Journal of KIIT. Vol. 23, No. 2, pp. 241-251, Feb. 28, 2025. pISSN 1598-8619, eISSN 2093-7571
http://dx.doi.org/10.14801/jkiit.2025.23.2.241

보안취약점 탐지를 위한 정적 분석기법 비교 연구

이 은 영*

A Comparative Study on Static Analysis Techniques for Detecting Software Vulnerabilities

Eunyoung Lee*

요 약

소프트웨어 보안취약점은 소프트웨어 시스템 침해 사고를 일으키기 위해서 공격자가 악용할 수 있는 시스템 내부의 약점을 말한다. 대부분의 보안취약점은 소프트웨어 내부의 보안약점으로 인하여 발생하기 때문에 소프트웨어를 실행하기 전에 시스템을 구성하는 소스 코드를 분석하여 코드 내부에 존재하는 보안취약점을 찾아내는 작업은 소프트웨어로 인하여 발생하는 많은 침해사고를 예방하는데 중요한 역할을 한다. 소프트웨어 분석을 위한 다양한 기법 중에서 정적 분석은 소프트웨어가 실행 전에 소스 코드를 이용하여 분석을 수행하는 기법이다. 본 논문에서는 보안취약점을 탐지하기 위한 정적분석 기법 중에서 정보 흐름과 접근 제어의 영역에 대한 기법을 중심으로 최근까지의 연구 동향을 분석하고 앞으로의 연구 방향에 대하여 논의하고자 한다.

Abstract

Software security vulnerabilities are weaknesses within a system that an attacker can exploit to cause a software system intrusion. Most security vulnerabilities are caused by security weaknesses within the software source code. Therefore, it is important to analyze the software that makes up the system before it is executed in order to find the security weaknesses and vulnerabilities that exist within it; the analysis can help prevent many system intrusions caused by the software vulnerabilities. Among the various techniques proposed for analyzing source code, the analysis technique of the source code before the execution is called static analysis. In this paper, the recent trends of the static analysis techniques are analyzed, and future research issues are discussed. The analysis in this paper is focused on the areas of information flow and access control for the discovery of security vulnerabilities.

Keywords
software vulnerability, static analysis, information flow, access control

* 동덕여자대학교 컴퓨터학과 교수
- ORCID: https://orcid.org/0000-0001-8703-9730

· Received: Dec. 12, 2024, Revised: Feb. 06, 2025, Accepted: Feb. 09, 2025
· Corresponding Author: Eunyoung Lee
Dept. of Computer Science, Dongduk Women's University, Korea
Tel.: +82-2-940-4588, Email: elee@dongduk.ac.kr

Ⅰ. 서 론

소프트웨어 시스템의 안전성을 보장하는 작업은 설계 단계부터 구현, 테스팅과 배포 단계까지 개발 단계별로 매우 다양한 이슈를 가지고 있으며, 모든 단계를 포괄하는 종합적인 소프트웨어 개발보안 솔루션을 도출하는 것은 쉽지 않은 작업이다. 실제로 소프트웨어 개발업체들은 잘 알려진 몇 가지 보안 원칙이나 가이드라인에 의지하여 소프트웨어를 설계하고 개발하고 있는 것이 사실이다[1][2].

개발되는 소프트웨어 시스템의 규모가 커짐에 따라 최근의 소프트웨어 개발은 필요한 모든 코드를 구현하기보다는 이미 개발이 완성된 소프트웨어 컴포넌트(Software component)를 목표 시스템 기능에 맞추어 조합하는 방향으로 발전하고 있다[3]. 예를 들어 아마존 웹서비스(AWS, Amazon Web Services)를 통해서 웹 애플리케이션을 개발하는 경우 개발자는 자신의 비즈니스 로직만을 직접 구현해서 웹 서비스를 구성할 수 있다. 애플리케이션 로직을 제외한 부가적인 공통 기능(입력이나 서버 관리, 네트워크 연결 등)은 AWS에서 제공되는 컴포넌트를 활용함으로써 전체 소프트웨어를 개발하는데 드는 시간과 노력을 절약할 수 있게 된 것이다.

문제는 최종 시스템을 구성하는 다양한 소프트웨어 컴포넌트가 서로 다른 보안 기준을 가지고 구현되었을 가능성이 있으며, 결과적으로 소프트웨어 보안의 관점에서 상충되거나 신뢰할 수 없는 상황이 발생할 수 있다는 점이다. 서드파티 컴포넌트(Third-party component)를 사용하는 경우 컴포넌트 API(Application Programming Interface)가 제공하는 보안과 관련된 정보가 충분하지 않다면 결과적으로 서드파티 컴포넌트를 사용하는 최종 시스템에 보안상 위험 요소가 발생할 수 있다.

소프트웨어 보안취약점(Software vulnerability)은 완성된 시스템 내부에 존재하는 보안상의 위험성을 의미한다[4]. 소프트웨어 보안취약점을 발생시키는 요인에는 여러 가지가 있지만 대부분의 보안취약점은 시스템을 구성하는 코드 자체가 보안상 안전하지 않기 때문에 발생한다. 소스 코드 자체에 존재하는 보안상의 위험요소는 소프트웨어 보안약점(Software weakness)이라 부르며, 소스 코드 내부에 숨어 있는 보안약점은 완성된 시스템의 보안취약점을 발생시키는 원인이 된다. 소프트웨어 보안약점은 구현 단계에서 개발자의 실수로 코드에 포함될 수도 있고, 혹은 설계 단계에서의 오류가 원인이 되기도 한다[5]. 특히 서드파티 컴포넌트를 사용하는 경우 개발자는 사용 전에 시스템 구현에 사용할 서드파티 컴포넌트의 보안 안전성에 대해서 확인할 필요가 있다.

