committed to providing the best Web application security solutions [Internet], http://www.trinitysoft.co.kr/page/solution_04.
[23] GTONE, SecurityPrism is secure coding solution to ensure safe application since the early stages of development [Internet], http://www.gtone.co.kr/main/ag/sp.php.
[24] Fasoo, SPARROW is a source code analysis tool, using static analysis [internet], http://www.fasoo.com/site/fasoo/source-codeanalysis/sparrow.do.
[25] Y. E. Kim, J. W. Song, and G. Woo, “A Design of a Korean Programming Language Ensuring Run-Time Safety through Categorizing C Secure Coding Rules,” *Journal of KIISE*, Vol.42, No.4, pp.487-495, 2015.
[26] V. B. Livshits and M. S. Lam, “Finding Security Vulnerabilities in Java Applications with Static Analysis,” *Usenix Security*, pp.18-18, 2005.
[27] D. E. Knuth, “An empirical study of FORTRAN programs,” *Software: Practice and Experience*, Vol.1, No.2, pp.105-133, 1971.
[28] A. V. Aho, R. Sethi, and J. D. Ullamn, “Compilers: Principles, Techniques, and Tools,” 2nd ed., PEARSON, 2014.
[29] T. Boland and P. E. Black, “Juliet 1.1 C/C++ and Java test suite,” *Computer*, Vol.10, No.45, pp.88-90, 2012.
[30] NIST and NSA CAS, Juliet Test Suite for Java and C/C++ [Internet], https://samate.nist.gov/SRD/testsuite.php.

강 도 훈
e-mail : dhkang@pusan.ac.kr
2014년 동서대학교 컴퓨터공학과(학사)
2014년～현 재 부산대학교 전기전자컴퓨터공학과 석사과정
관심분야 : Korean Programming Language, Functional Language

김 연 어
e-mail : yeoneo@pusan.ac.kr
2010년 동아대학교 컴퓨터학과(학사)
2012년 동아대학교 컴퓨터공학과(석사)
2012년～현 재 부산대학교 전기전자컴퓨터공학과 박사과정
관심분야 : Program Analysis, Static Analysis, Program Plagiarism Detection

우 균

e-mail : woogyun@pusan.ac.kr

1991년 한국과학기술원 전산학(학사)

1993년 한국과학기술원 전산학(석사)

2000년 한국과학기술원 전산학(박사)

2000년~2004년 동아대학교 컴퓨터공학과 조교수

2005년~현 재 부산대학교 전기컴퓨터공학과 교수

관심분야 : Program Language, Compiler, Functional Language, Grid Computing, Software Metric, Program Visualization