개발자는 수동 코드검사(Manual code inspection) 방식으로 시스템 구성에 사용할 소프트웨어 컴포넌트를 직접 검사하여 컴포넌트의 보안 안정성을 검증할 수 있다. 그렇지만 인간에 의한 직접적인 코드 검사는 시간과 비용 면에서 소모적일 뿐만 아니라 커버리지가 완전하지 않다는 치명적인 약점을 가지고 있다. 이 문제를 해결하기 위해서 소스 코드를 자동으로 분석하여 코드 내부에 존재하는 소프트웨어 보안약점을 찾아내는 연구가 꾸준히 진행되고 있다. 코드분석 기법들은 동적분석(Dynamic analysis)이나 정적분석(Static analysis) 방식을 택하거나 2가지 방식을 혼합하는 방식을 취하고 있다. 동적분석 방식은 프로그램을 직접 실행시켜서 프로그램 동작을 관찰하고, 이를 바탕으로 프로그램의 보안상 특징과 보안 취약점을 유추하는 방법이다. 동적분석 방식은 비교적 간단하게 타깃 프로그램 내부에 존재하는 보안취약점을 발견할 수 있다는 장점이 있지만, 보안취약점 발견 커버리지가 충분하지 못할 수 있다는 약점을 가지고 있다[6]. 이에 비해서 정적분석 방식은 주어진 보안 정책에 대해서 가능한 모든 위반 사항을 찾아내는 건전성 분석(Sound analysis)이 가능하다는 장점을 가지고 있지만, 거짓 알람(False alarm)이 많이 발생한다는 단점을 가지고 있다[4].

본 논문에서는 보안취약점을 발견하기 위해서 사용되는 정적분석 기법 중에서 정보 흐름과 접근 제어의 영역을 중심으로 최근까지의 연구 동향을 분석하고자 한다. 2장에서는 보안약점으로 인하여 발생하는 시스템 침해가능성을 무결성과 기밀성의 측면에서 살펴보고, 해당 보안요소를 정보흐름과 접근제어 관점에서 분석한다. 3장에서는 정보흐름을 분석하는 정적분석 기법에 대해서 논의하고, 4장에서는 접근제어 분석 기법을 스택 중심의 방식과 역할 중심의 방식으로 나누어 분석한다.

마지막으로 5장에서는 결론과 향후 추가적인 연구가 필요한 연구방향에 대해서 논의한다.

II. 보안취약점과 보안요소 침해

소프트웨어 시스템 내부의 보안취약점은 주로 시스템의 무결성(Integrity)과 기밀성(Confidentiality)를 침해하는 공격에 악용되기 쉽다. 본 장에서는 컴포넌트로 구성된 시스템에서 시스템 무결성과 기밀성 침해가 발생하는 상황을 분석하고, 이 2가지 보안요소가 정보흐름 및 접근제어와 어떤 연관성을 가지는지 설명한다.

2.1 컴포넌트 보안격자

데이터를 다루는 시스템에서는 중요도에 따라 데이터와 데이터를 처리하는 소프트웨어 컴포넌트 모두 보안 등급을 부여받게 된다. 일반적으로 데이터와 소프트웨어 컴포넌트의 보안 등급은 부분 순서 집합(Partially ordered set)의 형태를 가지게 되며, 특정 조건을 만족하는 경우 이 부분 순서 집합은 격자(Lattice)가 된다[7].

격자의 상한점이 높은 기밀성을, 격자의 하한점이 낮은 기밀성을 나타낸다고 가정하면, 시스템 전체의 기밀성 검증은 어떤 데이터도 높은 보안 레벨에서 낮은 보안 레벨로 흘러가지 않는다는 것을 증명하는 과정으로 귀결된다. 반대로 격자의 상한점이 낮은 신뢰도를, 격자의 하한점이 높은 신뢰도를 나타낸다고 가정하면, 시스템 전체의 무결성 검증은 격자의 상한점에서 하한점으로 어떤 정보도 흘러가지 않는다는 것을 증명하는 과정으로 볼 수 있다.

낮은 보안 레벨의 프로그램에서 높은 보안 레벨의 데이터를 사용하기 위해서 보안 보증(Security endorsement) 방식을 활용하는데, 이는 순간적으로, 혹은 근본적으로 시스템 전체의 보안 레벨을 격하시키는 효과를 가진다. 시스템 구조상 보안 보증이 불가피한 경우, 보안 보증의 범위와 지속 시간을 최소화하는 방향으로 시스템을 설계하고 구현하는 것이 매우 중요하다. 따라서 시스템 설계나 구현 단계에서 프로그램 내부의 정보흐름을 분석하여 코드 안에 내재하는 보안 취약점을 사전에 파악하고 수정하는 과정은 최종 시스템의 안전성을 보장하는 효과적인 방법이 된다.

2.2 무결성 침해

입력 값을 적절하게 검증하지 않거나 혹은 코드 인젝션 결함 등으로 발생하는 보안취약점은 시스템의 무결성을 훼손한다. 소프트웨어 보안의 관점에서 신뢰할 수 없는 소스에서 발생하는 모든 데이터는 시스템 무결성을 해칠 가능성이 있기 때문에 기본적으로 오염된 것으로 간주된다[8].

오염된 데이터와 이를 저장하거나 참조하는 변수들, 이를 다루는 컴포넌트는 낮은 보안 레벨을 부여하여 관리해야 한다. 그렇지만 어떤 경우에는 오염된 컴포넌트가 보안 등급이 높은 시스템 자원에 접근할 필요가 생기기도 한다. 사용자가 입력한 SQL 문을 실행해야 하는 데이터베이스 시스템의 경우가 대표적인 사례이다. SQL 인젝션(Injection)은 악의적인 사용자 입력이 적절한 필터링 과정을 거치지 않은 상태로 주요 시스템 자원에 접근하는 것이 허용되면서 발생한다. 이와 같은 공격을 막기 위해서는 오염 가능성이 있는 데이터가 중요한 자원에 접근하기 전에 반드시 안전성 검사(Sanity check)를 수행해야 한다. 그리고 안전성 검사를 통과한 경우에도 매우 제한적으로 사용되어야 한다[9]. 시스템 방어의 측면에서는 정적분석을 통하여 오염된 데이터에 대한 안전성 검사를 수행하는 코드의 여부를 확인하고, 필요한 경우 검사 코드를 추가하는 방식으로 시스템의 무결성을 유지할 수 있다.

2.3 기밀성 침해

시스템 기밀성 침해는 시스템 내부에서 기밀성 관련 권한을 부여하는 대상에 따라 2가지 측면에서 논의가 가능하다. 첫 번째는 실행되는 코드 자체에 기밀성과 관련된 권한을 부여하는 방식이다. 코드의 기밀성 권한은 모듈 혹은 함수 단위로 부여되기 때문에 코드의 권한을 확인하기 위해서 런타임 호출 스택을 분석하는 방식이 활용된다.

호출 스택을 기반으로 분석이 수행되기 때문에 이와 같은 방식을 스택기반 접근제어(SBAC, Stack-Based Access Control)이라고 부른다. 스택기반 접근제어 시스템에서는 코드가 가지는 권한을 상승시켜서 실행할 수 있는 방식을 허용하는 경우 기밀성 문제가 발생할 수 있다. 스택기반 접근제어 시스템에서 기밀성 유지를 위한 기본 원칙은 다음과 같다. 특정 코드가 자바의 doPriviliged나 .NET의 Assert 키워드를 사용하여 보안 등급을 높인 상태에서 실행되었고, 그 결과로 발생한 값이 있다고 가정하자. 이 경우 결과 값은 특권 코드가 실행된 소프트웨어 컴포넌트 내부에서만 사용되어야 하며, 컴포넌트 밖으로 정보 흐름이 있는 경우에는 정보 흐름에 대한 안전성 검사를 반드시 거쳐야 한다. 이 기본 원칙은 어떤 코드에 실행시 특권을 부여할 것인지 여부를 결정하는 기준으로도 활용될 수 있다.

두 번째 방식은 코드를 실행시키는 주체에 권한을 부여하는 방식이다. 이를 역할기반 접근제어(RBAC, Role-Based Access Control)라고 부르는데 코드 실행자의 권한에 따라 같은 코드라고 할지라도 접근할 수 있는 데이터의 범위가 달라진다.

역할기반 접근제어 시스템에서 기밀성 위반은 주체자-대리자 관계를 정의하는 보안 정책의 오류에서 발생하며, 이 경우 시스템의 기밀성 위반은 바로 무결성 위반으로 연결된다. 주체자-대리자 관계 정책에 따라 특정 사용자의 역할이 한시적으로 상승하는 경우, 상승된 권한으로 실행시킨 코드에서 발생한 모든 데이터는 권한이 상승된 코드 내부에서만 사용되어야 하며, 코드 밖으로 유출되어서는 안 된다.

III. 정보 흐름 분석

3.1 정보흐름분석 기본 원리

정보흐름을 완벽하게 분석하는 문제는 결정 불가능 문제(Undecidable problem)로 알려져 있다[10]. 그렇지만 근사법을 활용하면 프로그램 내부의 정보 흐름을 분석하여 정보흐름 보안취약점을 확인하는 것이 가능하다. 안전하지 않은 정보 흐름을 찾기 위해서 사용하는 정적분석 기법은 프로그램 내부에서 서로 다른 보안 레벨을 가진 변수 사이에 정보흐름이 있는지를 분석한다. 분석을 위하여 프로그램 내부의 각 변수에는 보안 레벨이 부여된다.

만약 변수 x가 변수 y의 값에 영향을 준다면, 변수 x에서 변수 y로 정보 흐름이 있다고 정의한다. 보안적으로 안전한 시스템에서는 변수 x가 속한 보안 레벨에서 변수 y가 속한 보안 레벨로 정보흐름이 시스템 보안정책으로 허용되는 경우에만 가능하다[10]. 시스템 전체의 보안 레벨이 격자로 정의되고 보안 레벨 a에서 레벨 b로 정보흐름이 허용된 상태라고 가정하자. 만약 보안 격자가 기밀성을 모델링하고 있다면, 레벨 a는 레벨 b보다 낮거나 같은 기밀성을 가지고 있어야 한다. 마찬가지로 보안 격자가 무결성을 모델링하고 있다면, 레벨 a는 레벨 b보다 보안 레벨이 높거나 적어도 같은 보안 레벨이어야 한다.

$$\text{if } (x < 0) \ then \ y := 1 \ else \ y := 0 \ endif \qquad (1)$$

프로그램 내부의 변수 사이에 발생하는 정보 흐름은 명시적(Explicit)인 경우뿐만 아니라 묵시적(Implicit)인 경우도 고려되어야 한다. 두 변수 사이에 대입문이 존재하는 경우, 이는 명시적인 정보 흐름에 해당한다. 식 (1)에 예시된 조건문을 살펴보면 조건식에 사용되는 변수 x와 조건 블록에서 사용되는 변수 y 사이에 명시적인 정보 흐름은 존재하지 않는다. 그렇지만 조건에 따라 조건 블록 안에서 변수 y의 값이 달라지기 때문에 이 경우 두 변수 사이에는 묵시적인 정보흐름이 존재한다.

3.2 정보흐름분석 동향

정보흐름 분석에서 널리 사용되는 오염 변수(Tainted variable) 개념은 Perl 언어에 포함된 이후 본격적으로 알려지게 되었다. 오염 변수는 시스템 보안을 훼손하는 정보흐름이 있는지를 탐지하기 위해서 사용된다.

CQual은 변수의 오염 여부를 표시하는 추가적인 타입 한정자(Type quantifier)를 제공한다.

변수 오염을 표시하기 위해서 사용되는 타입 한정자는 tainted와 untainted로, 개발자는 각 변수에 대해서 오염 가능성을 표시할 수 있다. 어떤 변수가 외부 공격자에 의해서 오염될 가능성이 있는 경우에는 tainted 타입으로, 그렇지 않은 경우에는 untainted 타입으로 표시한다. 정보흐름 분석기는 주어진 타입 한정자 정보를 이용하여 제약조건 그래프(Constraint graphs)를 생성하고, 그래프 상에서 tainted 노드에서 untainted 노드로 정보흐름이 있는지 분석하여 정보흐름 안전성을 확인한다[11].

Newsome과 Song은 오염 변수를 런타임에 모니터링하고 분석하는 기법을 제안하였다[8]. 제안된 기법에서는 신뢰할 수 없는 출처에서 전달된 값이나 이 값을 기반으로 계산된 값을 모두 오염된 값으로 간주하고 런타임에 오염된 변수에 대한 모니터링을 수행하였다.

Enck 등은 Newsome이 제안한 동적인 오염변수 분석 기법을 안드로이드 플랫폼에 적용한 TaintDroid 프레임워크를 제안하였다. TaintDroid는 정보흐름 분석을 활용하여 스마트폰 서드파티 앱에서 접근하는 개인 정보를 실시간으로 추적하고 보호하는 것이 가능함을 보였다[12].

Shivakumar는 정보 흐름을 고려한 프로그래밍 언어 FaCT를 제안했는데, FaCT 컴파일러는 프로그램 내부의 정보 흐름을 분석하여 정해진 보안 규칙을 벗어나는 정보 흐름을 찾아낼 수 있다[13]. 프로그래머는 예외적인 경우 declassify 구성자(Construct)를 활용하여 의도적으로 정보를 유출시킬 수 있는데, 이 방법을 통해서 정보흐름 분석에서는 비밀로 간주되는 암호문이나 퍼블릭 암호화키를 외부로 공개할 수 있다. 또한 Shivakumar는 FaCT를 활용하여 프로그래머의 의도와는 다르게 원하는 않는 정보를 유출하는 Spectre-PHT 공격에 대한 PoC(Proof of Concept)를 제시하였다.

Bocci는 클라우드 에지 환경에서 서버리스 애플리케이션을 서버리스 에지 환경에 효과적으로 할당하는 기법을 제안하였다[14]. 제안된 기법에서는 가장 적합한 서버리스 에지 서버를 찾기 위해서 정보흐름 분석을 활용하였으며, 이를 통하여 사이드 채널을 통한 정보유출 방지를 시도하였다.

FLARE는 보안성을 강화한 분산 데이터 분석 프레임워크이다[15]. 분산 환경에서 데이터를 처리할 때 발생할 수 있는 정보의 유출을 막고 분석 연산의 기밀성과 무결성을 유지하기 위해서, 제안된 기법은 분산 노드에 차별적으로 신뢰도를 부여하고 이를 정보흐름 기법을 통하여 분석하였다.

표 1에서는 주요 정보흐름 분석 연구를 지원 프로그래밍 언어와 런타임 모니터링 지원 여부 등의 특징으로 정리하였다.

표 1. 보안취약점 검출을 위한 정적분석 기법 특징 비교
Table 1. Comparison of static analysis techniques on software vulnerabilities

Work	Analysis area	Analysis strategy	Language	Runtime support	Description
Foster et al. [2002]	Information flow	Tainted data	C	No	
Newsome and Song [2005]	Information flow	Tainted data	x86	Yes	
Enck et al. [2014]	Information flow	Tainted data	Java	Yes	Android support
Shivakumar et al. [2023]	Information flow	Covert channel	FaCT	No	
Bocci et al. [2023]	Information flow	Covert channel	Prolog	Yes	FaaS orchestration
Li et al. [2023]	Information flow	Tainted data	Java bytecode	Yes	
Wallach et al. [2000]	Access Control	SBAC	Java	No	SPS calculus
Felt et al. [2011]	Access Control	SBAC	Java	No	Android support
Pottier et al. [2005]	Access Control	SBAC	Java	No	λ_{sec} calculus
Bartoletti et al. [2001]	Access Control	SBAC	Java	Yes	
Jackson [2002]	Access Control	RBAC	Alloy	No	Alcoa analyzer
Sullivan et al. [2017]	Access Control	RBAC	Alloy	No	Extension to Alloy

Ⅳ. 접근 제어 분석기법

소프트웨어 시스템은 접근 제어에 대한 정책을 수립하고 이를 런타임에 감시할 필요가 있다. 보안에 민감한 데이터 사용과 시스템 작동이 모두 접근 제어의 대상에 포함된다. 소프트웨어 시스템은 사용자가 속한 그룹을 기준으로 민감 자원에 대한 접근을 제어한다. 서로 다른 신뢰 수준을 가진 소프트웨어 컴포넌트로 이루어진 시스템에서 접근 제어가 제대로 준수되고 있는가를 확인하는 것은 상당히 복잡한 작업이다. 만약 접근제어 정책이 사용자에게 충분한 권한을 부여하지 못한다면 사용자는 필요한 자원에 접근할 수 없게 된다. 반대로 접근 제어 정책이 사용자에게 필요 이상의 권한을 부여하는 경우, 이는 시스템 침해 사고로 이어질 수 있다.

다양한 접근제어 기법 중에서 가장 인기를 끌고 있는 방식은 스택기반 접근제어 방식과 역할기반 접근제어 방식이다. 2가지 방식은 상호 보완적인 성격을 가지고 있으며, 자바 런타임 시스템이나 .NET CLR과 같은 상용 시스템에서는 2가지 방식을 동시에 사용하고 있다.

본 장에서는 컴포넌트 기반 아키텍처에서 활용 가능한 2가지 접근제어 기법을 간단히 비교하고, 이를 구현하는 정적 분석 연구의 최신 동향을 분석한다. 표 1에서는 주요 접근제어 분석 연구를 지원 프로그래밍 언어와 런타임 모니터링 지원 여부 등의 특징으로 정리하였다.

4.1 접근제어기법 비교

스택기반 접근제어는 1997년도에 처음 제안된 이후 사실상 프로그래밍 언어를 활용한 접근제어 기법의 표준으로 자리잡고 있다[16]. 스택기반 접근제어를 사용하는 런타임 시스템은 프로그램 코드가 제한된 자원에 접근하려고 할 때마다 호출 스레드가 위치하는 런타임 스택상의 모든 호출자(Caller)가 접근허가(Permission) 조건을 충족시키는지를 체크한다. 스택기반 접근제어는 신뢰도가 낮은 코드가 신뢰도가 높은 코드를 호출함으로써 제한된 자원에 대한 접근 권한을 얻는 것을 방지하기 위하여 제안되었다. 스택기반 접근제어 기법에서는 소프트웨어 컴포넌트별로 허용된 접근허가를 선언적인 방식으로 데이터베이스에 저장하고 관리한다. 예를 들어 자바 SE에서 시스템 관리자가 어떤 자바 JAR 파일에 FilePermission을 부여하면, 해당 JAR 파일 내부에 정의된 모든 클래스는 런타임에 동일한 권한을 가지게 된다[17].

역할기반 접근제어는 사용자가 조직에서 가지는 역할에 기반을 두고 보안 정책을 세우고 접근 제어를 수행하는 방식이다. 사용자가 애플리케이션의 특정 엔트리 포인트를 실행하는 상황을 가정해 보자. 역할기반 접근제어를 사용하는 경우 사용자는 해당 엔트리 포인트를 지나서 순차적으로 접근할 가능성이 있는 모든 동작에 필요한 역할을 가지고 있어야만 한다[18].

그림 1은 역할기반 접근제어의 작동 원리를 개념적으로 설명하고 있다. 구현된 시스템 내부에서 엔트리 포인트 m_0의 실행은 다른 컴포넌트 혹은 같은 컴포넌트에 속한 모듈에 대한 순차적인 메소드 호출로 이어지며, 그림에서 노드 사이 에지는 호출 관계를 표현하고 있다. 역할기반 접근제어에서 각 모듈에는 해당 모듈을 실행하기 위해 필요한 역할이 부여된다. 예시된 시스템에는 customer, employee, manager의 3가지 역할이 존재하고 manager가 가장 권한이 높은 역할을, customer가 가장 권한이 낮은 역할을 나타낸다. 주어진 시나리오에서 사용자 Alice는 'employee' 역할을 부여받은 상태에서 엔트리 포인트 m_0을 실행하려고 한다. 그렇지만 이 경우 사용자 Alice의 시도는 역할기반 접근제어 규칙을 위반하게 되어, 실행이 불가능하다. 주어진 시나리오에서 m_0에서 실행을 시작하는 경우 사용자 Alice는 콜 체인을 통해서 모듈 m_{10}부터 모듈 m_3까지 간접적으로 실행하게 된다. 이 경우 사용자 Alice가 가지고 있는 'employee' 역할로 'customer' 역할 권한이 필요한 모듈 m_0, m_{10}을 실행하는 것은 문제가 없지만 모듈 m_{11}을 실행하는데 필요한 'manager' 역할보다는 권한이 낮아 실행이 불가능하기 때문이다.

역할기반 접근제어를 사용하는 경우 사용자에게는 필요 이상의 역할을 부여해서는 안 된다.

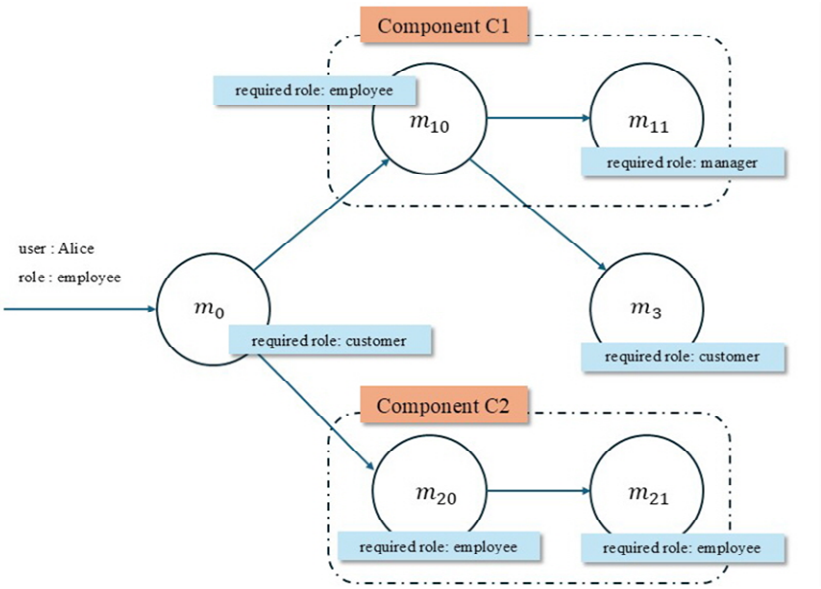


그림 1. 역할기반 접근제어 시나리오
Fig. 1. Scenario of role-based access control

이 법칙은 최소 권한의 법칙이라고 부르는데, 사용자에게 필요 이상의 권한을 부여하는 것은 시스템 침해 사고를 일으킬 수 있는 주요 요인으로 작용할 수 있다. 각 사용자에게 역할과 권한을 부여하는 것은 시스템 관리자의 주된 업무지만, 다수의 컴포넌트로 구성된 복잡한 시스템에서 최소 권한을 분석하는 것은 쉽지 않은 일이다. 소프트웨어 컴포넌트 사이의 콜 패턴을 분석하여 역할기반 최소 권한을 자동으로 분석할 수 있는 정적 분석 도구에 대한 연구는 향후 많은 관심이 필요한 주제라고 판단된다.

4.2 정적분석을 활용한 접근제어기법 동향

시스템 보안에서 자원에 대한 보호는 오랫동안 운영체제 레벨에서 이루어졌다. 그렇지만 애플리케이션 소프트웨어의 규모가 점차 커지고 운영체제의 기능이 복잡해지면서, 운영체제만으로 애플리케이션 자체의 접근 제어까지 효과적으로 지원하는 것이 어려워졌다. 결과적으로 애플리케이션 수준에서 보안 정책을 수립하고 집행할 필요가 생겼는데, 이는 애플리케이션 개발자에게 적지 않은 부담으로 작용하게 되었다. 이런 상황에서 소프트웨어 전반에 걸친 보안 정책을 구현하고 적용하는 기능을 프로그래밍 언어에 추가하는 언어기반 보안(Language-based security) 연구가 시작되었다[19].

프로그래밍 언어 기반 보안에서 가장 많이 사용하는 기법은 코드 분석(Code analysis)과 코드 재작성(Code rewriting) 방식이다. 개발자 혹은 보안 책임자는 코드 분석을 활용하여 대상 코드에서 보안 정책에 위배되는 부분을 찾아낼 수 있다. 이후 코드 재작성 과정에서 1차 분석에서 찾아낸 안전하지 않은 코드를 보안 정책에 부합하는 코드로 다시 작성한다. 코드 분석과 재작성에 가장 편리하고 안전한 방법은 프로그래밍 언어의 타입 시스템을 활용하는 방법이다[20]. 소프트웨어 컴포넌트를 이용하는 코드 사용자(Code consumer)나 소프트웨어 컴포넌트를 구현하는 코드 제공자(Code provider)는 보안 정책을 구현한 타입 시스템과 컴파일러를 활용하여 코드 분석과 코드 재작성 과정을 보다 쉽게 진행할 수 있다. 타입 시스템을 활용한 보안 방식에서 코드 제공자는 타입 시스템에 부합하는 코드를 작성하고, 코드 사용자는 타입 체킹을 통해서 해당 컴포넌트가 보안 정책을 준수하는지 확인할 수 있다.

접근제어 기법에서도 타입 시스템을 활용한 연구가 꾸준히 진행되고 있으며, 본 절에서는 관련 연구를 스택기반 접근제어와 역할기반 접근제어 방식으로 크게 나누어서 설명하고자 한다.

SAFKASI는 타입 시스템을 활용한 대표적인 보안 아키텍처로 스택기반 접근제어 방식을 지원하고 있다[21]. SAFKASI에서는 접근 제어를 체크하기 위해서 기존의 스택 인스펙션(Stack inspection) 대신 SPS(Security-Passing Style)이라는 이름의 술어논리(Predicate calculus)를 사용한다. SPS는 호출 스택에 존재하는 스레드에 대한 권한을 검증하는 스택 인스펙션을 정형화했다고 볼 수 있다[22].

Pottier 등은 초기 SPS 타입 시스템을 확장한 $\lambda s_e c$를 제안하였다[23]. $\lambda s_e c$는 SPS가 가지는 논리학적 한계를 람다 논리체계(λ-calculus)를 활용하여 향상시켰다는 평가를 받고 있다.

Bartoletti 등은 런타임 접근제어 테스팅 성능을 향상시키는 기법을 제안하였다[24]. 제안된 기법은 런타임 스택을 모델링하기 위하여 연산적 의미론(Operational semantics)을 활용하여 중복되거나 불필요한 접근제어 테스팅을 제거하였다. 또한 접근제어 테스팅을 코드 내부에서 재배치하는 방법을 사용하여 테스팅 효율을 높였다.

역할기반 접근제어를 위한 모델을 세우고 구현하는 과정은 상당히 복잡하고 결과적으로 소프트웨어 아키텍처에 대한 복잡한 분석을 필요로 한다[18]. 이런 이유로 역할기반 접근제어에 대한 연구가 스택기반 접근제어에 대한 연구보다 상대적으로 적게 이루어지고 있는 것이 사실이다.

역할기반 접근제어와 관련된 초기 정적분석 연구로는 RBAC96 접근 모델을 서술하는데 사용되는 Alloy 기술 언어를 들 수 있다[25]. Alloy는 1차 논리(First-order logic)를 기반으로 둔 선언적 언어로 추이폐포(Transitive closure)와 관계함수(Relations)를 지원한다. 사용자는 Alloy로 소프트웨어를 모델링하고, 자동화된 분석기를 사용하여 기술된 모델을 검증한다. Alloy는 복잡한 소프트웨어 특성을 기술할 수 있는 명쾌하고 정확한 방법을 제공하여 인기를 얻었으며, 논리식으로 표현된 모델을 검증하는 자동 분석기 Alcoa에 대한 연구도 동시에 진행되었다[26].

그렇지만 모델링 언어로서 Alloy가 제공하는 표현력과는 별개로 사용자가 기술한 Alloy 모델이 해당 소프트웨어를 제대로 표현하고 있는지를 판별하는 것은 상당히 어려운 일이다. 특히 논리기반 기술언어에 익숙하지 않은 일반 사용자에게는 Alloy 언어를 사용해서 원하는 소프트웨어 속성을 논리식으로 표현하는 자체가 쉽지 않은 작업이다. 결과적으로 Alloy 기술 언어에 관련된 후속 연구들은 도메인에 특화된 소프트웨어 속성을 Alloy 모델로 구현하거나, 소프트웨어 테스팅에 활용하는 방향으로 발전하였다.

Sullivan 등은 소프트웨어 유닛 테스트 개념을 추가한 Alloy 모델과 Alloy 분석기에 제안하였다[27]. 절차형 프로그램(Imperative programs)에서 많이 사용되는 유닛 테스트는 소프트웨어 테스팅에서 매우 중요한 개념이다. 그러나 기존 Alloy 언어에는 유닛 테스트를 기술할 수 있는 툴셋(Tool-set)이 정의되지 않아서 Alloy 모델을 활용하여 절차형 프로그램의 정확도를 테스트하는데 어려움이 있었다. Sullivan은 Alloy 모델에 소프트웨어 유닛 테스트 개념을 추가하여 절차형 프로그램에 대한 테스트 케이스 생성과 검증의 효율을 높였다.

Ⅴ. 결론 및 전망

소프트웨어 보안취약점은 소프트웨어 시스템 침해사고를 일으키기 위해서 공격자가 악용할 수 있는 시스템 내부의 약점을 말한다. 대부분의 보안취약점은 소프트웨어 내부의 보안약점으로 인하여 발생한다. 따라서 시스템을 구성하는 소프트웨어를 실행 전에 분석하여 내부에 존재하는 보안 약점과 취약점을 찾아내는 것은 소프트웨어로 인하여 발생하는 많은 침해 사고를 예방하는 데 중요한 역할을 한다. 정적분석 기법은 소프트웨어가 실행되기 전에 소스코드를 이용하는 보안 안전성을 분석하는 기법이다. 본 논문에서는 보안취약점을 발견하기 위해서 시용되는 정적분석 기법 중에서 정보흐름과 접근제어의 영역에서의 최근까지의 연구 동향을 분석하였다.

접근제어와 정보흐름을 분석하여 소프트웨어 시스템의 보안성을 검증하는 방법은 지금까지 상당한 진전을 보여 왔으며 앞으로도 꾸준한 발전이 기대되고 있다. 최근에는 서드파티 컴포넌트를 활용하는 서버리스 컴퓨팅이나 마이크로 서비스 아키텍처, 기계 학습을 지원하는 다양한 라이브러리 등이 빠르게 성장하면서 소프트웨어 보안에 새로운 이슈들을 발생시키고 있다[28]-[30].

서버리스 컴퓨팅에서 서로 다른 보안 정책과 보안 지원수준을 가진 컴포넌트를 이용하여 오케스트레이션을 진행하는 경우 컴포넌트 사이의 보안성 침해 가능성을 분석하고 침해 사고를 예방할 수 있는 분석 기법에 대한 연구가 향후 중요한 연구 주제로 부상할 것으로 예상된다.

또한 거대언어모델에 대한 관심과 활용 방법에 대한 연구가 활발해지면서 거대언어모델과 기계학습 기법을 소프트웨어 정적분석에 활용하는 연구도 매우 활발하게 진행되고 있으며, 당분간 중요한 연구 주제로 그 위치를 유지할 것으로 예상된다.

마지막으로 암호화나 보안성을 강화한 통신 프로토콜 등 보안과 관련된 기능 역시 외부에서 제공되는 라이브러리나 컴포넌트를 사용하는 빈도가 증가하고 있는데, 이 경우 잘못된 알고리즘이나 안전하지 않은 구현 코드의 사용은 오히려 사용자 애플리케이션에 보안 취약점을 발생시키는 결과를 낳을 수도 있다. 이와 관련된 공급망에서의 안전성 분석에 관한 연구도 추가적인 관심과 연구가 필요한 영역이다.

References

[1] "MISRA C", https://misra.org.uk. [accessed: Jan. 10, 2025].

[2] "CERT Oracle coding standard for Java", https://insights.sei.cmu.edu/library/cert-secure-coding-books/. [accessed: Jan. 15, 2025].

[3] L. Liu, Y. Yao, and J. Li, "A review of the application of component-based software development in open CNC systems", The International Journal of Advanced Manufacturing Technology, Vol. 107, pp. 3727-3753, Apr. 2020. http://doi.org/10.1007/s00170-020-05258-1.

[4] H. Hanif, M. H. N. M. Nasir, M. F. A. Razak, A. Firdaus, and N. B. Anuar, "The rise of software vulnerability: Taxonomy of software vulnerabilities detection and machine learning approaches", Journal of Network and Computer Applications, Vol. 179, pp. 103009, Apr. 2021. https://doi.org/10.1016/j.jnca.2021.103009.

[5] Z. E. Sartabanova, V. T. Dimitrov, and S. M. Sarsimbayeva, "Applying the knowledge base of CWE weaknesses in software design", Journal of Mathematics, Mechanics and Computer Science, Vol. 108, No. 4, pp. 72-80, Dec. 2020. http://doi.org/10.26577/jmmcs.2020.v108.i4.06.

[6] C. Beaman, M. Redbourne, J. D. Mummery, and S. Hakak, "Fuzzing vulnerability discovery techniques: survey, challenges and future directions", Computers & Security, Vol. 120, pp. 102813, Sep. 2022. http://doi.org/10.1016/j.cose.2022.102813.

[7] X. Chen, J. An, Z. Xiong, C. Xing, N. Zhao, F. R. Yu, and A. Nallanathan, "Covert communications: A comprehensive survey", IEEE Communications Surveys & Tutorials, Vol. 25, No. 2, pp. 1173-1198, Apr. 2023. http://dx.doi.org/10.1109/comst.2023.3263921.

[8] J. Newsome and D. X. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software", Proc. of Network and Distributed System Security Symposium, Vol. 5, pp. 1-17, Feb. 2005.

[9] K. Ashcraft and D. Engler, "Using programmer-written compiler extensions to catch security holes", Proc. 2002 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, pp. 143-159, May 2002. http://doi.org/10.1109/secpri.2002.1004368.

[10] M. Pendleton, R. Garcia-Lebron, J.-H. Cho, and S. Xu, "A survey on systems security metrics", ACM Computing Surveys, Vol. 49, No. 4, pp. 1-35, Dec. 2017. http://doi.org/10.1145/3005714.

[11] J. S. Foster, T. Terauchi, and A. Aiken, "Flow-sensitive type qualifiers", ACM SIGPLAN Notices, Vol. 37, No. 5, pp. 1-12, May 2002. http://doi.org/10.1145/543552.512531.

[12] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A.

N. Sheth, "TaintDroid: An information-flow tracking system for realtime privacy monitoring on smartphones", ACM Transactions on Computer Systems, Vol. 32, No. 2, pp. 1-29, Jun. 2014. http://doi.org/10.1145/2619091.

[13] B. A. Shivakumar, J. Barnes, G. Barthe, S. Cauligi, C. Chuengsatiansup, D. Genkin, S. O'Connell, P. Schwabe, R. Q. Sim, and Y. Yarom, "Spectre declassified: Reading from the right place at the wrong time", 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, pp. 1753-1770, May 2023.http://doi.org/10.1109/sp46215.2023.10179355.

[14] A. Bocci, S. Forti, G.-L. Ferrari, and A. Brogi, "Declarative secure placement of FaaS orchestrations in the cloud-edge continuum", Electronics, Vol. 12, No. 6, pp. 1332, Mar. 2023. http://dx.doi.org/10.3390/electronics12061332.

[15] X. Li, F. Li, and M. Gao, "Flare: A fast, secure, and memory-efficient distributed analytics framework", Proc. of the VLDB Endowment, Vol. 16, No. 6, pp. 1439-1452, Feb. 2023. http://doi.org/10.14778/3583140.3583158.

[16] "Overview of Java security models", https://docs.oracle.com/core.1111/introjps.htm. [accessed: Jan. 15, 2025].

[17] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity principles, implementations, and applications", ACM Transactions on Information and System Security, Vol. 13, No. 1, pp. 1-40, Oct. 2009. http://doi.org/10.1145/1609956.1609960.

[18] S. Sicari, A. Rizzardi, L. Grieco, and A. Coen-Porisini, "Security, privacy and trust in Internet of Things: The road ahead", Computer Networks, Vol. 76, pp. 146-164, Jan. 2015. http://doi.org/10.1016/j.comnet.2014.11.008.

[19] M. Patrignani, A. Ahmed, and D. Clarke, "Formal approaches to secure compilation", ACM Computing Surveys, Vol. 51, No. 6, pp. 1-36, Nov. 2019. http://doi.org/10.1145/3280984.

[20] W. Khan, M. Kamran, A. Ahmad, F. A. Khan, and A. Derhab, "Formal analysis of language-based android security using theorem proving approach", IEEE Access, Vol. 7, pp. 16550-16560, Jan. 2019. http://dx.doi.org/10.1109/access.2019.2895261.

[21] D. S. Wallach, A. W. Appel, and E. W. Felten, "SAFKASI: a security mechanism for language-based systems", ACM Transactions on Software Engineering and Methodology, Vol. 9, No. 4, pp. 341-378, Oct. 2000. http://doi.org/10.1145/363516.363520.

[22] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified", Proc. of the 18th ACM conference on computer and communications security, Chicago Illinois USA, Vol. 30, pp. 627-638, Oct. 2011. http://doi.org/10.1145/2046707.2046779.

[23] F. Pottier, C. Skalka, and S. Smith, "A systematic approach to static access control", ACM Transactions on Programming Languages and Systems, Vol. 27, No. 2, pp. 344-382, Mar. 2005. http://doi.org/10.1145/1057387.1057392.

[24] M. Bartoletti, P. Degano, and G. Ferrari, "Static analysis for stack inspection", Electronic Notes in Theoretical Computer Science, Vol. 54, pp. 69-80, Aug. 2001. http://doi.org/10.1016/s1571-0661(04)00236-1.

[25] D. Jackson, "Alloy: A lightweight object modelling notation", ACM Transactions on Software Engineering and Methodology, Vol. 11, No. 2, pp. 256-290, Apr. 2002. http://doi.org/10.1145/505145.505149.

[26] D. Jackson, I. Schechter, and I. Shlyakhter, "Alcoa: the Alloy constraint analyzer", ICSE '00: Proceedings of the 22nd international conference on Software engineering, Limerick Ireland, pp. 730-733, Jun. 2000. http://doi.org/10.1109/icse.2000.870482.

[27] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid, "Automated test generation and mutation testing for Alloy", IEEE International Conference on Software Testing, Verification and Validation (ICST), Tokyo, Japan, pp. 264-275, Mar. 2017. http://doi.org/10.1109/icst.2017.31.

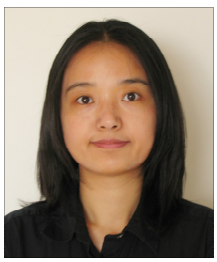
[28] S. Chakraborty, R. Krishna, Y. Ding, and B. Ray, "Deep learning based vulnerability detection: Are we there yet?", IEEE Transactions on Software Engineering, Vol. 48, No. 9, pp. 3280-3296, Sep. 2022. https://doi.org/10.1109/TSE.2021.3087402.

[29] N. Dragoni, S. Giallorenzo, A. L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina, "Microservices: Yesterday, today, and tomorrow", Present and Ulterior Software Engineering, pp. 195-216, Sep. 2017. http://doi.org/10.1007/978-3-319-67425-4_12

[30] J. Wen, Z. Chen, X. Jin, and X. Liu, "Rise of the planet of serverless computing: A systematic review", ACM Transactions on Software Engineering and Methodology, Vol. 32, No. 5, pp. 1-61, Jul. 2023. http://doi.org/10.1145/3579643.

저자소개

이 은 영 (Eunyoung Lee)



1996년 2월 : 고려대학교 전산과학과(학사)
2004년 1월 : Princeton University, USA(전산학박사)
2005년 3월 ~ 현재 : 동덕여자대학교 컴퓨터학과 교수
관심분야 : 소프트웨어 보안, 프로그래밍 언어, 클라우드 컴퓨팅